Elevated Learning Solutions

info@elevatedlearning.com

Introduction to Gradle

Lab Manual

Abstract

This is the companion Lab Manual to the Introduction to Gradle Course.  
  
Note: Gradle is a dynamic tools which continually offers features which are incubating. This means Gradle does not guarantee the will run correctly in every circumstance, and that they may change in future version of Gradle. When these features are used in these labs, they are clearly marked as incubating or in the incubating phase. For more information see section C.12 of the Gradle User’s Guide.

Table of Contents

[1 Install and Verify Gradle Installation (20 min) 4](#_Toc446171830)

[1.1 Gradle Installation 4](#_Toc446171831)

[1.2 Gradle Directory Structure and Verification 7](#_Toc446171832)

[2 Gradle Build Scripts (40 min) 9](#_Toc446171833)

[2.1 GRADLE PROJECT, Task Actions and Properties 9](#_Toc446171834)

[2.2 Gradle Build Life Cycle 12](#_Toc446171835)

[2.3 Action Dependencies in a Single Task 15](#_Toc446171836)

[2.4 Action Dependencies Between Different Tasks 16](#_Toc446171837)

[2.5 Dynamic Tasks 20](#_Toc446171838)

[2.6 Adding New Properties 21](#_Toc446171839)

[3 Gradle Projects (60 min) 24](#_Toc446171840)

[3.1 Creating a Gradle Java Project with Init 24](#_Toc446171841)

[3.2 Using Java Plugin 25](#_Toc446171842)

[3.3 Using the War and Gretty Plugin 28](#_Toc446171843)

[3.4 The Gradle Wrapper 29](#_Toc446171844)

[4 Using Gradle In Eclipse (20 min) 32](#_Toc446171845)

[4.1 Installing the Eclipse Gradle Plugin 32](#_Toc446171846)

[4.2 Import a Gradle Project into Eclipse 33](#_Toc446171847)

[4.3 Create a Gradle Project in Eclipse 35](#_Toc446171848)

[4.4 Executing Gradle Tasks in Eclipse 36](#_Toc446171849)

[5 Creating and Using Gradle Plugins (20 min) 38](#_Toc446171850)

[5.1 Creating a Gradle Script Plugin 38](#_Toc446171851)

[5.2 Creating a Gradle Binary Plugin 40](#_Toc446171852)

[6 Incremental Builds and Multi Projects (50 min) 42](#_Toc446171853)

[6.1 Creating and Executing a Custom Task Class 42](#_Toc446171854)

[6.2 Creating a Custom Incremental Build Task 43](#_Toc446171855)

[6.3 Working with Multiple Projects 46](#_Toc446171856)

[7 External Dependency Management (25 min) 50](#_Toc446171857)

[7.1 Working with External Dependencies and Repositories 50](#_Toc446171858)

[8 Gradle Testing Integration (20 min) 53](#_Toc446171859)

[8.1 Adding and Executing JUnit Tests 53](#_Toc446171860)

[9 Gradle Integration with Jenkins (40 min) 57](#_Toc446171861)

[9.1 Install and Configure Jenkins 57](#_Toc446171862)

[9.2 Adding Git Configuration Management 60](#_Toc446171863)

[9.3 Creating and Executing a Jenkins Build Job Using Gradle 61](#_Toc446171864)

# Install and Verify Gradle Installation (20 min)

Prerequisites

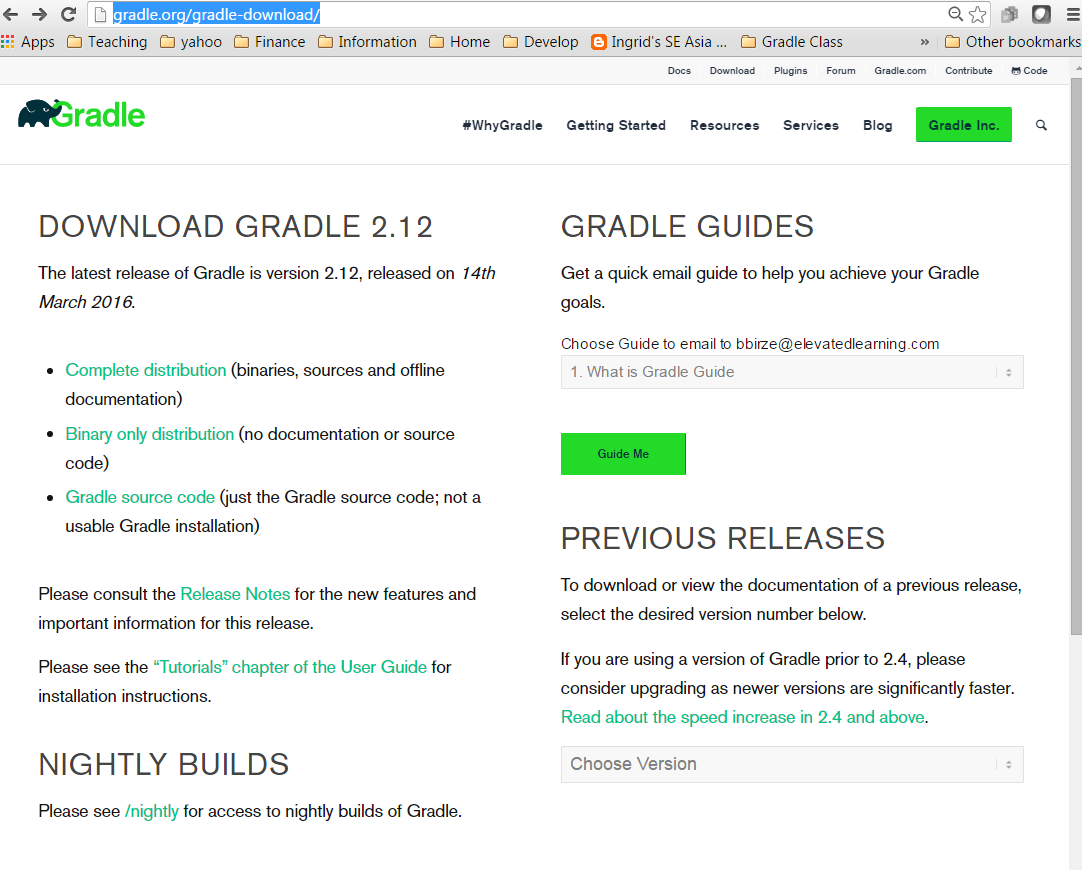
* Java JDK 1.5 or higher, verify installation by opening a command prompt and entering at the command prompt: C:\> java -version
* Internet access

Description

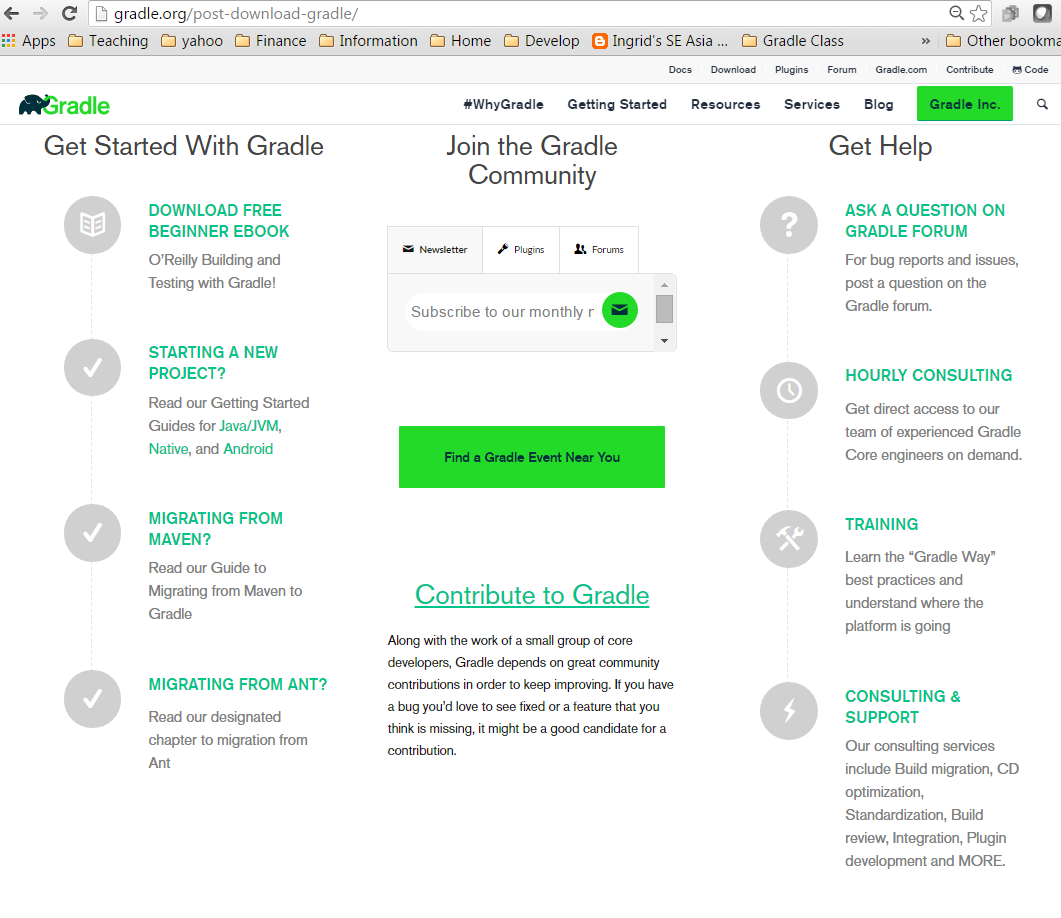
In this lab we will visit the Gradle home page, download and install Gradle, and then verify our installation.

## Gradle Installation

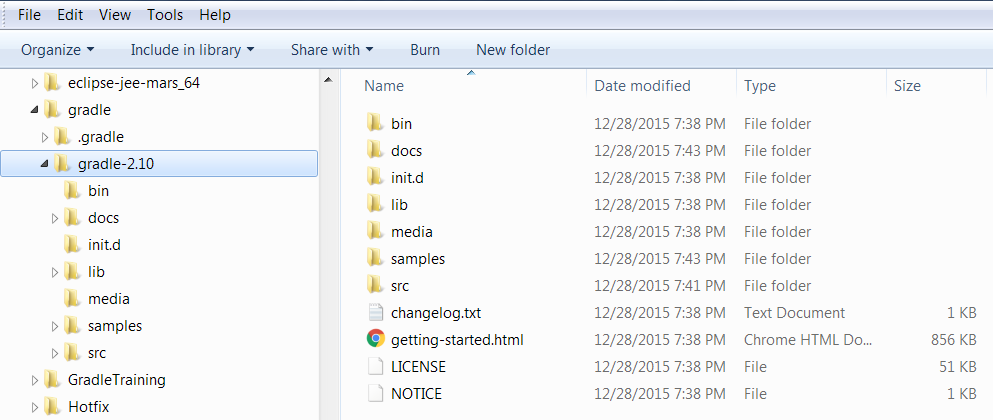
1. Open your browser and go to the page: <http://gradle.org/gradle-download/> . From here you can explore the menu items at the top of the screen. As someone new to Gradle, the items “Getting Started ->Java/JVM” and “Resources: will be of the most interest.
2. Notice the download options on the left hand side of the page. Click on the Complete Distribution to get the binaries, sources, documentation, and samples.



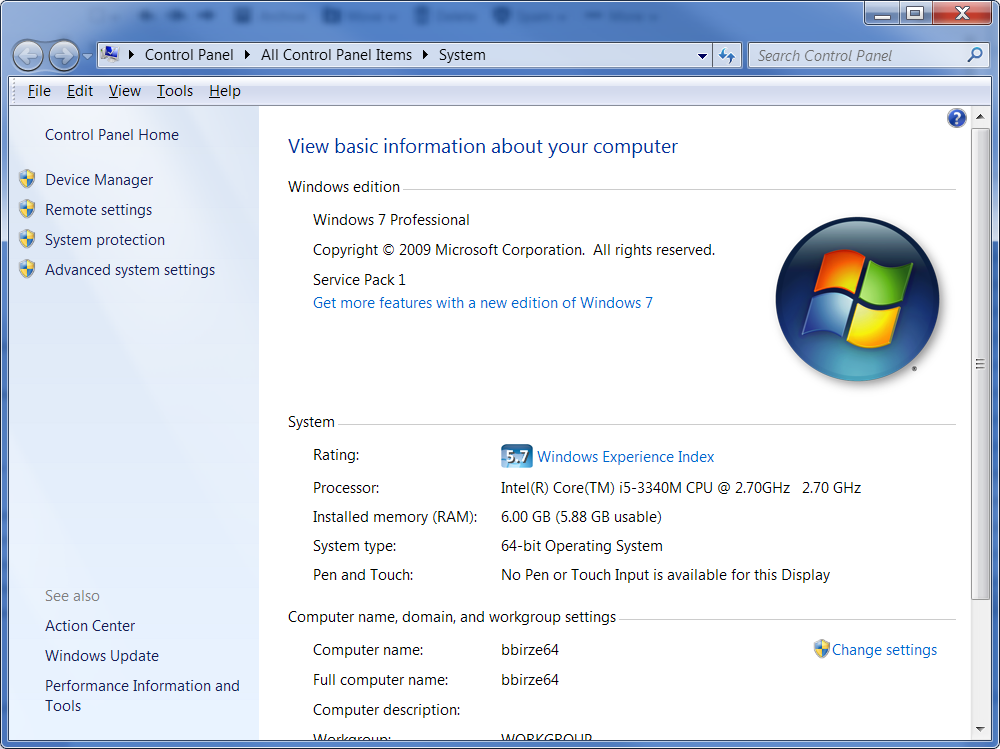
1. This will start the download and take you to a page full of resources for those new to Gradle at <http://gradle.org/post-download-gradle/>. There is even a free download of the O-Reilly book “Building and Testing with Gradle”. You’ll want to bookmark this page when you get back to your desk so you have it handy.



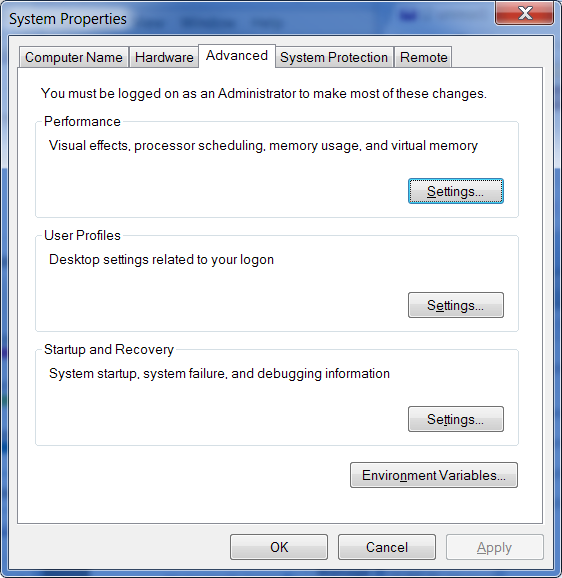
1. Open your File Explorer and go to your download directory and find the Gradle installation zip file. Extract the zip file contents to C:\Gradle. It will create a Gradle subdirectory as shown under C:\Gradle\Gradle-2.10 (Note: the Gradle version in the directory name may be different):



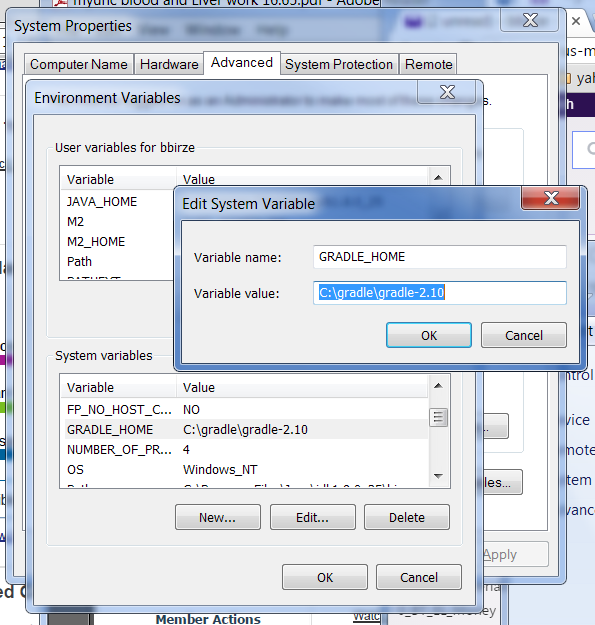
1. Double click on the file Getting-started.html to bring up the Installing Gradle page.
2. To finish installation we have to setup the GRADLE\_HOME environment variable and add it to our path. Click on the Windows Start button, right click on Computer, and select Properties. This brings up the system information windows. Click on Advanced system settings.



1. This will bring up the System Properties window. Make sure you are in the Advanced tab, and click on Environment Variables.

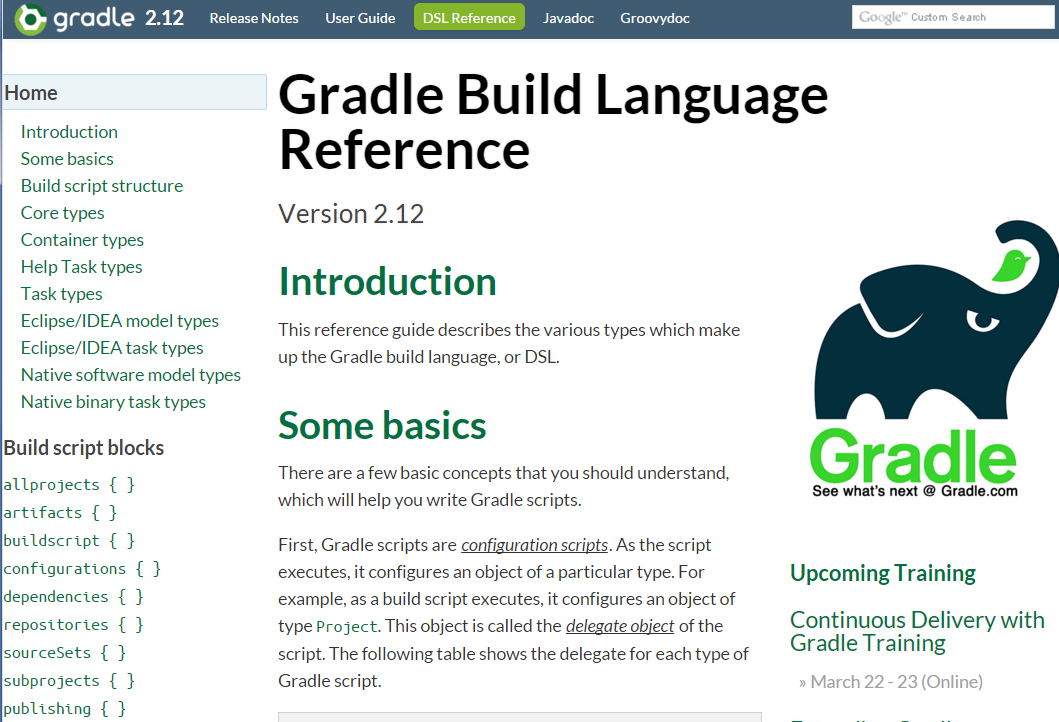


1. Add the environment variable GRADLE\_HOME to the System Variables, set it to the Gradle installation directory path. Then update your path system variable to add the path to %GRADLE\_HOME%\bin;. Be sure to add the forward slash between %GRADLE\_HOME% and   
   “bin”, and separate it from the other items in path with a semi-colon. This is all that’s needed to configure your Gradle installation



## Gradle Directory Structure and Verification

1. Open your file explorer and navigate to the Gradle installation directory, %GRADLE\_HOME%\docs. Here you’ll find 4 directories and release notes. Look through each briefly to get a feel for what is in each. Since Gradle used DDD (Domain Driven Development), it’s not surprising that the Domain Specific Language (DSL) terms are used in all 4 references.
   1. **dsl**: the Domain Specific Language (DSL), or vocabulary used in the Gradle build script. Click on index.html to read the DSL reference guide.
   2. **groovydoc**: Groovy Java Doc for plugin developers and Gradle build masters.
   3. **Javadoc**: Java Doc for plugin developers and Gradle build masters.
   4. **Userguide**: Everything you want to know about Gradle in a well-organized well written document. Click on userguide.html, or userguide.pdf to read the Gradle User’s Guide.
2. These references in your installation can also be accessed on the web. In your web browser go to <https://docs.gradle.org/current/dsl/>. Notice the menu items at the top of the page.



1. To verify your Gradle installation, open a windows command prompt from

Start->All Programs->Accessories->Command Prompt

1. Enter the command: **gradle -version**, which should show you the current Gradle version. Ensure the version matches the version you downloaded. If the gradle command can’t be found, ensure you opened the command window *After* you added your environment variables.

**Expected Result**: System with properly installed Gradle which can be accessed via the command line.

End of Lab

**If you get done Early:** browse the Gradle DSL documentation to understand Gradle script basics and the two core types, Project and Task.

# Gradle Build Scripts (40 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Browser open to the Gradle JavaDoc page.
* Work directory created as C:\GradleTraining\work\Exercise\_2. Note, the solutions are in directory C:\GradleTraining\Exercise\_2.
* Command prompt windows navigated to Exercise\_2 directory

Description

In this lab we will explore the different components in a Gradle build script and the dependencies and interactions between the components. We will also demonstrate which components are accessed in each phase of the Gradle life cycle.

## GRADLE PROJECT, Task Actions and Properties

Though a build.gradle file looks like a script, it is actually a programming language with a powerful API underneath. The two main APIs you will use in your Gradle build scripts are the Project and the Task interfaces, both in the org.gradle.api package. You can find the document for both of these interfaces on the Gradle JavaDoc page open in your browser (see lab PreReqs). The Gradle scripting hides the instances of Task and Project, but they are always available to you in your build scripts. Each Gradle build script is associated with one and only one Project instance and hold a collection of Task instances which perform the work of the build.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_2\Exercise\_2.1
2. In command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following task to the file and save it. Here we are creating an instance of org.gradle.api.Task named firstTask and creating an action, doLast, which will run when we invoke the task from Gradle. We are also setting values to the tasks description and group properties.

task firstTask {

description "Description of firstTask"

group "Lab 2.1 Group

doLast {

println ‘I AM the First Task!’

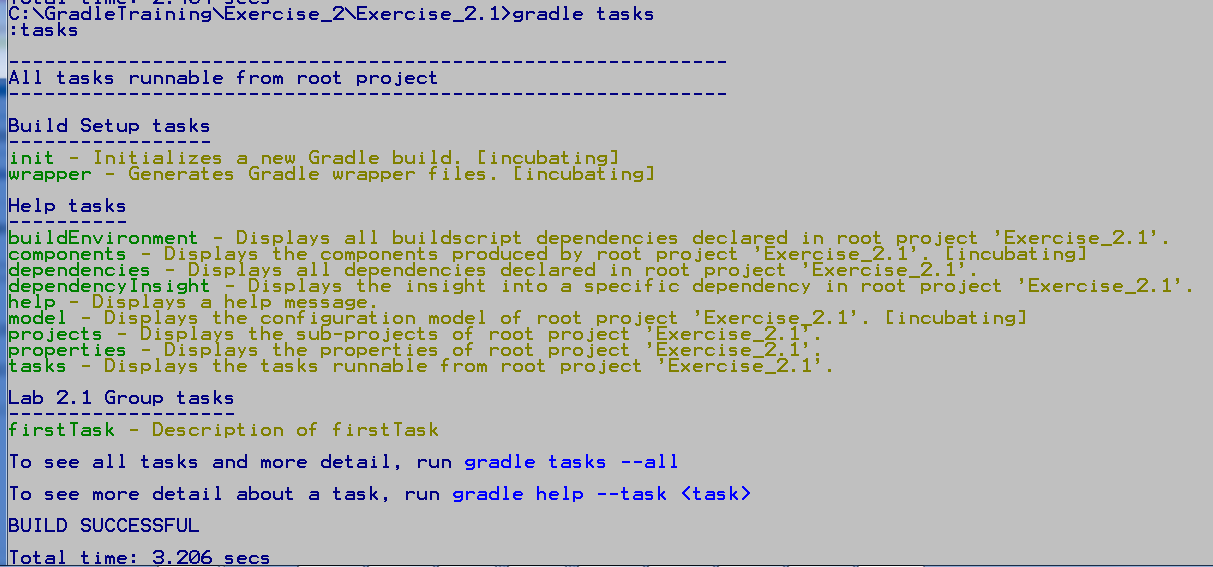
println "\tAccessing Project's properties directly project.projectDir: $project.projectDir"

}

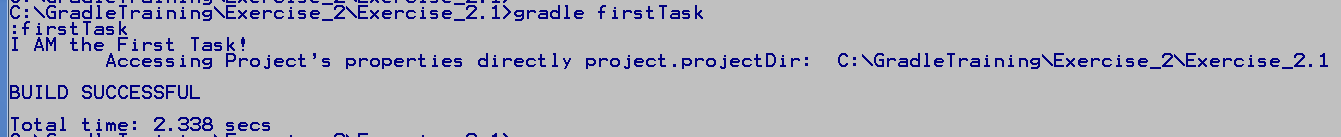
}

**Note** the last println is all on one line, do not insert a <cr>

1. On the command line run the command gradle tasks. This will show you the name and description of all the tasks available to you in this build script. Notice Gradle gives you some tasks by default, and that our task, firstTask, is grouped under the Lab 2.1 Group and our description is shown with the listing.



1. To run the task, enter gradle firstTask, this will execute the code in the doLast action closure, or curly braces {}. Notice we have printed out the Project directory path, accessing it from the Project instance variable projectDir. We did this by enclosing the string in double quotes and putting a ‘$’ before the property we wanted to have evaluated in the string. This turns the string into a Groovy GString which does interpolation on the placeholder when the string is evaluated with toString().



1. Add the following task to the build script below the first one. Here we are creating another instance of Task named secondTask and creating an action, doFirst, which will run when we invoke the task from Gradle.

In the action’s first println is enclosed in single quotes, this makes it a Java String, so we need to escape the quote symbol in I’m. The second println is a Groovy GString in double quotes, but we are not evaluating a method call, not a simple dotted expression. So we need to use the ${…} notation to enclose the expression we want evaluated.

Also note we are setting the task’s description and group properties via methods instead of accessing the property directly. You can access Project and Task instance properties by their name or with the setter and getter methods.

task secondTask {

setDescription "Description of secondTask"

setProperty('group', 'Lab 2.1 Group'

doFirst {

println 'I\'m the second task, I try harder.'

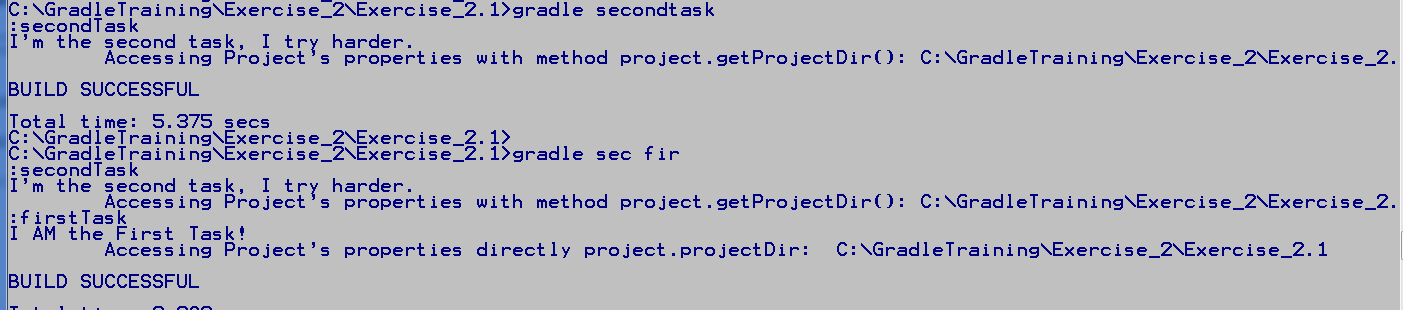
println "\tAccessing Project's properties with method project.getProjectDir(): ${project.getProjectDir()}"

}

}

**Note** the last println is all on one line, do not insert a <cr>

1. On the command line run the command gradle tasks. Notice how secondTask and its description appear grouped under firstTask.
2. To run the task, enter gradle secondTask, this will execute the code in the doFirst action closure, or curly braces {}. Gradle will do task name completion. Enter the command gradle sec fir to see Gradle figure out what task you want run and complete the name for you. Gradle tasks are case in-sensitive as well.



1. Add the following task to the build script below the other 2 tasks. Here we are creating another instance of Task named thirdTask, but this time we are creating an action with the short cut << symbol. The task consists solely of one action, so there is no place to set the task’s group or description.

In the action’s first println we use double quotes, but there is no expression to interpolate, so Gradle uses a Java String. In the second println we are using a Groovy GString and doing interpolation on the task property, name.

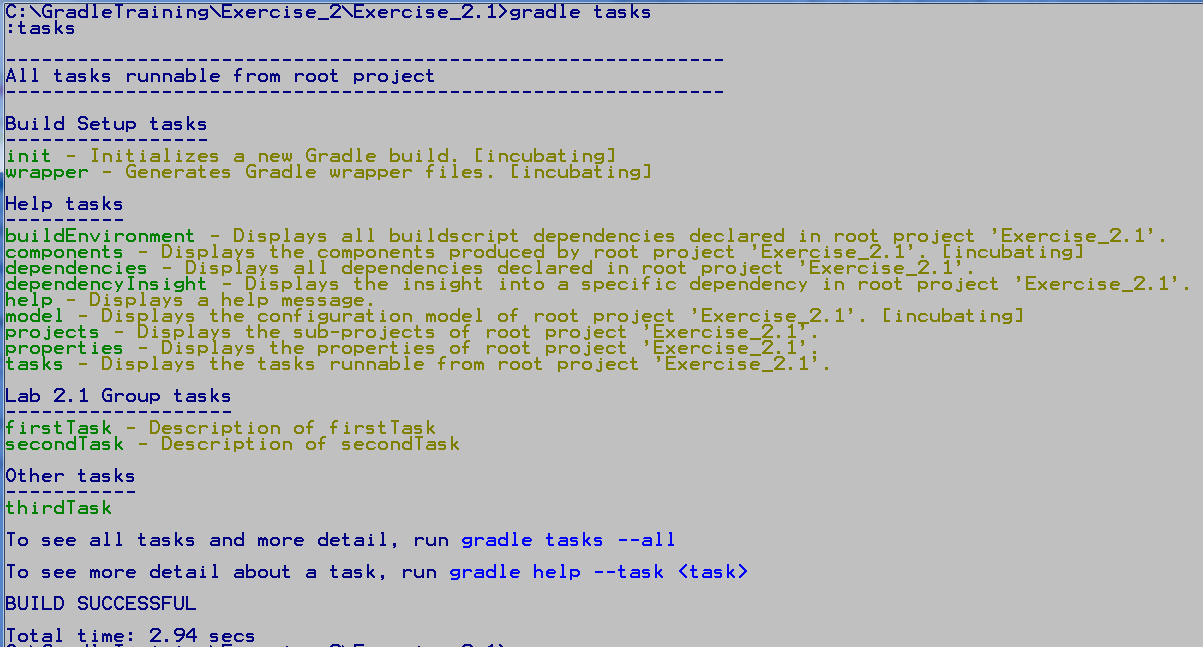
task thirdTask <<

println "I'm the third task, I get pampered!"

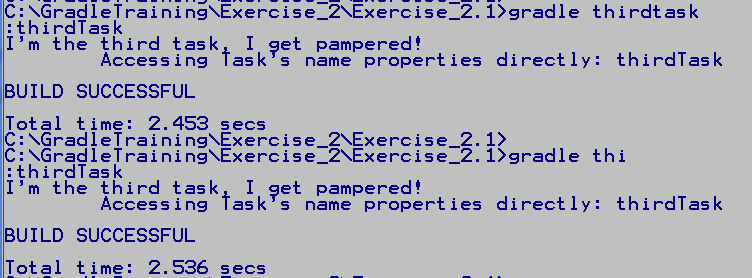
println "\tAccessing Task's name properties directly: $name"

}

1. On the command line run the command gradle tasks. Notice how thirdTask with no set group appears in the Other tasks group, and without a description.



1. To run the task, enter gradle thirdTask, of an abbreviation of thirdTask, so see the action run.



**Expected Result**: build.Gradle script demonstrating using Task and Project properties and executing actions multiple ways.

## Gradle Build Life Cycle

The Gradle Build Life Cycle has 3 phases, Initialization, Configuration, and Execution.

During the Initialization phase Gradle finds all the build scripts in a multi-project build and creates a Project instance for each one. An org.gradle.api.initialization.Settings instance is created to build a hierarchy of Project instances using information in the settings.gradle script. The settings.gradle file is optional if you only have one project in your build. Finally the Settings instance executes each Project’s build script, visiting parent projects before its children.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_2\Exercise\_2.2
2. In command prompt navigate to the Exercise\_2.2 directory.
3. Create a settings.gradle file in this directory and open it in Notepad++.
4. Add the following lines to the file and save it. Here we are printing a few items set during the initialization phase before the build.gradle file is even read.

println 'The settings.gradle file is executed during the initialization phase.'

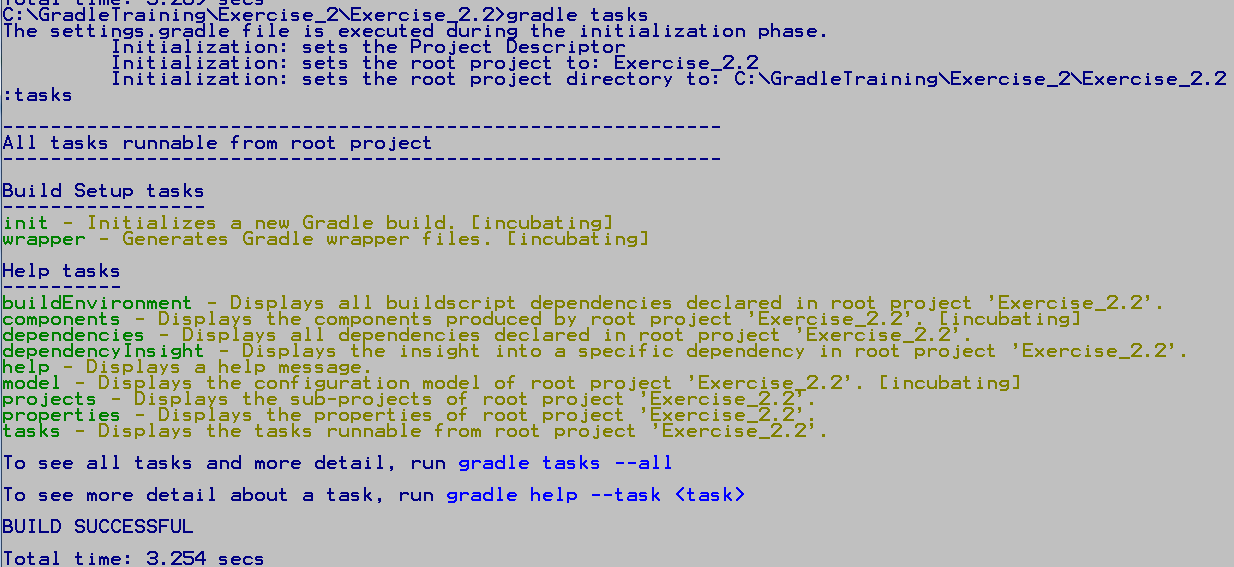
println "\t Initialization: sets the Project Descriptor"

println "\t Initialization: sets the root project to: $rootProject.name"

println "\t Initialization: sets the root project directory to: $rootProject.projectDir"

**Note** these printlns are all on one line, do not insert a <cr>

1. Create a build.gradle file in this directory and open it in Notepad++, but do not add anything to it.
2. On the command line run the command gradle tasks, this will execute the println statements in the settings.gradle file where the Project Descriptor has already been setup. Notice that the initialization phase happens, even though we don’t haven’t specified any tasks in our build file. We are actually asking Gradle to run the task “tasks”, under the Help tasks group.



During the configuration phase Gradle runs through all the build scripts in the project, or projects if it’s a multi-project build, and sets up its DAG (Dependency Acyclic Graph), or it’s dependency graph, and configures all the project objects. This phase is where we can initialize properties and setup our dependencies before any tasks are executed.

1. Go back to the build.gradle file in Notepadd++, add the following lines to the file, and save it. Here we are printing some lines at the top of the script to announce when the configuration phase begins, then adding some println statements to show the initialization that happens during configuration. We also have a single action in task configAndExecute which prints a notice when it executes.

Notice the allprojects followed by a closure {}. Allprojects is part of the org.gradle.api.Project I/F API, it iterates through all the projects in the build giving them one at a time to the enclosed Action. In our case our Action is afterEvalute, also in the Project I/F. Look up these methods in the org.gradle.api.Project JavaDoc to learn more.

println "Configuration Phase Begins for project: $project.name"

println "\tread through build.gradle setting up properties and dependencies"

allprojects {

afterEvaluate { proj ->

println "Configuration Phase Ends for project: $proj.name"

}

}

task configAndExecute {

println "\tconfigAndExecute Task: setting up my group and description properties"

description "Description of configAndExecute task"

group "Lab 2.2 Group"

doFirst {

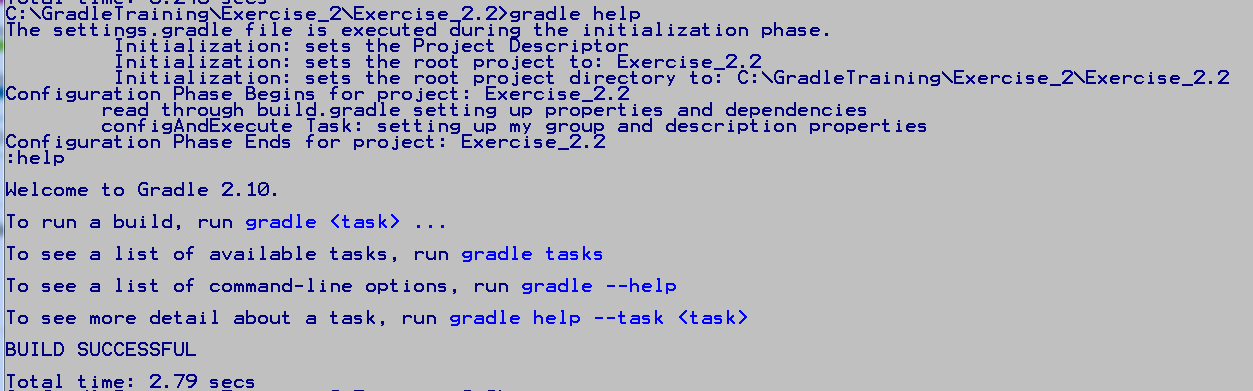
println "configAndExecute Task: Execute Phase, doing work!"

}

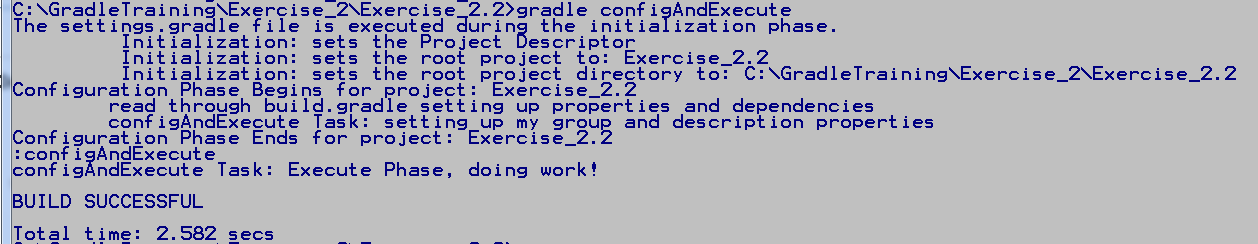
}

1. On the command line run the command gradle help, this will run through all three phases running the help task in the execution phase. Notice the first line in the build.gradle file is the beginning of the configuration phase, then the rest of the script is evaluated. The lines in the Task that are not in the Action are evaluated as well. Then the allprojects afterEvaluate action is run at the end of the configuration phase.

Lastly, the execution phase starts which is signified by the line :help. Whenever Gradle executes a task, it first prints out the name of the task preceded by a colon. When a build has lots of dependencies between tasks, Gradle printing out the task it is executing is very helpful is knowing where you are when the build fails.



1. On the command line run the command gradle configAndExecute to run our task’s action and see it print during the execution phase. Notice the line :configAndExecute is printed at the start of the execution phase to tell us which task Gradle is executing.



**Expected Result**: build.Gradle script demonstrating all three phases of the Gradle build life cycle.

## Action Dependencies in a Single Task

Each task can have many actions, they are all added to the Task’s action list. When the task is executed, it execute each task in its action list. The doFirst method adds a task to the beginning of the task action list, while the doLast method adds a task to the end of the task action list. So doFirst and doLast can be used to roughly control the order which actions within a single task execute.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_2\Exercise\_2.3
2. In command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following task to the file and save it. Here we are adding 4 actions to our task, 2 with doFirst and 2 with doLast.

task firstOrLast {

doLast {

println "doLast - 1: Executing! "

}

doFirst {

println "doFirst - 1: Executing! "

}

doLast {

println "doLast - 2: Executing! "

}

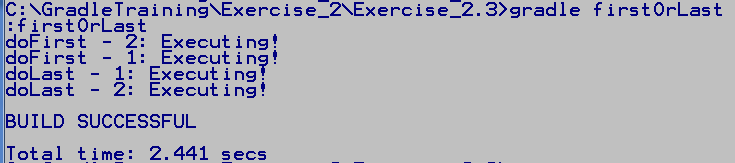
doFirst {

println "doFirst - 2: Executing! "

}

}

1. On the command line run the command gradle firstOrLast. Notice the order the actions are executed is not the order in which they were listed in the task. The actions were added to the task’s action list in the order they were listed in the task. However, doLast actions were added to the end of the action list while doFirst actions were pushed onto the front. The actions were then executed in the order they appeared in the task list, giving the order we see in the output.



**Expected Result**: build.Gradle script demonstrating dependencies in the task’s action list.

## Action Dependencies Between Different Tasks

A Project can contains many Tasks. Sometimes Tasks have dependencies on each other, meaning one Task must run before or after another Task. We can declare dependencies between Tasks with the Task dependsOn method.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_2\Exercise\_2.4
2. In command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following task to the file and save it. Here we are creating two Tasks and specifying needsRunFirst depends on runFirst.

**task runFirst {**

**doLast {**

**println "running task $name \n"**

**}**

**}**

**task needsRunFirst {**

**dependsOn runFirst**

**doLast {**

**println "running task $name \n"**

**}**

**}**

1. On the command line run the command gradle needsRunFirst. This will force the execution of the runFirst Task before the needsRunFirst Task.
2. Now add the follow task to the top of the build.gradle file, before needsRunFirst has been defined. Save the file. Notice you do not need to have a task defined in order to setup a dependency on it. You just need to quote the name of the task. This is very helpful in multi-project builds where tasks can appear in different build.gradle files.

**task runNext {**

**dependsOn "needsRunFirst"**

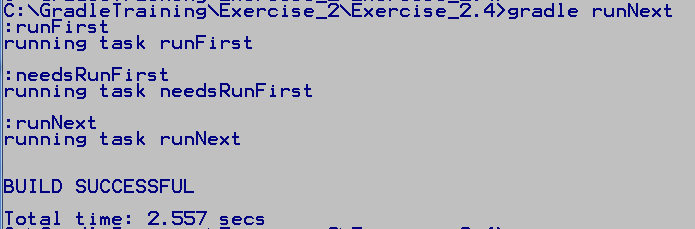
**doLast {**

**println "running task $name \n"**

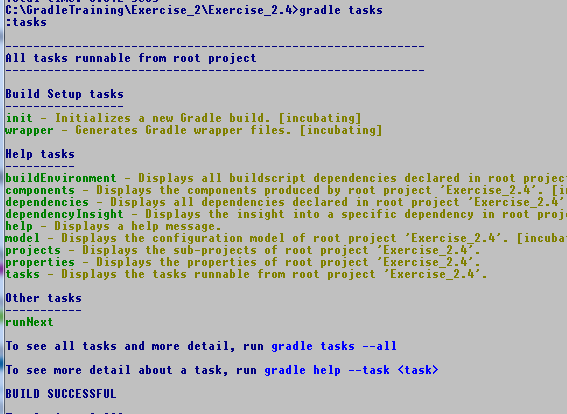
**}**

**}**

1. On the command line run the command gradle runNext. This will force the execution of needsRunFirst, which will force the execute of runFirst

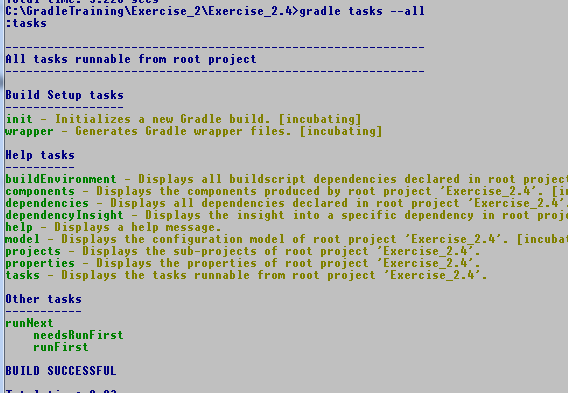


1. Now run the command gradle tasks. Notice that only our runNext task is listed in the Other Tasks grouping



Show only top level tasks

This because Gradle thinks the only reason these tasks exist is for the use of the runNext Task. To see the Tasks available, and their dependencies, use the command gradle tasks --all.



Show all tasks & dependencies

The new finalizedBy method is in the incubating phase. finalizedBy is used to run a task after the finalized task runs, even if the finalized task fails. If the finalized task does not run or is up to date and did not do any work, the finalized task will also consider itself up to date and will not run. finalizedBy is useful for doing cleanup of resources needed for the finalized task.

1. In your build.gradle file, add the following code. Here we are adding two Tasks, the first is starting an application server to test our web application. The second task stops our application server after the test is complete.

task runWebApp {

finalizedBy "closeAppServer"

doLast{

println "runWebApp: Start Application Server "

println "runWebApp: Navigate to app URL \n"

}

}

task closeAppServer {

doLast {

println "closeAppServer: Stop Applicaiton Server \n"

}

}

1. On the command line run the command gradle runWebApp. This will run, then the closeAppServer task will run after it is finished. Now run gradle tasks and notice the finalized Task and the finalizedBy Task both show up. This is because there is not a dependency defined between the Tasks as with dependsOn. The Finalizer task is not added to the task graph until the finalized task is scheduled to be executed.

The new tasks ordering dependency management methods mustRunAfter and shouldRunAfter are in the incubating phase. They define ordering between tasks that have no dependencies on each other, they can each run independently. But you still want to enforce an ordering if they both will execute in the build. The mustRunAfter rule ensures the order of execution if both tasks run. The shouldRunAfter rule will do a best effort to enforce the ordering. shouldRunAfter will fail if enforcement puts the build in a cycle, or if the build is using parallel execution and all other of the task’s dependencies have been satisfied already.

1. In your build.gradle file, add the following code. Here we are adding two Tasks, the first is compiling our code, and the second is cleaning up all files generated from our last build. These tasks have no dependency on each other, either can run independently. However, if both are run, we obviously want the cleanLastBuild task to run first. This is a good reason to use the mustRunAfter ordering rule.

task compileCode {

mustRunAfter "cleanLastBuild"

doLast{

println "running task $name \n"

}

}

task cleanLastBuild {

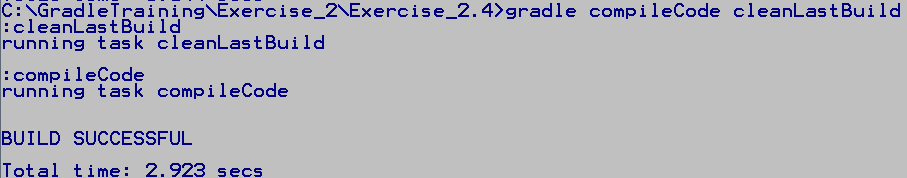
doLast {

println "running task $name \n"

}

}

1. Now run the command gradle compileCodecleanLastBuild. Notice that even though we specified compileCode first in the build script and first on the command line, cleanLastBuild runs first.



**Expected Result**: build.Gradle script demonstrating

* Dependencies between tasks with dependsOn
* Cleaning up resources with finalizedBy
* Ordering tasks with no dependencies on each other with mustRunAfter

## Dynamic Tasks

Gradle allows you to create tasks on the fly during the configuration phase. You can also manipulate existing tasks during this phase.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_2\Exercise\_2.5
2. In command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following task to the file and save it. Here we are using the Project instance to dynamically create a task during the configuration cycle

println "Configuration Phase Begins"

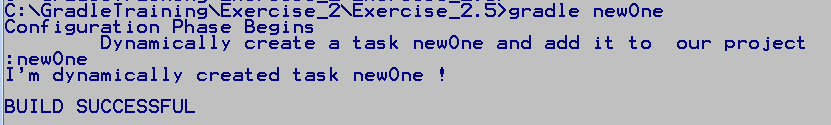
println '\tDynamically create a task newOne and add it to our project'

project.task('newOne') << {

println "I’m dynamically created task $name !"

}

1. Now run the command gradle newOne. Notice during the configuration phase we have dynamically created a new task, which we can invoke with Gradle.



1. We can also dynamically change this task changing the values of properties and adding dependencies. To add dependencies we’ll use the dependsOn method, but do it as a method call instead of in a closure. Add the following code to your build.gradle file.

Here we are adding a new task, foundationTask, and then in the configuration phase adding a dependency on this task and printing out we are doing it.

task foundationTask {

doLast {

println "Adding the Foundation \n"

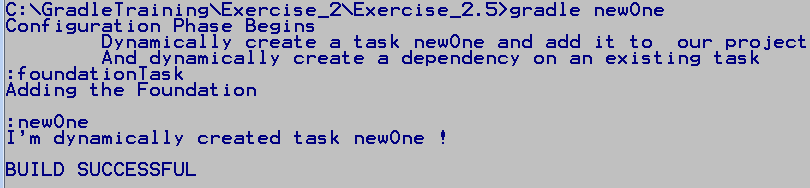
}

}

println '\tAnd dynamically create a dependency on an existing task'

newOne.dependsOn foundationTask

1. Now run the command gradle newOne. Notice during the configuration phase we dynamically created a new task, add a dependency to the task, which plays out when we invoke it from Gradle.



**Expected Result**: build.Gradle script demonstrating dynamically creating a new task with executing code, then also dynamically adding a dependency to the task.

## Adding New Properties

In your build.gradle file you can define local variable or extra properties on the Project or Task instance(s). Local variables are defined in the underlying Groovy environment, and very similar to Java local variable. They are only available in the scope in which they are defined.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_2\Exercise\_2.6
2. In the command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following task to the file and save it. Here we are add a local variable to the Project scope, projVar, and a local variable to the addProps Task Action scope, taskVar. We can see projVar from inside or outside the Task. However, we can only see the taskVar inside the doLast action.

println "Configuration Phase Begins"

String projVar = "local project scope var"

task addProps {

doLast {

String taskVar = "local task scope var"

println "addProps: access project scope variables: $projVar"

println "addProps: access task scope variables: $taskVar"

}

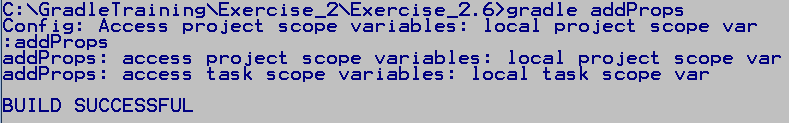
//println "Config: Cannot access task scope variables: $taskVar"

}

println "\t Access project scope variables: $projVar"

//println "\t Cannot access task scope variables: $taskVar"

1. In the command prompt enter gradle addProps. Notice the project scope variable is always available, and the Task Action scope variable is available in the Task Action where it is defined. Go ahead and uncomment the printlns to see Gradle generate an error when you try to access the Task Action scope variable outside its scope.



You can also define and initialize new properties on the Project and Tasks with the “ext” namespace. Once defined you can access the new property directly from the owning object via dot notation. Properties defined this way can be read or set, but they must be initialized when they are declared with “ext”.

1. Add the highlighted code to the build.gradle script. Here we are adding a new property to the project instance, newProjProp, and a new property to the Task addProps instance, newTaskProp. Notice when we print these properties out we access then via dot notation from the project or addProps task instances.

println "Configuration Phase Begins"

String projVar = "local project scope var"

project.ext.newProjProp = 45

println "\t Just added newProjProp with value $project.newProjProp"

task addProps {

doLast {

String taskVar = "local task scope var"

println "addProps: access project scope variables: $projVar"

println "addProps: access task scope variables: $taskVar"

println "addProps: access new task property $newTaskProp"

}

//println "Config: Cannot access task scope variables: $taskVar"

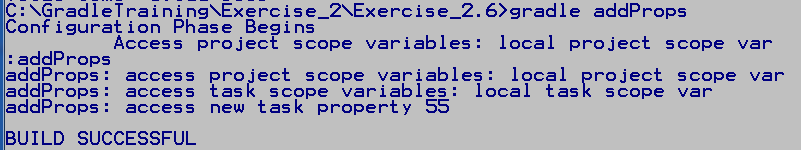
}

println "\t Access project scope variables: $projVar"

//println "\t Cannot access task scope variables: $taskVar"

addProps.ext.newTaskProp = 55

1. In the command prompt enter gradle addProps. Notice the properties were added to the project and addProps task instances since we can print out the values.



**Expected Result**: build.Gradle script demonstrating how to add and use local variable, and properties on the Project and Task instances.

End of Lab

**If you get done Early:** Read the introduction in Gradle JavaDoc pages for the Interfaces org.gradle.api.Project and org.gradle.api.Task.

# Gradle Projects (60 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Browser open to the Gradle JavaDoc page.
* Work directory created as C:\GradleTraining\work\Exercise\_3. Note, the solutions are in directory C:\GradleTraining\Exercise\_3.
* Command prompt windows navigated to this new directory

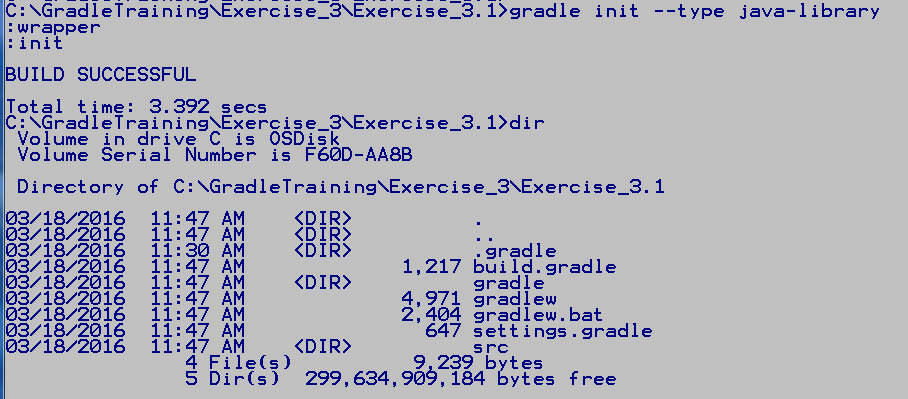
Description

In this lab we will use Gradle plugins to build and execute several different types of projects include Java and web applications

## Creating a Gradle Java Project with Init

The Build Init Plugin is an incubating feature, it supplies the wrapper and the init tasks. The Build Init Plugin is automatically applied, so it doesn’t need to be applied in a build.gradle file before it is used. This automatic apply also the plugin’s Init task can be run in an empty directory to create a new Gradle directory structure, complete with a default build.gradle script.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_3\Exercise\_3.1
2. In command prompt navigate to this new directory.
3. Enter the command gradle init --type java-library. This will create a Java library project. The type flag is used to specify the type of project you want to create, Java, Maven, Scala, Groovy, or a basic type that simply creates a sample build.gradle file.



1. In the file explorer open the directories in the created project. Notice the src directory has a folder for both the main application and the tests.
2. Open the generated build.gradle and the settings.gradle files to see what was configured for you. We’ll be looking at the gradlew, Wrapper files in a later lab.

**Expected Result**: Java gradle project generated from scratch by the Build Init plugin.

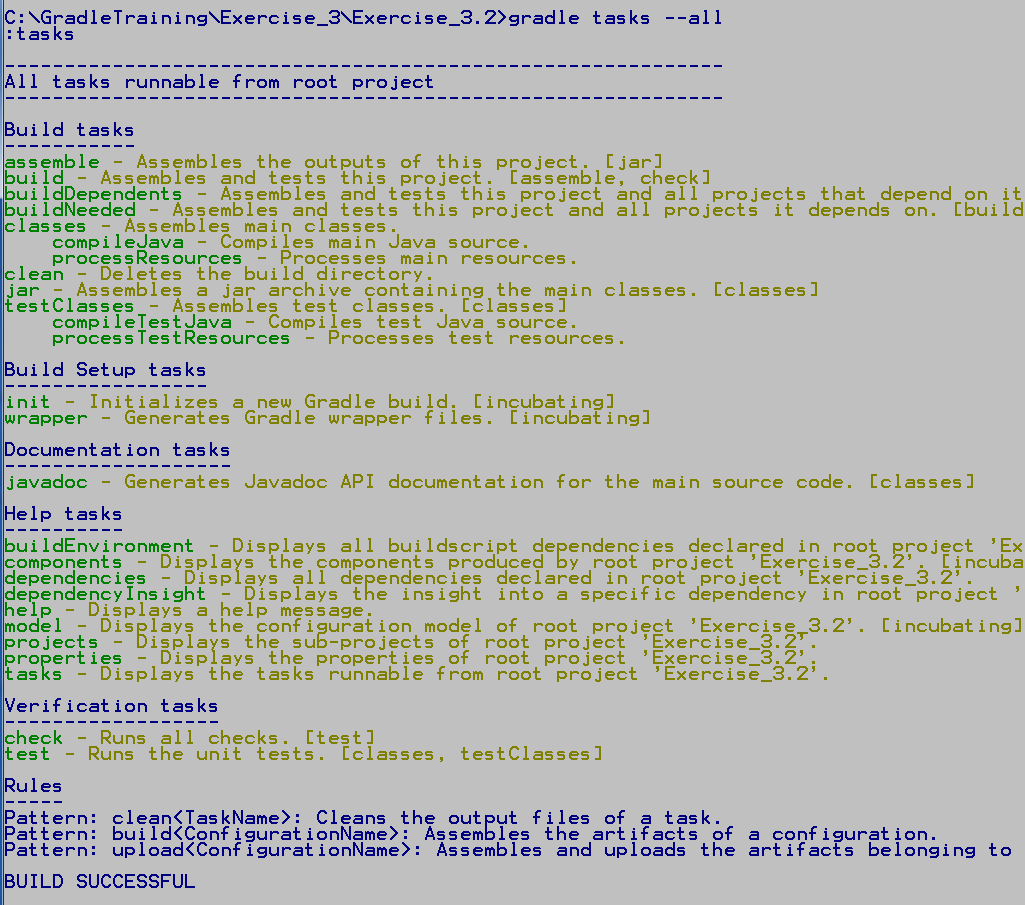
## Using Java Plugin

The Gradle Java Plugin adds a number of tasks that make setting up a Java build very easy. These task will clean, compile, jar, and test upload your project, as well other helpful tasks to help you manage your code base.

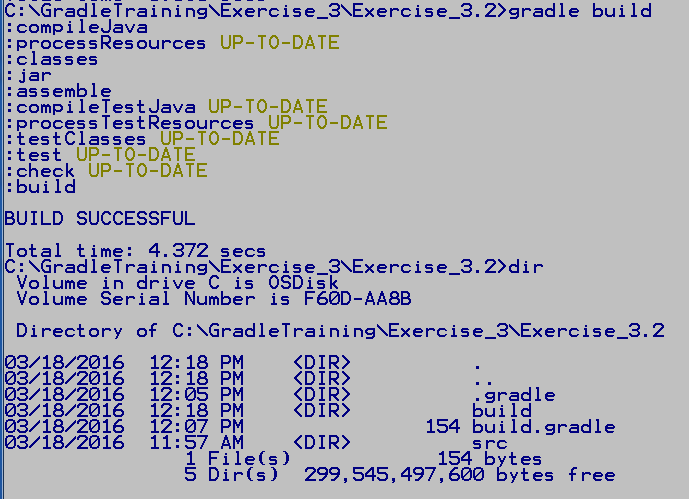
1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_3\Exercise\_3.3
2. Copy the directory C:\GradleTraining\Exercise\_3\Exercise\_3.2\src to your new work directory.
3. In command prompt navigate to this new work directory.
4. Create a build.gradle file in this directory and open it in Notepad++.
5. Add the following task to the file and save it. Here we are applying the gradle Java Plugin to have access to the many tasks it gives us.

apply plugin: 'java'

1. On the command line enter the command gradle tasks --all. Notice the new task groups added by the Java plugin, Build Tasks, Documentation tasks, and Verification tasks. Notice the dependencies between the build tasks, for instances, Gradle will need to execute compileJava and processResources before it can execute the classes task. A few new Rule have also been defined for us.



1. To build the project enter the command gradle build. Notice the all the tasks that Gradle executes which the build task depends upon. Also notice that after the build we have a new build directory that contains the results of our build. The jar file we just created is in the build\libs directory.



1. On the command line run the jar with the command java –jar build\libs\xercise\_3.2.jar. You’ll get an error about not having a main manifest attribute in the jar. Without this attribute Java doesn’t know which class holds our main.
2. Add the following code to the build.gradle file. Here we are adding attributes to the jar manifest inside the Gradle build script.

jar {

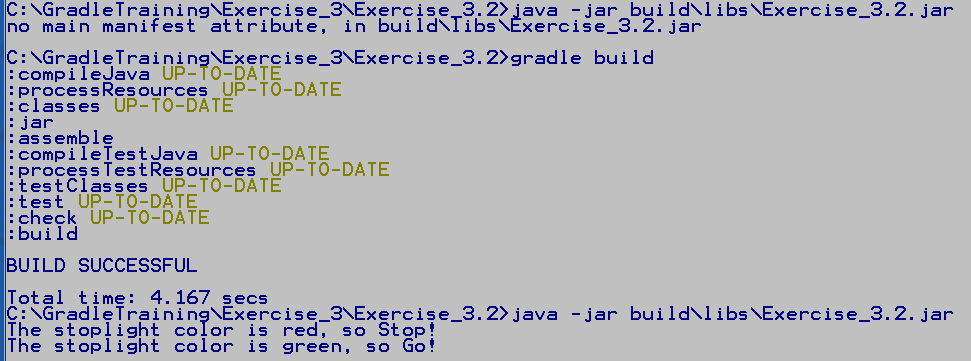
manifest {

attributes 'Main-Class': 'com.training.gradle.SwitchStoplight'

}

}

1. Not enter the command gradle build again. Notice most tasks are up-to-date, only the jar, assemble, and build tasks actually did any work. Now when we run our jar file it will execute the main in our SwitchStoplight class.



**Expected Result**: Simple Java application using the Gradle Java Plugin to build the jar and update the jar manifest.

## Using the War and Gretty Plugin

The Java Plugin makes simple Java application builds very easy since it does so much for you. The same is true of what the Gradle War and Gretty plugins do for web application development. The Gretty plugin is a full featured servlet container that requires JDK 6 or higher, Gradle 1.10 or higher, and system resource to easily run Gradle and the Jetty or Tomcat web containers.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_3\Exercise\_3.3
2. Copy the directory C:\GradleTraining\Exercise\_3\Exercise\_3.3\stoplight to your new work directory.
3. In command prompt navigate to the new work directory, stoplight.
4. There is a solution.gradle file in this directory, but you will be creating your build.gradle file from scratch. Create a build.gradle file in this directory and open it in Notepad++.
5. Add the following code to the file and save it. Here we are applying the war and org.akhikhl.gretty plugins. The War plugin extends the Java plugin, so we don’t have to explicitly include it. Since we are specifying a community plugin with a version, we are using the new plugins DSL, or syntax, to declare plugins in our Gradle script. This new plugin DSL is still incubating. It allows us to specify the fully qualified plugin id from the community plugin portal, which is all Gradle need to know to locate, download, and make it available to the build. The Gradle community plugin portal is located at <https://plugins.gradle.org/>.

We’ve configured the Gretty plugin to use the Jetty web container and have it listen on port 9080. And we’ve setup Gradle to go to the Maven repository to download our needed servlet dependencies. We will cover external dependencies and repository later in this course.

plugins {

id "org.akhikhl.gretty" version "1.2.4"

id 'war'

}

gretty {

servletContainer = 'jetty9'

port = 9080

}

repositories {

mavenCentral()

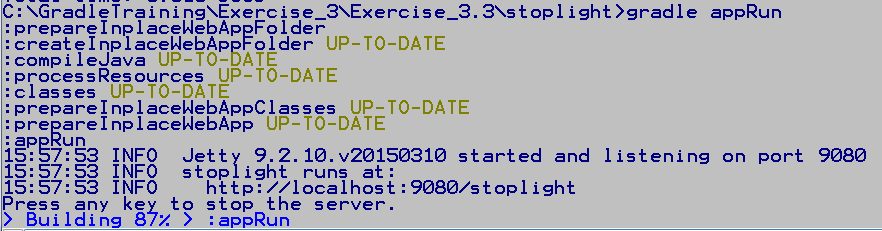
}

dependencies {

providedCompile 'javax.servlet:javax.servlet-api:3.1.0'

}

1. On the command line enter gradle build. When the build finishes enter gradle appRun to run the Jetty9 web container. The output will tell you the URL to use to access the web application.



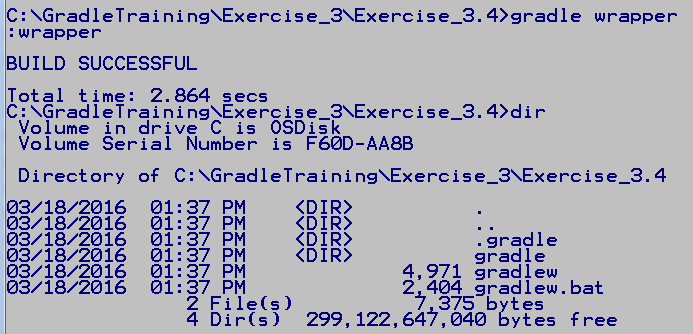
1. Open your browser and go to the URL to run the webApp. Add a color, or any text to the text box and hit the Submit button. Hit the back button to try a different input. To stop the Jetty9 server, hit any key at the command line.

**Expected Result**: Gradle build web application running in a web container provided by the Gradle Gretty plugin.

## The Gradle Wrapper

The Gradle Wrapper allows you to move your project to a build server and build it with the Gradle version that was used in development, even if Gradle is not installed on the build server. This makes historic builds much more reliable and consistent. Even if the build server has a different version of Gradle installed, the Wrapper will download the correct version and use it to build the project.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_3\Exercise\_3.4
2. In command prompt navigate to this new directory.
3. Enter the command gradle wrapper in the empty directory. Notice Gradle runs and creates the Gradle Wrapper files and directories. The Wrapper scripts you execute are gradlew for the Unix shell and gadlew.bat for Windows. In the .\gradle\wrapper directory you’ll find the gradle-wrapper.jar file and the gradle-wrapper.properties file that holds the specifics about your build. Open the properties file to see the build properties the Wrapper stores by default.



1. Create a build.gradle file in this directory and open it in Notepad++.
2. Add the following code to the file and save it. Here we are creating a task of type Wrapper, this creates an org.gradle.api.tasks.wrapper.Wrapper instance and associates it with our task. We then use the task closure to set some of the Wrapper properties. We want to use Gradle version 2.10 for our build, set the saved Gradle distribution archive directory to gradleArchive, and set the directory where the Gradle distribution is unzipped to gradleDist. Both these folders are relative to the Gradle distribution base directory which defaults to the .gradle folder in your home directory.

task wrapper(type: Wrapper) {

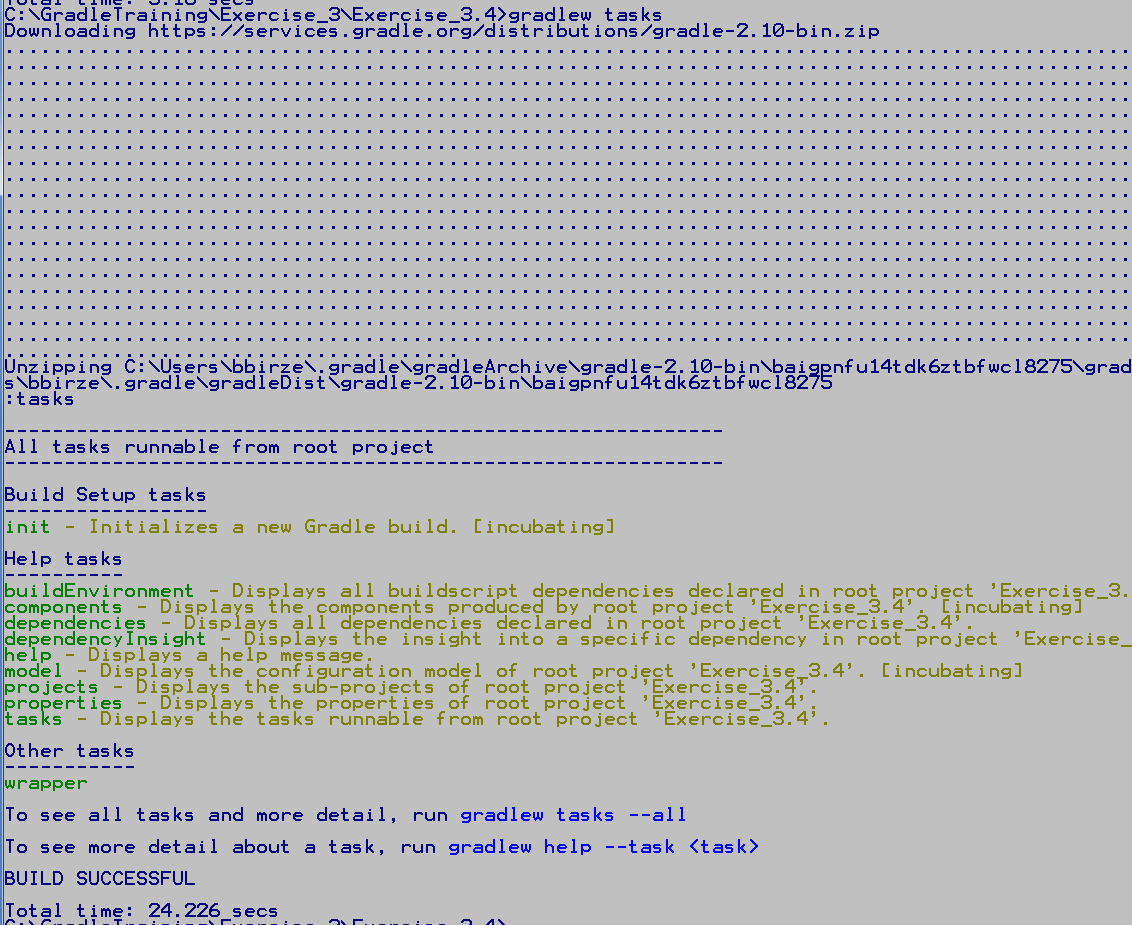
gradleVersion = '2.10'

archivePath = "gradleArchive"

distributionPath = "gradleDist"

}

1. From the command line enter gradle wrapper to create a new Wrapper distribution. Then enter gradlew tasks to use the gradlew.bat file to see the tasks. Notice you can use gradlew just as you do the gradle command. Notice how the wrapper task use to be under the Build Setup tasks group, but is now under the Other tasks group. This is because we created a task of type Wrapper, if not specified our tasks go in the catch all Other tasks group. However, it is still creating an org.gradle.api.tasks.wrapper.Wrapper class under the covers to do the work.



1. If you go to the Gradle distribution base directory in c:\Users\Elevated\.gradle, you’ll see two new directories, gradleArchive and gradleDist. Each contains a gradle-2.10-bin directory. If you drill down into each you’ll find gradleArchive holds the gradle 2.10 zip file, and gradleDist holds the unzipped distribution.

**Expected Result**: project that demonstrates how to create a Gradle Wrapper and how to configure the properties of org.gradle.api.tasks.wrapper.Wrapper to control how the Wrapper does its work.

End of Lab

**If you get done Early:** Read more about the plugins we used in the lab including

* Build Init Plugin: can be used to setup different types of projects, <https://docs.gradle.org/current/userguide/build_init_plugin.html>
* Java Plugin: <https://docs.gradle.org/current/userguide/java_plugin.html>
* War Plugin: <https://docs.gradle.org/current/userguide/war_plugin.html>
* Gretty Plugin: supplies both a Jetty and a Tomcat imbedded web server, <http://akhikhl.github.io/gretty-doc/>

# Using Gradle In Eclipse (20 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Eclipse version 3.6.2 or later

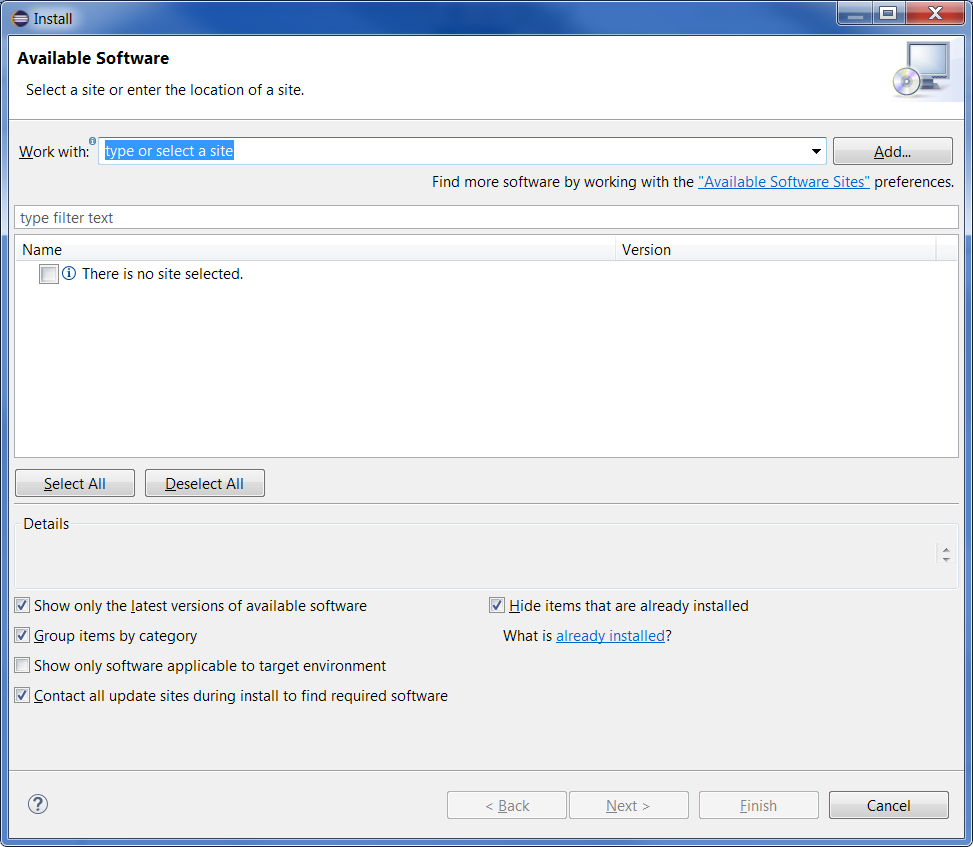
Description

In this lab we will be configuring our Eclipse IDE to recognize Gradle projects, adding the ability to import, create, and control the build of these projects.

## Installing the Eclipse Gradle Plugin

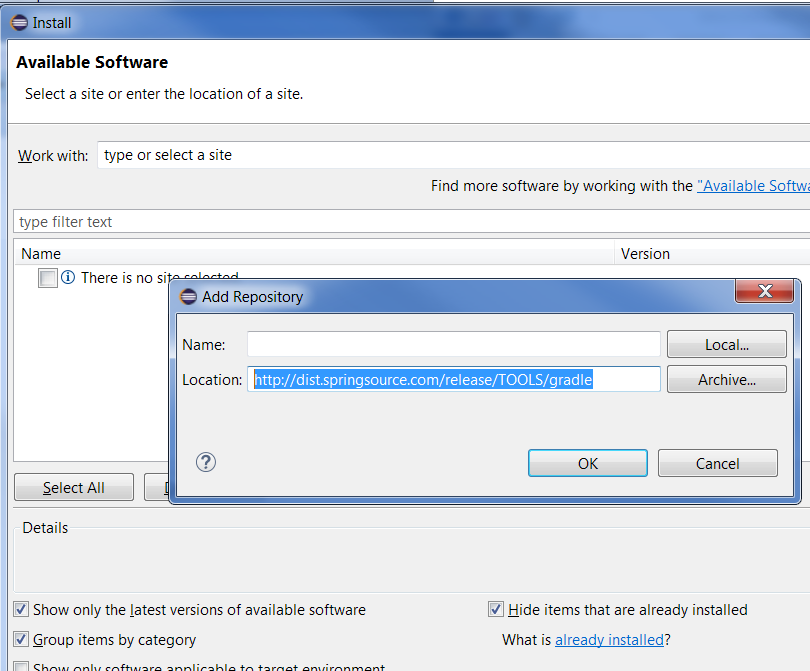
Before we can import a Gradle project into Eclipse and build it, we need to install the Gradle plugin.

1. From the menu bar, select Help->Install New Software… This will bring up the Available Software window. Click on the Add… button.



1. In the Add Repository window Location textbox, enter the following and click OK

<http://dist.springsource.com/release/TOOLS/gradle>



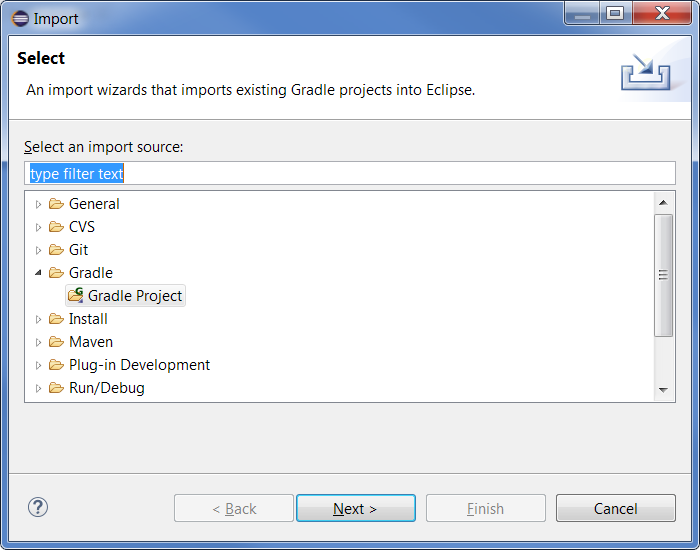
1. Select the Extensions / Gradle Integration option and click Next. Click Next on the next screen, then accept the terms of the license agreement and click Finish.
2. When Eclipse prompts you to restart for the changes to take effect, select Yes.

**Expected Result**: Eclipse with Gradle plugin installed and read to use.

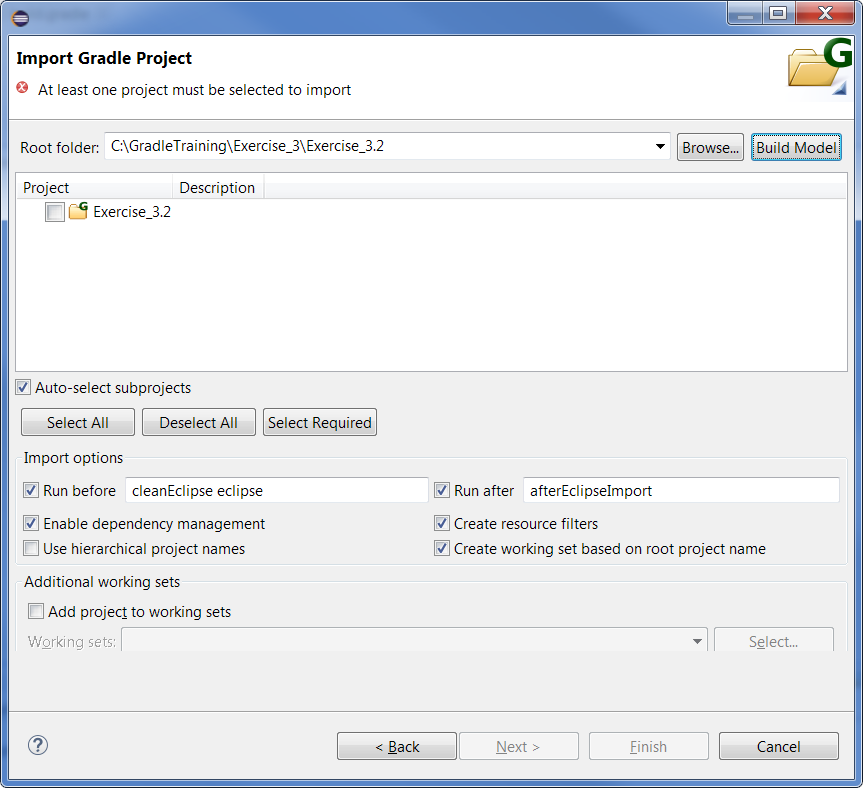
## Import a Gradle Project into Eclipse

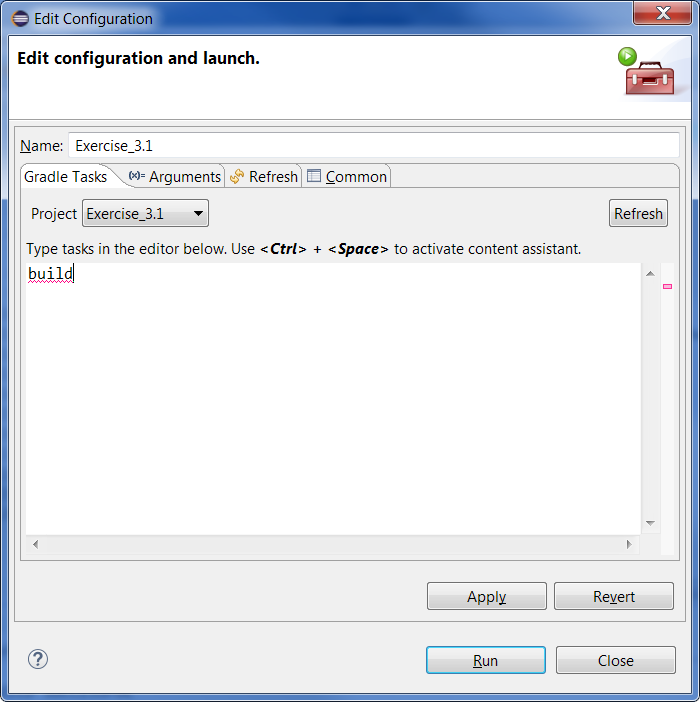
We are going to be importing the project we created in Lab 3.2 into Eclipse.

1. Open Eclipse and in the Project Explorer window right click and select Import->Import… This will bring up the Import Wizard.
2. Select the Gradle folder and then the Gradle Project option. Hit the Next button.



1. In the Import Gradle Project window, Browse to the 3.2 project directory. The path will appear in the Root folder: text box. Click on the Finish button, then hit the Build Model button. If the project is a part of a hierarchy of sub-projects, Build Model will create a project/sub-project structure to import the sub-projects and correctly represent the relationship between projects including resolving their dependencies.



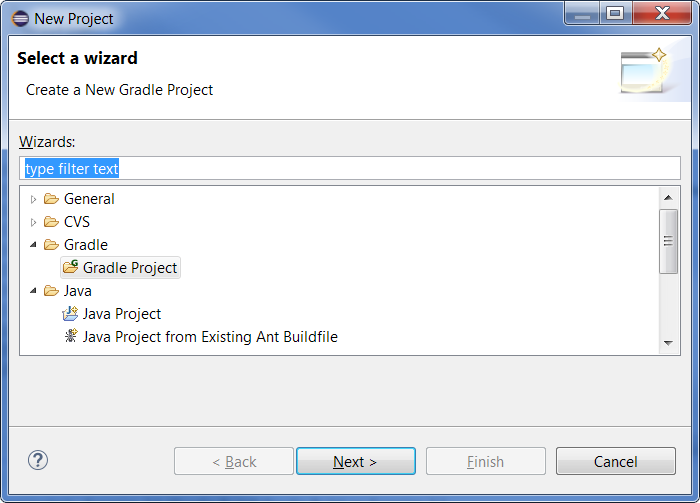
1. Once the model is built, the project will show up in the Project window. Select its checkbox and click on the Finish button.
2. Right click on the imported project in the Package Explore and select Run As->Gradle Build. When the Edit Configuration window comes up, type build in the tasks window and hit the Run button. Watch the build happen in the Console window.
3. 

**Expected Result**: Gradle project imported and built in Eclipse environment.

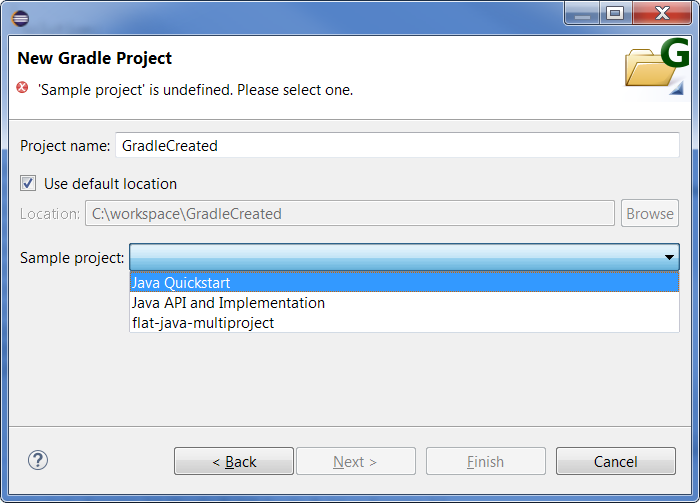
## Create a Gradle Project in Eclipse

The Gradle plugin will also create a Gradle project structure for you from scratch.

1. In the Package Explorer right click in an open space and select New->Project. The New Project wizard will come up.
2. Select the Gradle->Gradle Project option and select Next.



1. Enter project name “GradleCreated”. From the Sample project dropdown list, select Java Quickstart. Then select Finish. A new project will be created for you.



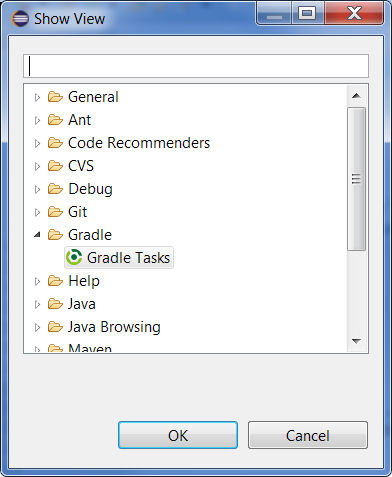
1. Open the new project to see the file structure. Open the build.gradle file to see the build script generated for you. We’ll learn about those dependencies and repositories in a later module.
2. Right click on the imported project in the Package Explore and select Run As->Gradle Build. When the Edit Configuration window comes up, type build in the tasks window and hit the Run button. Review the build output in the Console window. You should see some warning as bootstrap is expected by this generated project.

**Expected Result**: Gradle project created and built in Eclipse environment.

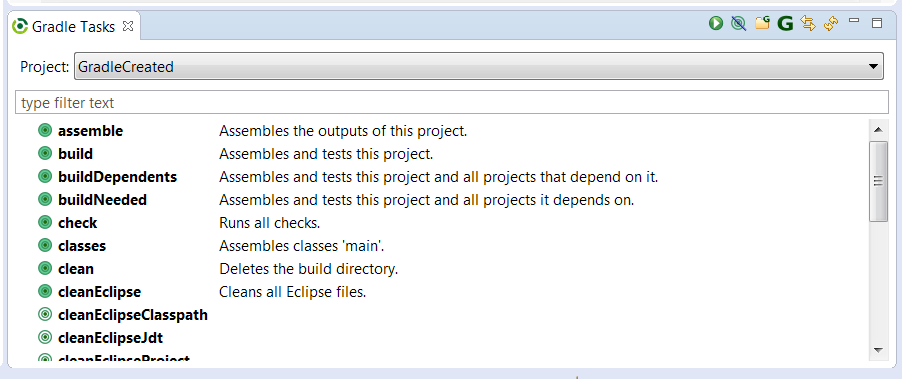
## Executing Gradle Tasks in Eclipse

Though you can specify your task in the Edit Configuration window, that’s really isn’t as easy and convenient as the Gradle Task window. The Gradle Eclipse plugin gives you the ability to use the Gradle task window.

1. To bring up the Gradle task window, from the Eclipse menu bar select Window->ShowView->Other. This will bring up the Show View window, where you can select Gradle->GradleTask. Then hit the OK button.



1. You can position the Gradle Tasks window in the side or bottom grouping by clicking and holding on the Gradle Task tab, and moving the window where you like. Notice the Gradle Task window has a pull down Project where you can select the working Gradle project. Use it to select your GradleCreated project



1. The list below includes all the Gradle task you can run in your project. You can use the filter text box just below the Project selector to start typing a task name to find it quickly. Type “bui” in the filter text box to see the list reduced to build, buildDependents, and buildNeeded tasks. Double click on build to run this task, review the output in the console window.

**Expected Result**: Knowledge of how to build selected Gradle tasks in Eclipse environment.

End of Lab

**If you get done Early:**

* Read about the BuildshipEclipse Gradle Plugin which was released 6/25/2015 in a partnership between Gradle and Eclipse. <http://gradle.org/press-release/eclipse-gradle/>.

# Creating and Using Gradle Plugins (20 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Browser open to the Gradle JavaDoc page.
* Work directory created as C:\GradleTraining\work\Exercise\_5. Note, the solutions are in directory C:\GradleTraining\Exercise\_5.
* Command prompt windows navigated to this new directory

Description

Gradle provides many core plugins that extend its functionality. The Gradle community is constantly creating new plugins to further expand this functionality. In this project we will create both a Gradle Script plugin and a Gradle binary plugin. Then we will load them into our Gradle build script and execute their tasks.

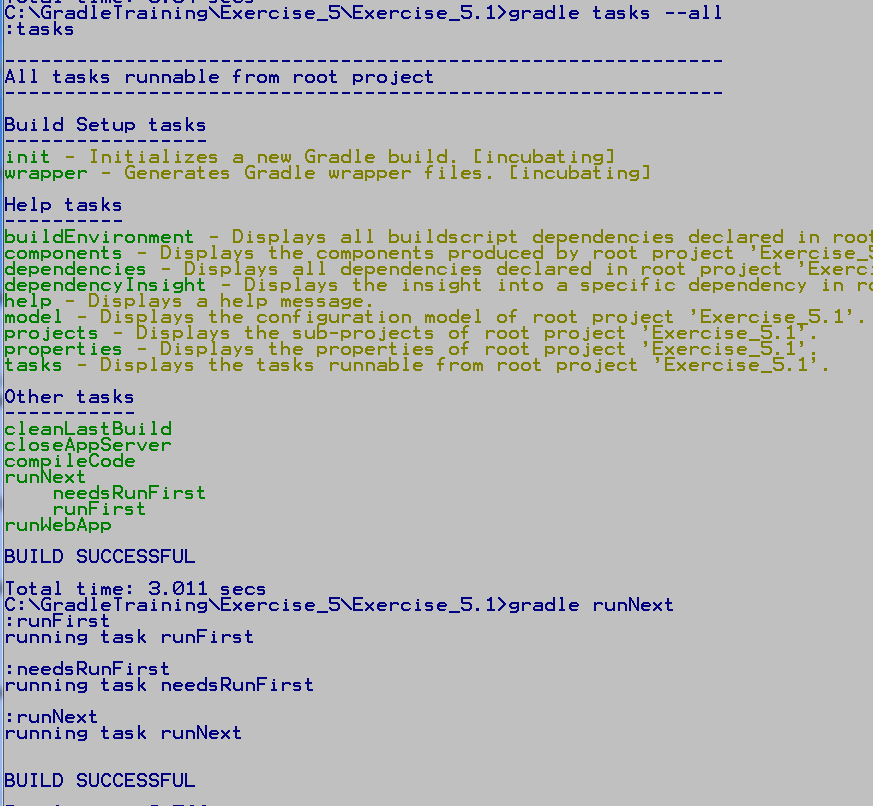
## Creating a Gradle Script Plugin

Gradle script plugins are just portions of a build.gradle script we’ve moved into another gradle file with a different name. We can then reuse these code snippets in multiple builds, hiding the complexity of the reused code from the mainline build script. Script plugins can be in the same build directory structure, but they can be accessed remotely.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_5\Exercise\_5.1
2. In command prompt navigate to this new directory.
3. Copy your build.gradle file from the 2.4 lab into your new work directory. This is the lab where we worked with action dependencies between different tasks. Open the file you copied into your work directory in Notepad++.
4. Create another file in this directory dependencyTasks.gradle. Open it in Notepadd++ as well.
5. We are going to copy dependency tasks that do not show up in the Gradle task list unless we use the --all flag. This will clean up our build script, making it easier to read. First move tasks runNext and runFirst into the dependencyTasks.gradle file. They will no longer appear in your main build script. Then add the following line to the top of the build.gradle file. Save both files

apply from: 'dependencyTasks.gradle'

1. From the command line run gradle tasks. Notice how the tasks we moved are still showing up in the task list. And if we run gradle runNext, the tasks it depends on are run first.



1. Looking at our build.gradle file, notice that even though the dependency tasks were in a different file, needsRunFirst did not have to quote runFirst in its dependsOn call. This is because the tasks in dependencyTasks.gradle were applied, or included, before needsRunFirst was created. Move the apply from line below the needsRunFirst task definition, and run gradle tasks again. You’ll get an error because the task runFirst was used before it was defined, and it was not quoted.

task needsRunFirst {

dependsOn runFirst

doLast {

println "running task $name \n"

}

}

apply from: 'dependencyTasks.gradle'

1. Now quote runFirst in the dependsOn call. Try to run gradle tasks again. Now it runs fine. Applying a script plugin is the same as if the code in that script had been inserted in the build.gradle file wherever the plugin was applied.

**Expected Result**: Gradle build that demonstrates creating and using a Gradle script plugin.

## Creating a Gradle Binary Plugin

Gradle binary plugins are compiled Java or Groovy code that defines a class implementing the Plugin interface. Binary plugins can be built in the same project or accessed already built from a jar. They can also be specified by the fully qualified name of the class.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_5\Exercise\_5.2
2. In command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following code to the file and save it. Here we are implementing a binary plugin in the same build script by creating a class which implements Plugin. Gradle will compile and include this plugin in the classpath, so we can simply apply it to use its task. This plugin gives us the full ability of the Groovy programming language using the underlying API classes, however, it cannot be re-used outside the build where it is defined.

apply plugin: SimplePlugin

class SimplePlugin implements Plugin<Project> {

void apply(Project proj) {

String myName = "SimplePlugin:"

proj.task('simple') << {

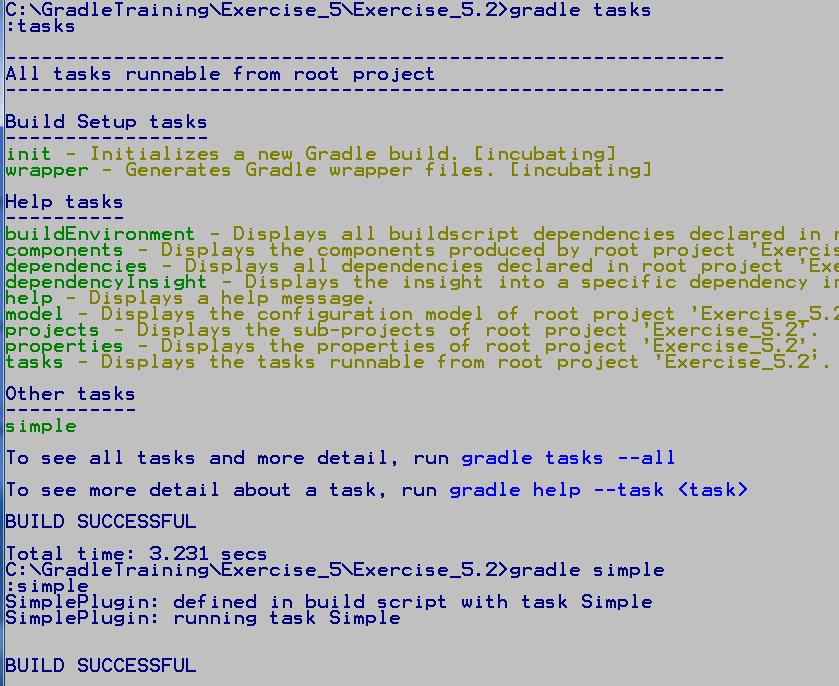
println "$myName defined in build script with task Simple"

println "$myName running task Simple \n"

}

}

}

1. Go to the command line and run gradle tasks, notice the task simple for our new plugin shows up. Run gradle simple and see the output.

**Expected Result**: Gradle build that demonstrates creating and using a simple Gradle in-script binary plugin that is completed by Gradle and applied to the script so its tasks can be executed.

End of Lab

**If you get done Early:**

* Read Gradle user’s manual chapter on writing custom at <https://docs.gradle.org/current/userguide/custom_plugins.html>.
* Read Gradle user’s manual chapter on the Java Gradle Plugin Development Plugin at <https://docs.gradle.org/current/userguide/javaGradle_plugin.html>
* Look at the Sample project that builds a standalone Groovy custom plugin in your gradle installation under samples/customPlugin.

# Incremental Builds and Multi Projects (50 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Browser open to the Gradle JavaDoc page
* Work directory created as C:\GradleTraining\work\Exercise\_6. Note, the solutions are in directory C:\GradleTraining\Exercise\_6.
* Command prompt windows navigated to this new directory

Description

In this project we will create a custom task class and execute its actions in a Gradle Build Script. We will also modify that task to use the Gradle IncrementalTaskInputs object to preform incremental builds. Then we’ll explore the structure and dependencies between multiple projects in a single build.

## Creating and Executing a Custom Task Class

We’ve been creating simple tasks in our Gradle build scripts. But Gradle also supports enhanced tasks with configuration properties, and the ability to create custom task classes.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_6\Exercise\_6.1
2. In command prompt navigate to this new directory.
3. Create a build.gradle file in this directory and open it in Notepad++.
4. Add the following code to the file and save it. Here we are defining a custom task class which extends org.gradle.api.DefaultTask. Our custom task class has a single property, effort, which is used in our method doWork. The @TaskAction annotation tells Gradle our doWork method is an action. We can define many actions in our custom task class.

class customTask extends DefaultTask {

String effort = "unknown"

@TaskAction

def doWork() {

println "Task $name is doing $effort!"

}

}

Now we need a way to invoke our custom task. We do this by creating an enhanced task which gets its behavior from our custom task class. Enhanced tasks can also set the properties of the custom task class they declare

1. Add the following code to your build.gradle file and save the file. We have defined 3 tasks, each of type customTask. Each task sets the effort property to change the behavior of the doWork method.

task workHard(type: customTask) {

effort = "hard work"

}

task workEasy(type: customTask) {

effort = "easy work"

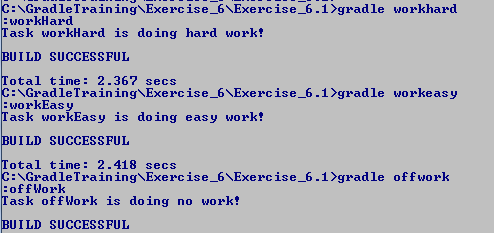
}

task offWork(type: customTask) {

effort = "no work"

}

1. From the command line enter gradle workHard, then gradle workEasy, then gradle offWork. Notice how each task has set a different value for the effort property, which changed the output. In this way we can write a custom task class with configuration properties and create many tasks of its type, each setting a different configuration by setting its properties.



**Expected Result**:

## Creating a Custom Incremental Build Task

We will be simulating keeping the files on a warm start server synced with the active server files.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_6\Exercise\_6.2
2. Copy the three directories from C:\GradleTraining\Exercise\_6\Exercise\_6.2 to your new work directory. There is a directory ActiveServer with several server files, an application server and data base configuration, a batch file output, and a dat file. The WarmStartServer directory is empty. The 0InitalFiles directory holds a copy of the original ActiveServer files to easily reset the files after running a few test.
3. In command prompt navigate to this new directory.
4. Create a build.gradle file in this directory and open it in Notepad++.
5. Add the following custom task class to the file and save it. Here we are creating an Incremental Tasks because it has an Incremental Task Action, doSync. Incremental Task Actions methods take a single argument of type IncrementalTaskInputs. The annotations tell Gradle the location of the two files groups that need to be compared. And the doSync method assigns an outOfDate and a remove Actions to the given IncrementalTaskInputs. Gradle will use these Actions on the files that have changed or been deleted.

class SyncServers extends DefaultTask {

@InputDirectory

def File inputDir

@OutputDirectory

def File outputDir

@TaskAction

void doSync(IncrementalTaskInputs changes) {

if (changes.incremental == true) {

println "Some output files need updating"

}

else {

println "All output files need updating"

cleanOutDir() // ensure Gradle updates everything

}

// assign Actions

// Steps to perform on each file that was updated

changes.outOfDate { f ->

println "${f.file.name} was updated in $inputDir"

def ourDirFile = new File(outputDir, f.file.name)

ourDirFile.text = f.file.text

}

// Steps to perform on each file that was deleted

changes.removed { f ->

println "${f.file.name} was deleted from $inputDir"

def ourDirFile = new File(outputDir, f.file.name)

ourDirFile.delete()

}

}

def cleanOutDir() {

project.delete(outputDir.listFiles())

}

}

1. Now we need to define a task to configure our new task class. At the beginning of the build.gradle file add the following task. This declares a task syncWarmStart of type SyncServer. We then assign values to the SyncServer properties inputDir and outputDir.

task syncWarmStart(type: SyncServers) {

inputDir = file('ActiveServer')

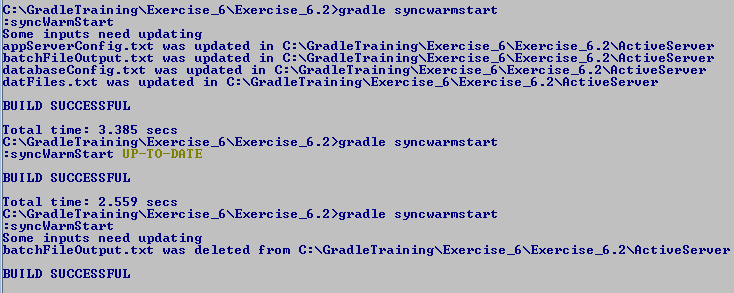
outputDir = file('WarmStartServer')

}

1. At the command line enter gradle syncWarmStart. Notice when the doSync action runs it queries the changes IncrementalTaskInputs argument it gets from Gradle. If its incremental property is set to true, Gradle knows exactly which files need updating. If it is false, Gradle doesn’t know which files changed and we’ll need to update all files. We assign our outOfDate action to the changes argument and it is run for each file that needs updating, outputting the file name and input directory. If you look at the WarmStartServer directory you’ll notice it now matches the files in the ActiveServer directory. We are synced up!

If we run gradle syncWarmStart again, Gradle says we are up-to-date since no files have changed. Then if we delete the file ActiveServer\batchFileOutput.txt, and run gradle syncWarmStart again, the removed action is called for each file removed from the input directory.

< Initial run, sync all files



< deleted one input file

< all files synced, no changes

1. Try updating, adding, or deleting files from the ActiveServer directory and running the syncWarmStart task to see how Gradle reacts. What happens if you change a file in the WarmStartServer output directory? You can always restore the original state of the directories by deleting the files in both directories and copying the files from 0InitalFiles into ActiveServer.

**Expected Result**:

## Working with Multiple Projects

When working with multiple projects in a single build, we are required to supply a settings.gradle file which tells Gradle which project is the root project, and which sub-project to include in the project hierarchy.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_6\Exercise\_6.3
2. In command prompt navigate to the new work directory, rootProj.
3. Copy the directory C:\GradleTraining\Exercise\_6\Exercise\_6.3\rootProj to your new work directory. The rootProj directory includes 2 sub-projects subProj1 and subProj2. To tell Gradle about our subprojects we need to create a settings.gradle file in the rootProj directory. Once created, open the file in Notepad++ and add the following line to declare our subprojects. Save the file.

include 'subProj1', 'subProj2'

1. Each project contains a solution.gradle file in its home directory, but you will be creating your build.gradle file from scratch. Create a build.gradle file in the rootProj directory and open it in Notepad++.
2. Add the following lines to the file and save it. Here we are outputting the start of the configuration phase for this project. Then we use allprojects to output the end of the configuration phase for each project in the build.

println "Configuration Phase Begins for project: $project.name"

allprojects {

afterEvaluate { proj ->

println "Configuration Phase Ends for project: $proj.name"

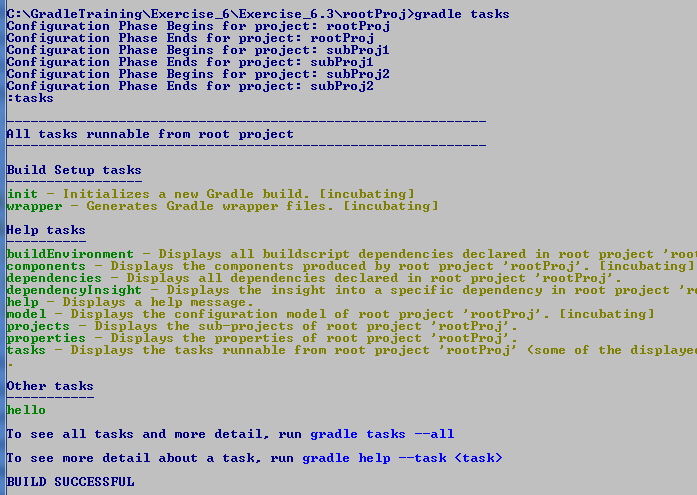
}

}

1. Create a build.gradle in the subProj1 directory. Add the following line to it to output the start of the configuration phase for this project, then save it.

println "Configuration Phase Begins for project: $project.name"

1. Copy the subProj1\build.gradle file to subProj2\build.gradle. Now all three projects have a Gradle build script.
2. At the command line in the rootProj directory run gradle tasks. Notice how each project is visited during the configuration phase, before tasks is invoked during the execution phase. Also, notice that we defined common behavior for all the projects in the build in the rootProj build script, where we used allprojects to output when each project’s configuration ended. Gradle lets you access any Project associated with the build from any build script in the build.



1. Add the following code the rootProj\build.gradle. Here we are adding a task, rootTask, with a dependency on subtask in subProj2. In our Action we print out our project and task name.

task rootTask {

dependsOn ':subProj2:subTask'

doLast {

println "In project: $project.name running task $name"

}

}

1. Add the following code the subProj2\build.gradle. Here we are adding a task, subTask, with a dependency on subtask in subProj1. In our Action we print out our project and task name.

task subTask {

dependsOn ':subProj1:subTask'

doLast {

println "In project: $project.name running task $name"

}

}

1. Add the following code the subProj1\build.gradle. In our Action we print out our project and task name. This task has no dependencies.

task subTask {

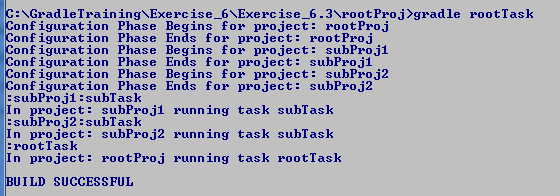
doLast {

println "In project: $project.name running task $name"

}

}

1. At the command line in the rootProj directory run gradle rootTask. Notice how the dependencies between project tasks is honored. Also, when Gradle prints out the task it is about to run, if the task is in a sub-project, the project name is prefixed onto the task name.



1. Now add the following task to each of the 3 build scripts and save the files. Now each project has a new task, independentTask, which does not depend on any other tasks, and is not a dependency of any other tasks.

task independentTask {

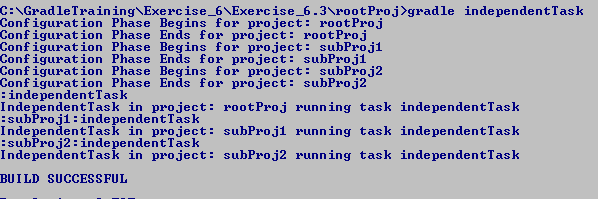
doLast {

println "IndependentTask in project: $project.name running task $name"

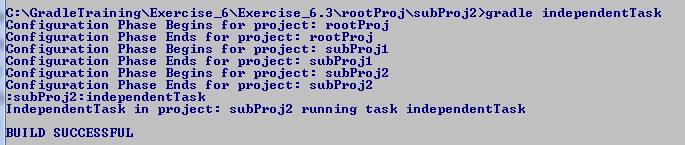
}

}

1. At the command line in the rootProj directory run gradle independentTask. Notice Gradle runs the IndependentTask in each of the three projects.



1. Now in the command prompt, go to the subProj2 directory and enter gradle independentTask again. This time only the subProj2 independentTask is run, even though from the configuration output we know all three projects are included in the build. This is because when Gradle executes tasks, it looks for the task first in the *current* directory, then looks for the task in projects further down the project hierarchy.



**Expected Result**: Multiple projects included in the same Gradle multi-project build demonstrating accessing other Projects in the build, sharing behavior between projects, declaring dependencies between projects, and demonstrating how Gradle orders execution of projects.

End of Lab

**If you get done Early:**

* Read the Gradle User’s Manual chapter on Multi-Project Builds at <https://docs.gradle.org/current/userguide/multi_project_builds.html>
* Read the Gradle User’s Manual chapter on Writing Custom Task Classes at <https://docs.gradle.org/current/userguide/custom_tasks.html>

# External Dependency Management (25 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Internet access
* Browser open to the Gradle JavaDoc page
* Work directory created as C:\GradleTraining\work\Exercise\_7. Note, the solutions are in directory C:\GradleTraining\Exercise\_7.
* Command prompt windows navigated to this new directory

Description

In this lab we will work with Maven external and local repositories to include external tool dependencies in our build.

## Working with External Dependencies and Repositories

Gradle can help us manage external dependencies required by our project. But to do this Gradle needs to know two things

* The identifying information about each dependency, including version
* And where to find the dependency

With this information Gradle will be able to download the specified external dependencies to your environment and include them in your build.

1. Create a directory for this exercise named C:\GradleTraining\work\Exercise\_7\Exercise\_7.1
2. Copy the directory C:\GradleTraining\Exercise\_7\Exercise\_7.1\dependencies to your new work directory. The sample application has one application class and one test class, both use log4j. The test class also uses Junit. So we’ll need to resolve these external dependencies in our Gradle build script.
3. In command prompt navigate to the new work directory, dependencies.
4. There is a solution.gradle file in this directory, but you will be creating your build.gradle file from scratch. Create a build.gradle file in this directory and open it in Notepad++.
5. Add the following code to the file and save it. Here we are applying the Java plugin, then telling Gradle where to find our dependencies. Gradle will search the given repositories in the order they are listed.

If we used Maven on this machine in the past, we will have a Maven local repository which may hold a cache of the dependencies we need. Getting them from the local machine will be faster than downloading them from Maven’s central repository.

**apply plugin: 'java'**

**repositories {**

**mavenLocal()**

**mavenCentral()**

**}**

The Java plugin gives us a number of dependency configurations we can use to specify what part of our project needs the dependency, application or test code, and when the dependency is needed, compile time or runtime.

1. Add the following code to the build.gradle file. We have added 4 dependencies to our project using 3 different dependency configurations. Our application depends on log4j during compile time, so we use the compile dependency configuration to specify this. And our unit test requires both log4j and JUnit to compile. During runtime, out unit test need the file testdata.txt, but this file is not in a repository. So we use the testRuntime dependency configuration to set a file dependency on our test execution.

dependencies {

compile 'log4j:log4j:1.2.17'

testCompile 'log4j:log4j:1.2.17'

testCompile 'junit:junit:[4.8,)'

testRuntime files('testFiles\\testdata.txt')

}

1. On the command line enter gradle build. This will build our application and run out tests. Notice Gradle goes to Maven central to locate and download our dependencies. Notice Gradle has performed transitive dependency resolution on JUnit. This is where our specified dependency has its own dependencies.
2. Our log4j properties files is in the main and test resources directories. The output file is configured to go to build/reports/log.txt. Open this file to see the output from the tests. Notice our unit test used the testdata.txt file to input test data.
3. The output from the unit test is in C:\GradleTraining\Exercise\_7\Exercise\_7.1\dependencies\build\reports\tests. Click on index.html to see the result our unit test. Drill down by clicking on the com.training.gradle package link, then the StoplightTest class link. The log4j configuration specifies both a log file and logging to standard out, so you can click on the standard out button here to see console output that was capture when Gradle ran our test.

**Expected Result**: Gradle project that uses Gradle to download the application and test compile and runtime dependencies and use them in the build.

End of Lab

**If you get done Early:**

* Read the Gradle User’s Manual chapter on dependency management at <https://docs.gradle.org/current/userguide/dependency_management.html>
* Review all the dependency configurations the Java plugin gives us to use at <https://docs.gradle.org/current/userguide/java_plugin.html#tab:configurations>

# Gradle Testing Integration (20 min)

Prerequisites

* Java JDK 1.5 or higher
* Gradle installed and verified
* Browser open to the Gradle JavaDoc page
* Work directory created as C:\GradleTraining\work\Exercise\_8. Note, the solutions are in directory C:\GradleTraining\Exercise\_8.
* Command prompt windows navigated to this new directory

Description

In this lab we will add JUnit tests to a sample project, then use Gradle to execute those tests as part of our build.

## Adding and Executing JUnit Tests

We are using our SwitchStopLight class for our application under our project directory under src\main\java\com\training\gradle. We also have a unit test class, StoplightTest under our project test directory at src\test\java\com\training\gradle. We’ll be seeing how easy it is to add and run unit test to a project with Gradle.

1. For this exercise we are going to reuse Exercise\_3.2 that we imported into the Eclipse environment. Bring up Eclipse Create a directory for this exercise named C:\GradleTraining\work\Exercise\_8\Exercise\_8.1
2. Copy the directory C:\GradleTraining\Exercise\_8\Exercise\_8.1\StopLight to your new work directory.
3. In command prompt navigate to this new work directory, StopLight.
4. There is a solution.gradle file in this directory, but you will be creating your build.gradle file from scratch. Create a build.gradle file in this directory and open it in Notepad++.
5. Add the following code to the file and save it. Here we are applying the Java Plugin and ensuring we have JUnit available for our compile. We don’t have to worry about a test run dependency because Junit will already have been installed into our project.

apply plugin: 'java'

repositories {

mavenCentral()

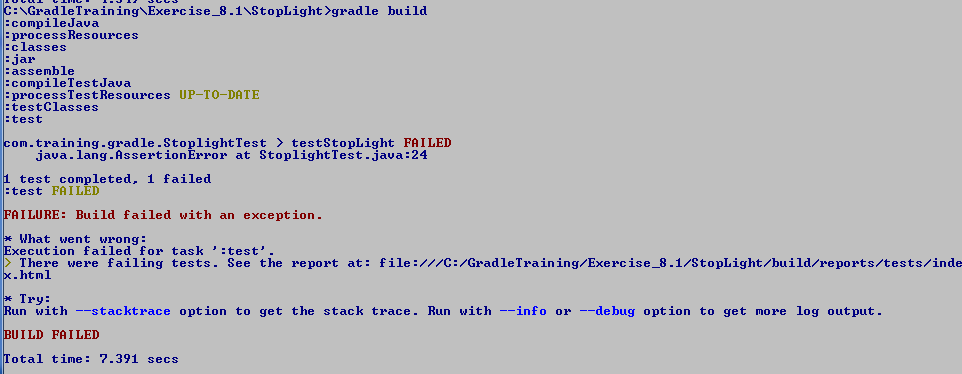
}

dependencies {

testCompile 'junit:junit:[4.8,)'

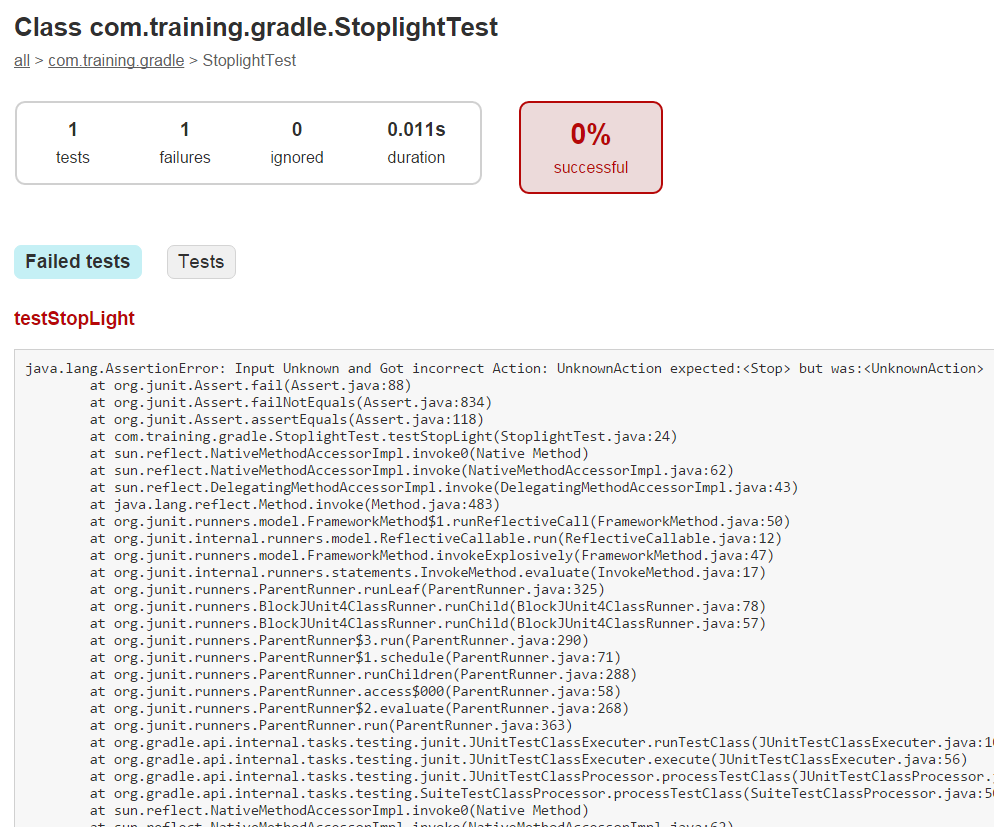
}

1. Now run the command gradle build. The Build task will compile and assemble our code and then run any existing unit test against it. We don’t have to do any configuration to run our unit tests, they run for free since we annotated our test method with @Test. As we see, we built and assembled our application just fine. But we had an issue with a failing unit test.



1. The output tells us to look at the index.html file in our project’s build/report/tests directory. Double click on this file to bring up the test results HTML page. It will show a link to our failed tests. Click on the link and the page will display the stack trace of the failed test.

This stack track shows we failed because our test of Unknown Color expected a Stop, but got UnknownAction. Obviously our test code is incorrect.



1. Open the unit test C:\GradleTraining\ Exercise\_8\Exercise\_8.1\StopLight\src\test\java\com\training\gradle\StoplightTest.javea and change the last test to be the following:

color = UnknownColor;

ssert.assertEquals("Input Unknown and Got incorrect Action: " +

light.returnAction(color), UnknownAction,

light.returnAction(color));

1. Not rerun the gradle build task. Notice the build and the tests now pass! Having unit tests make the build fail is very import as it tells the build server the code is not ready to move on to the next step. In Continuous Integration pipelines this is a very valuable feature. We’ll learn about Continuous Integration in the next section.

**Expected Result**: Gradle project with unit tests that run when the project it built, and cause the build to fail if the unit tests fail.

End of Lab

**If you get done Early:** Research the internet to learn about TestNG and how it is an improvement on our well-loved Junit.

# Gradle Integration with Jenkins (40 min)

Prerequisites

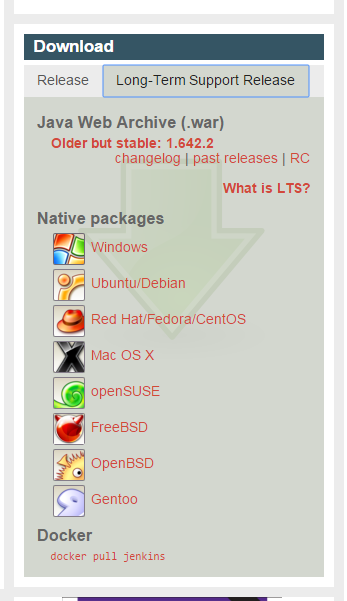
* Java JDK 1.5 or higher
* Gradle installed and verified
* Internet access
* Git installed and verified

Description

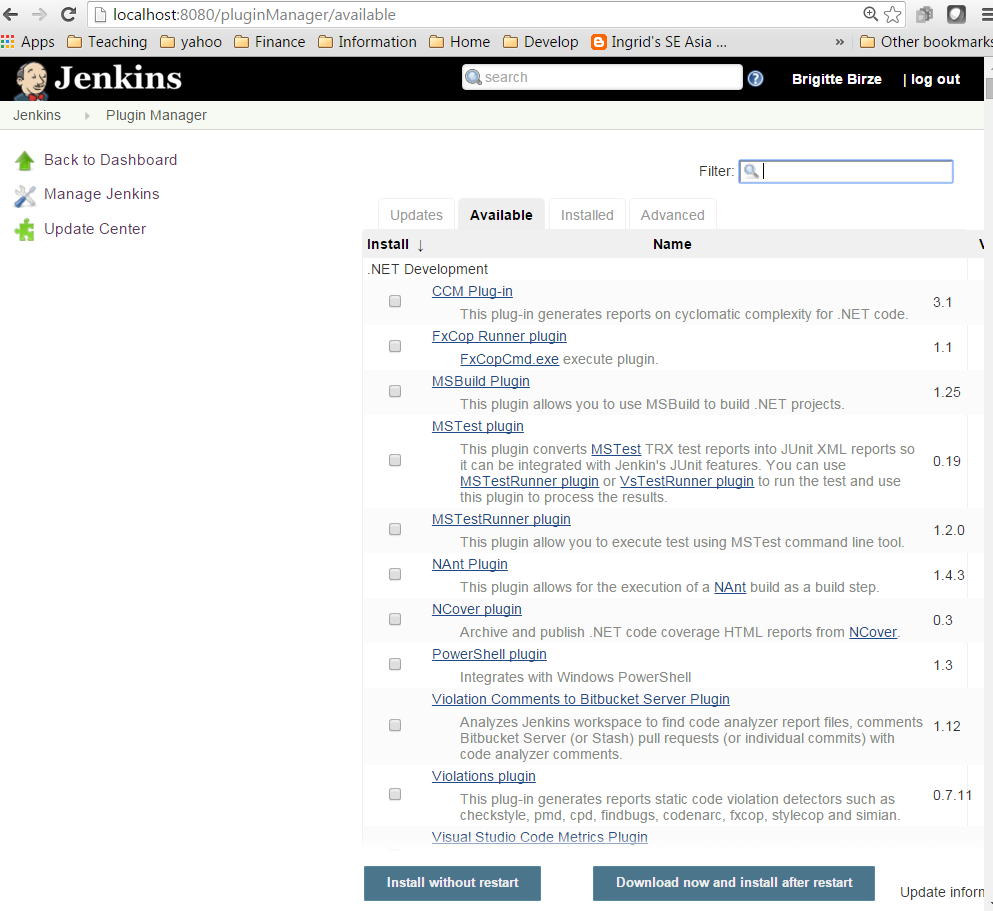
In this lab we will install and configure Jenkins to enable builds of Gradle projects stored in Git. We will then configure a Jenkins job to build our Gradle project from the Jenkins build server.

## Install and Configure Jenkins

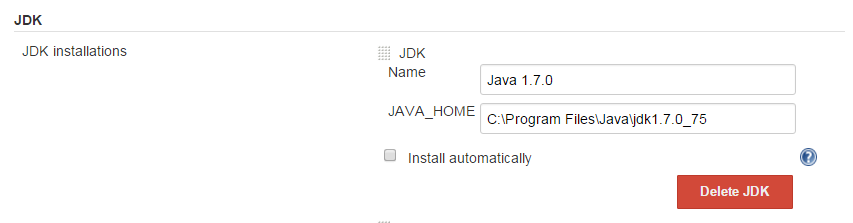
1. Open your browser and go to the page: <http://jenkins-ci.org/>.
2. Notice the download options on the right hand side of the page. Click on the Long-Term Support Release to get the more stable version of Jenkins. Click the link under the Java Web Archive (.war) heading.



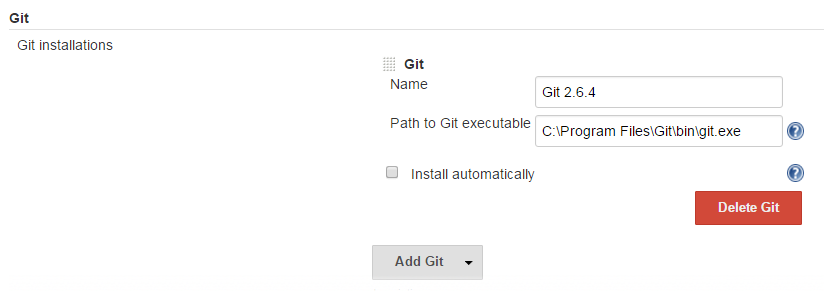
1. We are going to run Jenkins from the command line, so move the downloaded Jenkins.war file to its own directory under C:\jenkins.
2. Open a command prompt and navigate to C:\jenkins. At the prompt enter the following command to start the Jenkins server. C:\> java –jar Jenkins.war
3. Open your browser and navigate to <http://localhost:8080>. When Jenkins is fully up the Jenkin’s application will open.
4. We do not have security enabled so we’ll be able to do all our Jenkins configuration without needing to login. From the menu on the left side of the screen, Click on the Manage Jenkins option. This will display a number of administration tools. Click on the Manage Plugins tool.
5. The Manage Plugin screen has 3 tabs at the top of the screen. We need to go to the Available tab to install plugins for Git and Gradle.
6. Once in the Available tab, look for the Filter text box in the top right hand corner of the page. You’ll use this Filter to find our needed plugins.



1. Type the word “Gradle” in it and notice how the available plugins in the list are filtered. (do not hit return or you will be taken out of the Manage Plugin page). When you see the Gradle plugin, select its checkbox, and click on the “Install without restart” button at the bottom of the page. You will be taken to the installing plugin page.
2. On the left hand menu, click on Manage Plugins and then on the Available tab. In the Filter text box enter “Git plugin”. If you just enter Git there will be several screen of plugins to search through. You want the Git plugin, Not the Github or any other type of plugin with Git in the name. Select the Git plugin checkbox and click on the “Install without restart” button at the bottom of the page. This will take you to the installing plugin page.
3. When the plugins are fully installed you’ll need to restart Jenkins. Go to the command line where you started Jenkins, and hitting Cntl-c to stop it. Then enter C:\> java –jar Jenkins.war.
4. Not that we’ve installed our plugins, we need to configure the Jenkins server to be able to find them. On the left hand menu, click on Manage Jenkins, then click on Configure System, the first item in the list of administration tools.
5. This is a long page, and every plugin you install adds a new section to be configured. Start with the JDK section, click the “Add JDK” button in this section and unclick the “install automatically” checkbox. We need to tell Jenkins where our JAVA\_HOME is located and how we’ll refer to that installation when we configure build jobs. Enter the name Java 1.7, and the path to JAVA\_HOME. You can find the JAVA\_HOME path from the command prompt by entering C:\> set JAVA\_HOME. You can configure multiple versions of tools on your build server. Including the version in the name of the tool will help distinguish between multiple versions when you create your build job. Hit the Apply button at the bottom of the screen to save your changes.



1. Now scroll down to the Git section and click on the “Add Git” button and unclick the “install automatically” checkbox. Add a name for your Git installation and the path to your git.exe file. Note you need the full path including get.exe at the end as shown below. Hit the Apply button at the bottom of the screen to save your changes. BTW, you can click on those blue question mark circles to get pop-up help on most fields in Jenkins!



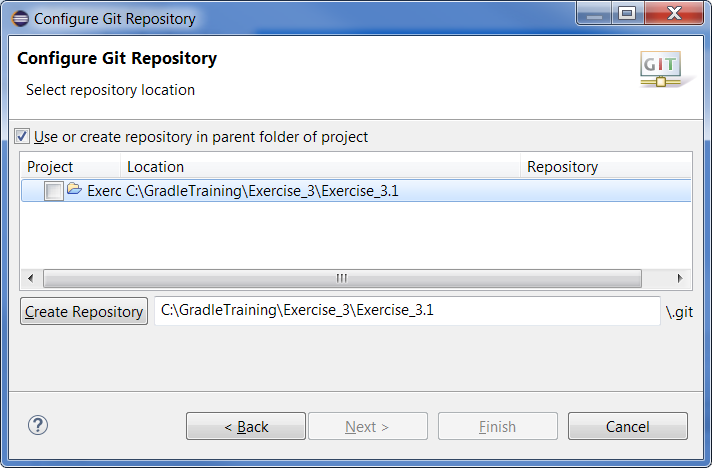
1. Scroll down to the Gradle section and click on the “Add Gradle “ button and unclick the “install automatically” checkbox. Add a name for your Gradle installation, including the version will help distinguish between multiple versions installed on your build server. Then add the path to your Gradle installation in the GRADLE\_HOME text box. Hit the Apply button at the bottom of the screen to save your changes.

**Expected Result**: Jenkins Continuous Integration Server installed and configured to do Java builds with Git and Gradle.

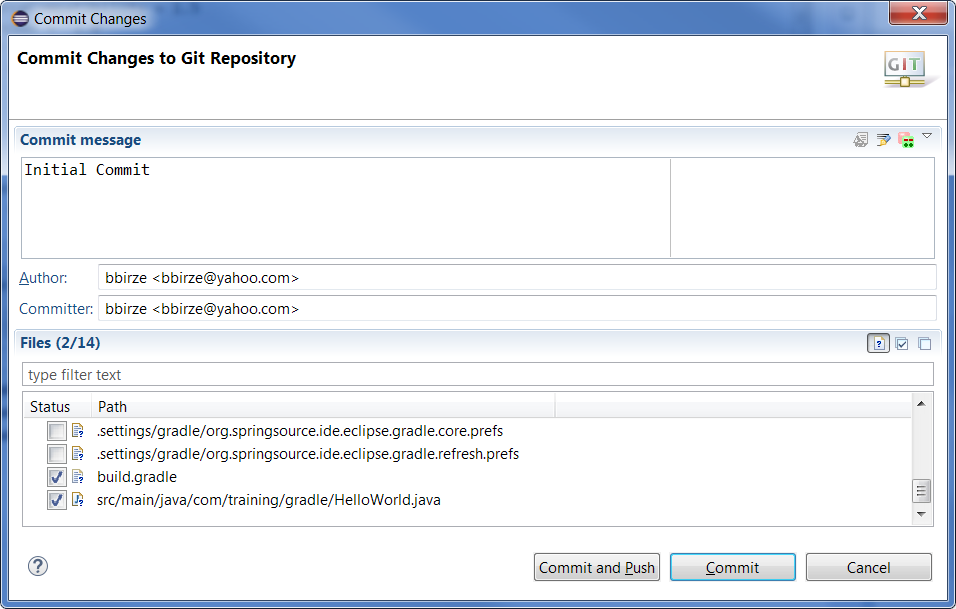
## Adding Git Configuration Management

We will be working with the project you created for Lab 3.2. For Jenkins to see and build this project, it needs to be under a source control management system. We’ll be using Git as our SCM and enabling it through Eclipse. We already imported our Gradle Lab 3.2 into Eclipse, so all we need to do is enable Git.

1. Open Eclipse and select the Lab 3.2 project we imported in Lab 4.1. Right click on the project and select Team->Share Project. This will bring up either the Share Project window or the Configure Git Repository window. If the Share Project window comes up, choose Git and Next to get to the Configure Git Repository window.
2. In the Configure Git Repository window select the “Use or create repository in parent folder or project” checkbox. Then select the project. This will allow you to click the Create Repository button.



1. Click the Create Repository button then click the Finish button.
2. Now we need to do our initial commit. Right click on the project in the Package Explorer and choose Team->Commit to bring up the Commit Changes window. Note, you may get an Identify Yourself window asking for your Full name and E-mail. Git uses this to correlate changes to the repository to an identity. Enter your name and an email address and hit OK to continue.
3. Add a Commit message and select the build.gradle and java source file as shown below. Then hit the Commit button. This gives us a baseline for our project in our local Git SCM repository.

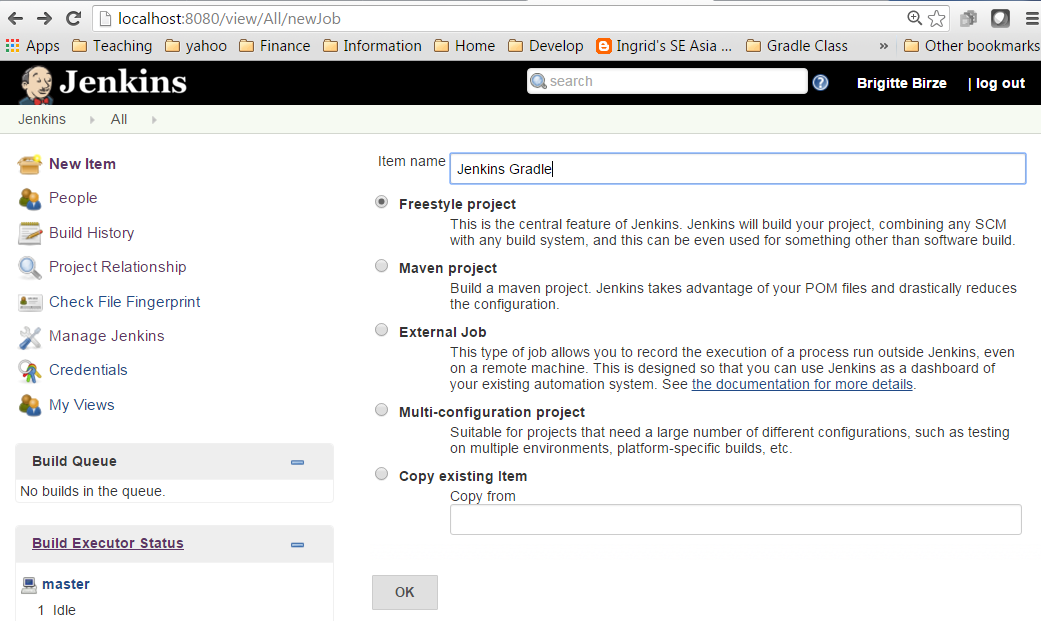


**Expected Result**: Gradle project controlled by local Git SCM repository

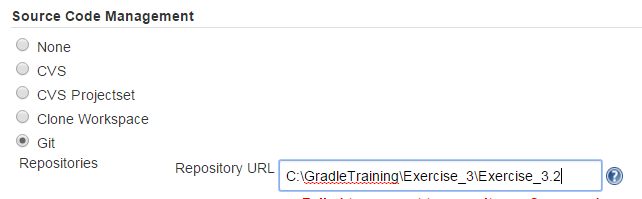
## Creating and Executing a Jenkins Build Job Using Gradle

We now have Jenkins configured for Java Gradle builds from a Git repository, and we have our Java Gradle project controlled in a Git repository. Everything is in place to create a Jenkins build job to pull our project from Git and build it whenever we make a change.

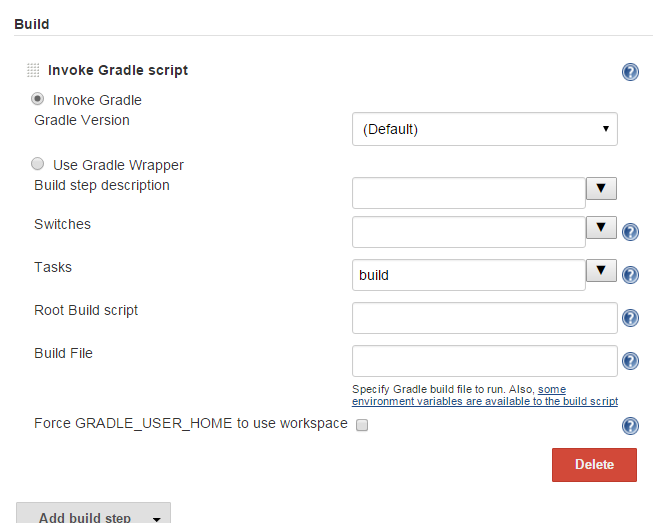
1. Ensure Jenkins is running and open the browser going to URL localhost:8080.
2. In the Jenkins dashboard left hand menu, select New Item. This will bring up the window to allow us to name and configure our new build job.
3. In the item name textbox enter build job “Jenkins Gradle”, and select Freestyle project. All Gradle project will be of type Freestyle. Select OK to bring up the job configuration page.



1. On the job configuration page scroll down to the Source Code Management section. Select the Git radio button. In the Repository URL text box enter the path to your Lab 3.2 project. Scroll to the bottom of the page to hit the Apply button to save your change.



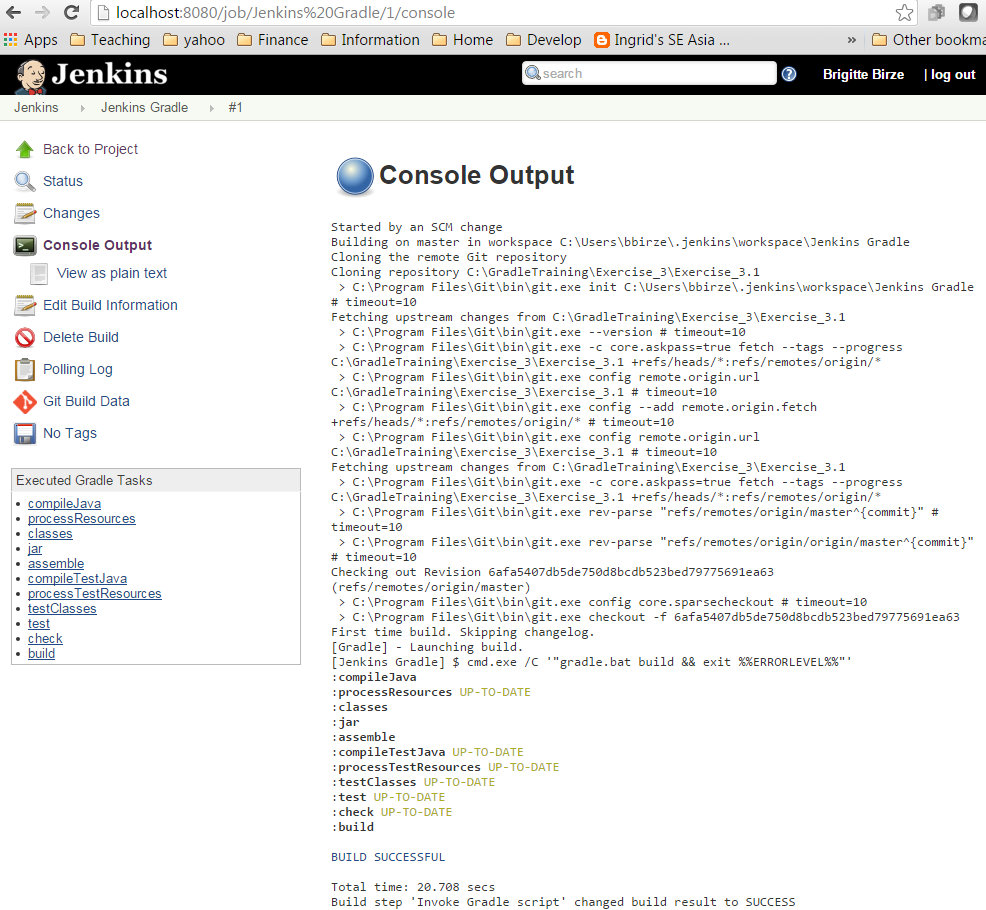
1. Scroll to the Build Triggers sections. Select the Poll SCM radio button. In the Schedule box type 5 asterisks with a space between each one, \* \* \* \* \*. This will make Jenkins poll our Git repository every minute to see if anything has be changed. If any committed file in project repository has changed, Jenkins will pull the entire project from GIT and perform the build steps we specify.
2. Now scroll to the Build section to specify a build Step. Click on the Add build step button and select Invoke Gradle Script. In the Gradle script configuration section, add the build task to the Tasks option.



1. That’s it! Save your changes and you’ll find yourself in the Project Jenkins Gradle page. Checkout the items in the left hand menu you can select for this build job.



1. Within a minute a build will start in the Build History section. Click on the build to get to the build page for this specific build. The left hand menu shows you the options you can select to drill down into this specific build. Click on Console Output in the left hand menu to see the output from the build.



1. Though developers will not be configuring the Jenkins server itself, they will be configuring their Jenkins build jobs. Click around in the Build page and Project page to understand the different options available to you as you configure your build jobs and investigate the outcome of each build. If you like, change the code in the project from Eclipse and commit the changes to trigger off another Jenkins build.

**Expected Result**: Gradle project configured in Jenkins server and built automatically whenever a new commit happens in the Project’s Git repository.

End of Lab