GRADLE SUCCINCTLY

BY **JOSÉ ROBERTO OLIVAS MENDOZA**

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Gradle Succinctly

By
José Roberto Olivas Mendoza

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

About the Author

I’m an IT business entrepreneur, a software developer, and a huge technology fan. My company went to market in 1990, focused mostly in custom software development. We started with COBOL as our main programming language, and we’ve been evolving along the years up to .NET and Microsoft Office technologies. Nowadays, with mobile devices and the Internet being the cutting edge of Information Technologies, applications development must be faster and accurate in order to comply with customers’ needs and to deliver reliable products. Using tools to automate certain tasks in the development process, such as building or testing an application, is very important to accomplish those goals.

We heard about Gradle for the first time when we started Android applications development with Android Studio, because it is included by default. At the beginning, we focused on creating applications with the configuration provided by Android Studio itself, with some minor changes. Later, a member of our development team realized this tool could be extended in order to integrate the entire development lifecycle. When he came to me to talk about what he had found regarding Gradle, I encouraged him to do some deep research and document how Gradle could help us to improve our development process. He did it, and his work was so good that the entire team was convinced of Gradle’s worth.

Now, Gradle is an essential tool for our mobile applications development, and it has improved our development lifecycle. As a result, we’re delivering our products faster and much more reliably, and our revenue for these kinds of projects has grown.

Who Is This Book For?

This book is written primarily for those IT professionals who want to learn about Gradle as an automated build system for the Java world. The book starts with an introduction to Gradle, its history, and a detailed overview.

The second chapter covers the Gradle installation process, including prerequisites and where to get Gradle distributions. The rest of the book shows you how to implement and integrate Gradle into projects, starting with some basics and ending with a detailed explanation about tasks, which are the fundamental units for building activity.

Then, we'll get further into Gradle, exploring some capabilities like hooks, dependency management and, of course, testing. I'll also explain how to deal with multi-project builds.

While Gradle is multiplatform, all examples and use cases shown throughout this book have been developed on Windows, with a focus on Java. Gradle 2.12 (the latest release at the time this e-book was written) is used for the purposes of this book, and Visual Studio Code is the text editor I employed.

Visual Studio Code can be downloaded here, and all samples described in this book can be downloaded here.

I hope that by the end of this book, IT professionals who are not familiar with Gradle feel encouraged to use it to improve their projects' development cycle in order to shrink delivery times and build more reliable products.

Chapter 1 Introduction

What is Gradle?

Gradle is an open-source build automation system that was conceived upon a Groovy-based[1] domain-specific language[2] (DSL). Gradle was designed for multi-project builds, which can grow to be quite large. It supports incremental builds by intelligently determining which parts of the project are up-to-date, so that any task dependent upon those parts will not be re-executed.

The initial release of Gradle was published in 2007, and the project is considered to have active development status. The current stable release (at the time of writing) was published in March of 2016.

Gradle is being heralded as more than a build tool, but also as a means for automating the compilation, test, and release process. Qualified by its developers as a quantum leap for building technology in the Java world, some of Gradle's features are:

- **Declarative builds and build-by-convention:** Based on a rich, extensible Domain Specific Language (DSL) based on Groovy, Gradle provides declarative language elements that the user can assemble as desired. Those elements also provide build-by-convention support for Java, Groovy, OSGi, Web, and Scala projects. The declarative language is extensible, allowing you to add new language elements or enhance the existing ones.
- **Language for dependency-based programming:** The declarative language lies on top of a general purpose task graph. It provides utmost flexibility to adapt Gradle to specific or unique development needs.
- **Custom-structured builds:** Gradle allows you to apply common design principles to a build. For example, it's very easy to compose a build from reusable pieces of build logic, creating a well-structured and easily maintained build.
- **Multiproject builds:** Gradle allows users to model project relationships in a multi-project according to the user's layout. Also, it provides partial builds, so that when a single subproject is built, Gradle takes care of building all subprojects that subproject depends on. The user can choose to rebuild the subprojects that depend on a specific subproject.
- **Groovy:** Build scripts are written in Groovy. This is intended to orientate Gradle to be used as a language, not as a framework.
- **Free and open source:** Gradle is an open-source project, licensed under the ASL (Apache Software License), which can be viewed here.

1 Object-oriented programming language for the Java platform. It can be used as a scripting language, and is dynamically compiled to Java Virtual Machine (JVM) bytecode.

Why are scripts written in Groovy?

Since Gradle’s main focus is Java projects, the team members for those projects are very familiar with Java. Gradle developers thought that a build script should be as transparent as possible to all team members. They didn’t use Java because its limitations prevent a nice, expressive, and powerful build language.

Groovy was chosen because it provides the greatest transparency for Java people. Its base syntax is the same, as well as its type system and other things. Thus, every Gradle build file is an executable Groovy script. For Gradle beginners, being aware that they’re writing Groovy code is not needed. But when your needs become more sophisticated, the power of the Groovy language may become very important. In that way, Gradle’s Groovy-based build files allow the users to do general-purpose programming tasks in their build file, offering control flow and execution of nonstandard tasks within the build process.

About domain-specific build languages

At some point, every developer who has been maintaining a complex build wants to write a little bit of code for a build file. Sometimes, the unlimited ability to code a build according to the user-desired way can result in a catastrophe of maintainability. Although Gradle build files are Groovy scripts, Gradle itself intends to present the user not with mere Groovy, but with a domain-specific language (DSL) oriented to the task of building code. So a new Gradle user could learn this language with no previous knowledge about Groovy, and use Gradle effectively. This DSL describes the build using idioms appropriate for performing the task of building software, and not necessarily for general-purpose programming. In this way, Gradle gently nudges the user toward using the idioms of its DSL first, even when general-purpose coding is always available.

About Gradle Inc.

Gradle Inc. is a Silicon Valley startup that develops, distributes, and supports the Gradle open-source project. Its mission statement is “to transform how software is built and shipped.” Also, it claims to exist “to end once-and-for-all the worst things about big software and restore the reason the user got into coding in the first place.”

Besides supporting Gradle, the company offers services related to Gradle, build automation, and continuous integration pipelines. According to its website, some of these services are:

- **Build migration:** Create a seamless build migration to Gradle and replicate custom build functionality, including Ant or Maven scripts.
- **CD optimization:** Leverage Gradle to automate your continuous delivery pipeline and best practices.
- **Performance tuning:** Improve build performance and developer productivity.
- **Standardization:** Get help to centralize your Gradle build infrastructure.
- **Build review:** Review your current Gradle build infrastructure with a core developer, and get recommendations on improvements.

- **Integration:** Integrate Gradle with your existing tools and platforms.
- **Plugin development:** Identify opportunities to leverage Gradle plugins to put in place a manageable build infrastructure.

Chapter summary

Gradle is an open-source build automation system that is conceived upon a Groovy-based domain-specific-language (DSL). Gradle was first released in 2007, and the project is considered to have an active development status. This e-book uses release 2.12, published on March 14, 2016.

Gradle is being heralded as a build tool, and also a tool for automating the compilation, test, and release process. Gradle's main focus is Java projects, and for that reason its developers chose Groovy because it provides the greatest transparency for Java people. Its base syntax is the same, as well as its type system and other things.

Although Gradle build files are Groovy scripts, Gradle itself intends to present the user not with mere Groovy, but with a domain-specific language (DSL) specifically oriented to the task of building code. So a new Gradle user could learn this language with no previous knowledge of Groovy, and use Gradle effectively.

Gradle Inc. is a Silicon Valley startup that develops, distributes, and supports Gradle. It offers services around Gradle, build automation, and continuous integration pipelines.

Chapter 2 Installing Gradle

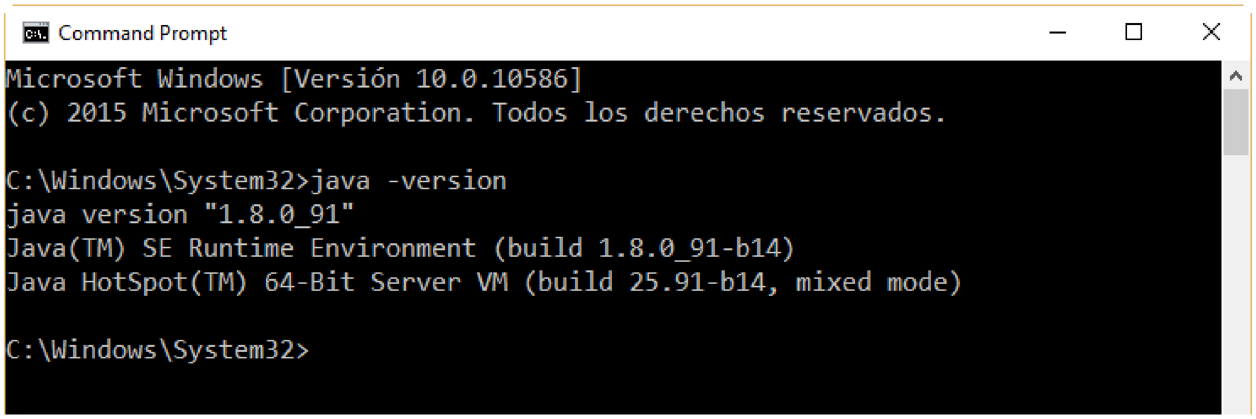
Getting started

Prerequisites

Gradle requires a Java JDK or JRE version 7 or higher to be installed first. There is no need to install Groovy because Gradle ships with its own Groovy library; any existing installation of Groovy is ignored by Gradle.

Java JDK can be downloaded here.

To check the Java version, open a Command window and issue the **`java -version`** command.



```
Microsoft Windows [Versión 10.0.10586]
(c) 2015 Microsoft Corporation. Todos los derechos reservados.

C:\Windows\System32>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)

C:\Windows\System32>
```

Figure 1: Checking Java version

Getting Gradle

Any Gradle distribution can be downloaded here. Every distribution comes packaged as a .zip file. A full distribution contains:

- The Gradle binaries
- The user guide in both HTML and PDF formats.
- The DSL reference guide.
- The API documentation, both Javadoc and Groovydoc.
- Extensive samples, along with some complete and more complex builds that can be used as a starting point for custom builds.
- The binary sources, for reference only. If the user wants to build Gradle, the source distribution needs to be downloaded (or check out the sources from the source repository). The Gradle website contains further details.

Gradle installation process

To install Gradle on Windows, follow these steps:

1. Unzip the distribution downloaded from Gradle's website to a folder location. It's suggested to name this folder location something similar to **C:\gradle-2.12**.
2. Right-click on the **My Computer** (or **This PC** in Windows 10) icon and select **Properties**.

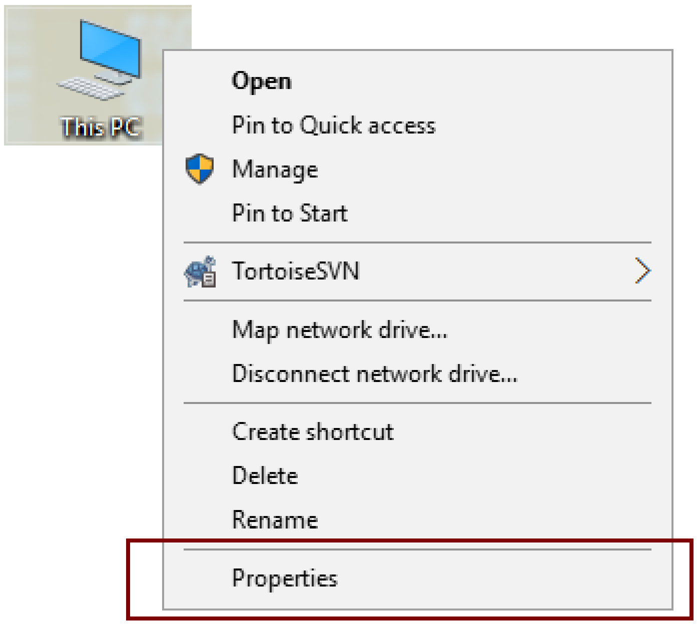
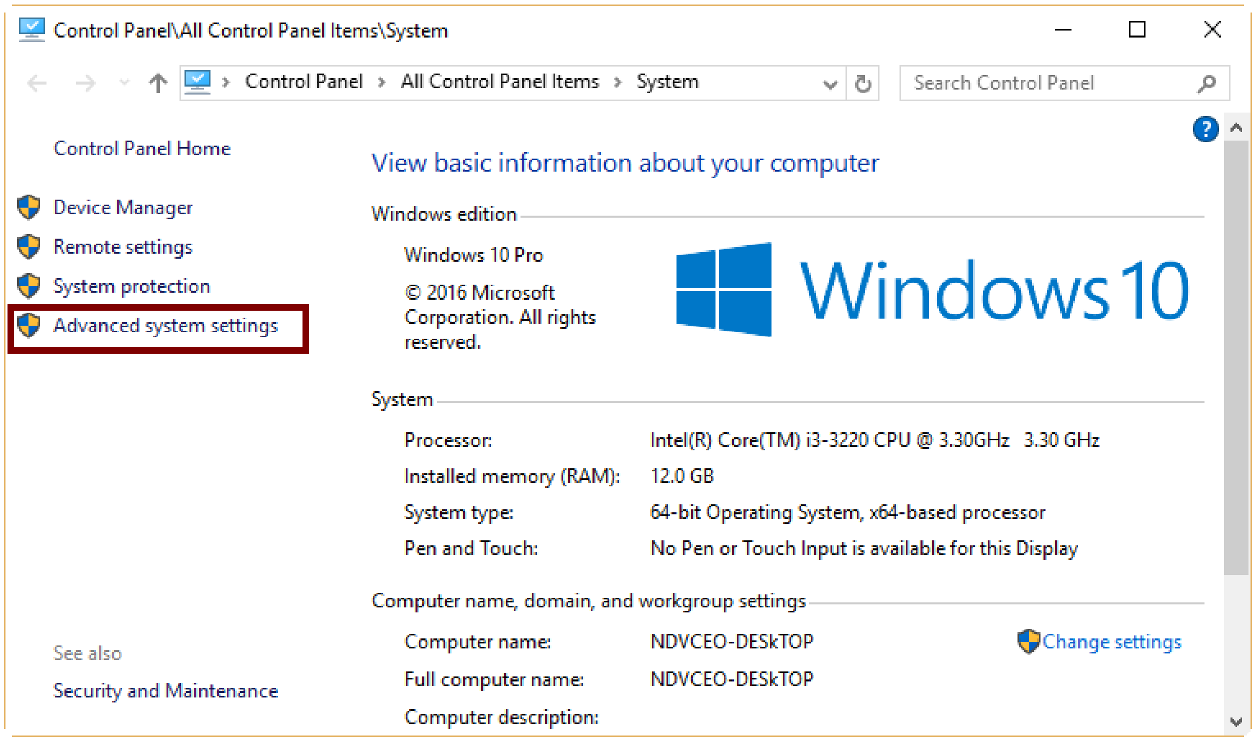


Figure 2: This PC Context Menu

3. Select **Advanced System Settings** from the links displayed at the left of the System Control Panel window.



Control Panel\All Control Panel Items\System

Control Panel > All Control Panel Items > System

Search Control Panel

Control Panel Home

Device Manager

Remote settings

System protection

Advanced system settings

See also

Security and Maintenance

View basic information about your computer

Windows edition

Windows 10 Pro

© 2016 Microsoft Corporation. All rights reserved.

Windows 10

System

Processor:	Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz 3.30 GHz
Installed memory (RAM):	12.0 GB
System type:	64-bit Operating System, x64-based processor
Pen and Touch:	No Pen or Touch Input is available for this Display

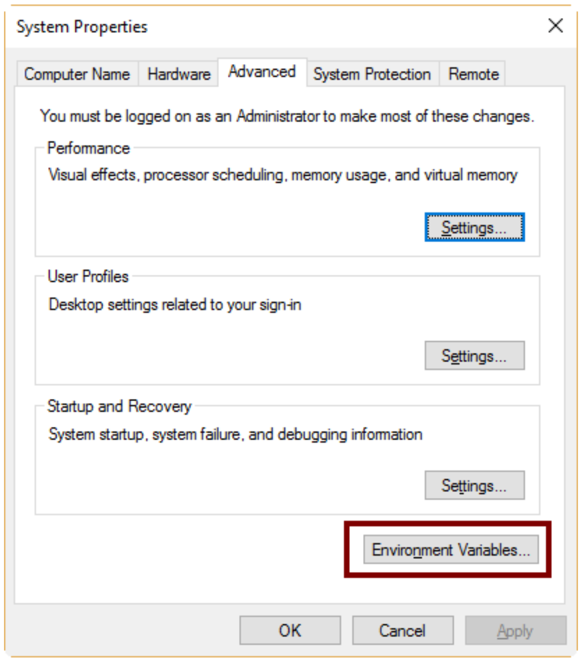
Computer name, domain, and workgroup settings

Computer name:	NDVCEO-DESkTOP
Full computer name:	NDVCEO-DESkTOP
Computer description:	

Change settings

Figure 3: The System Control Panel Window

4. Click on the **Environment Variables** button, located at the bottom of the Advanced System Settings dialog box.



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as an Administrator to make most of these changes.

Performance

Visual effects, processor scheduling, memory usage, and virtual memory

Settings...

User Profiles

Desktop settings related to your sign-in

Settings...

Startup and Recovery

System startup, system failure, and debugging information

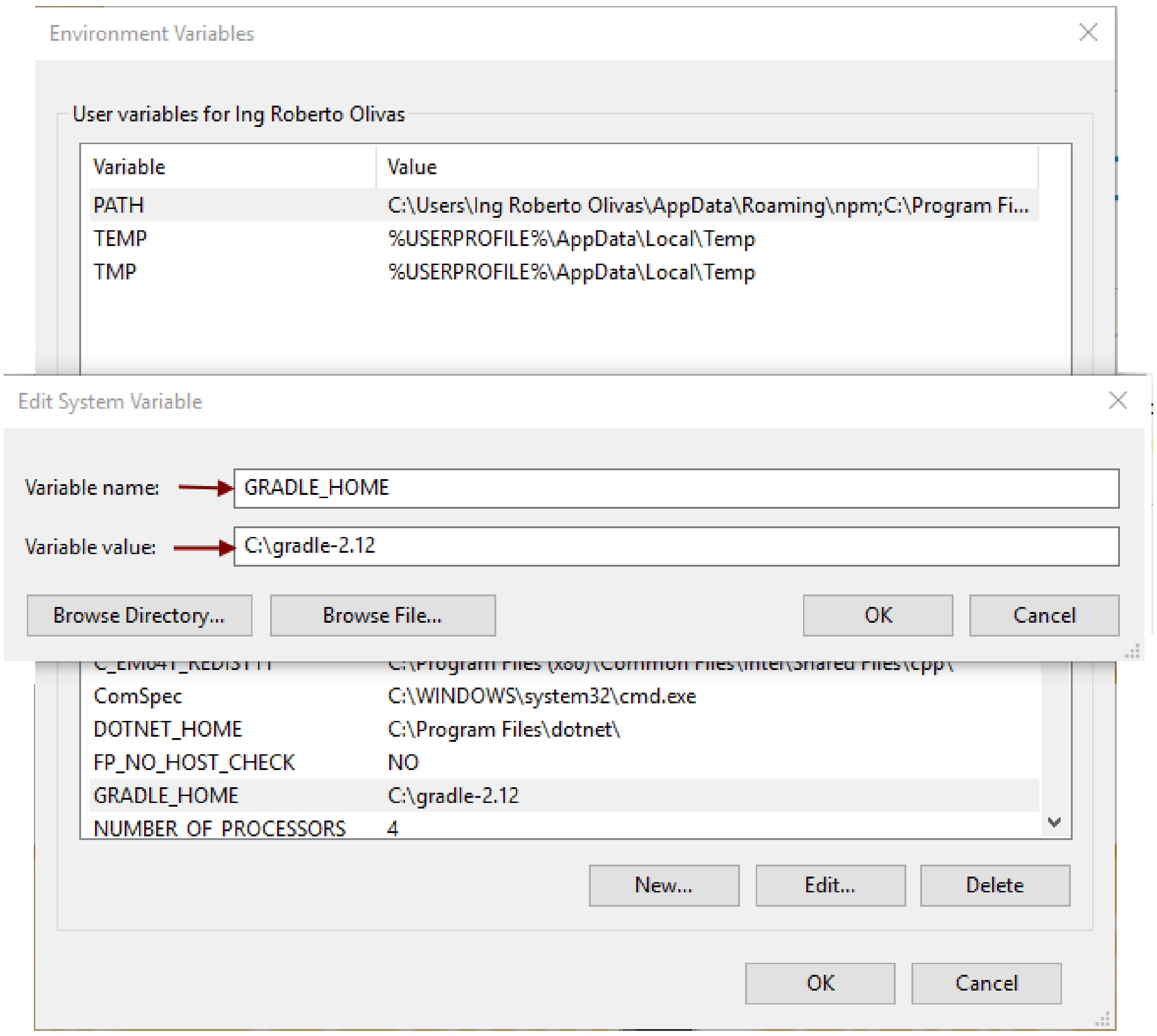
Settings...

Environment Variables...

OK | Cancel | Apply

Figure 4: Advanced System Settings Dialog Box

5. In the **Environment Variables** dialog box, click **New** under System Variables. Name the environment variable **GRADLE_HOME**, and give it the value **C:\gradle-2.12**.


Environment Variables

User variables for Ing Roberto Olivas

Variable	Value
PATH	C:\Users\Ing Roberto Olivas\AppData\Roaming\npm;C:\Program Fi...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Edit System Variable

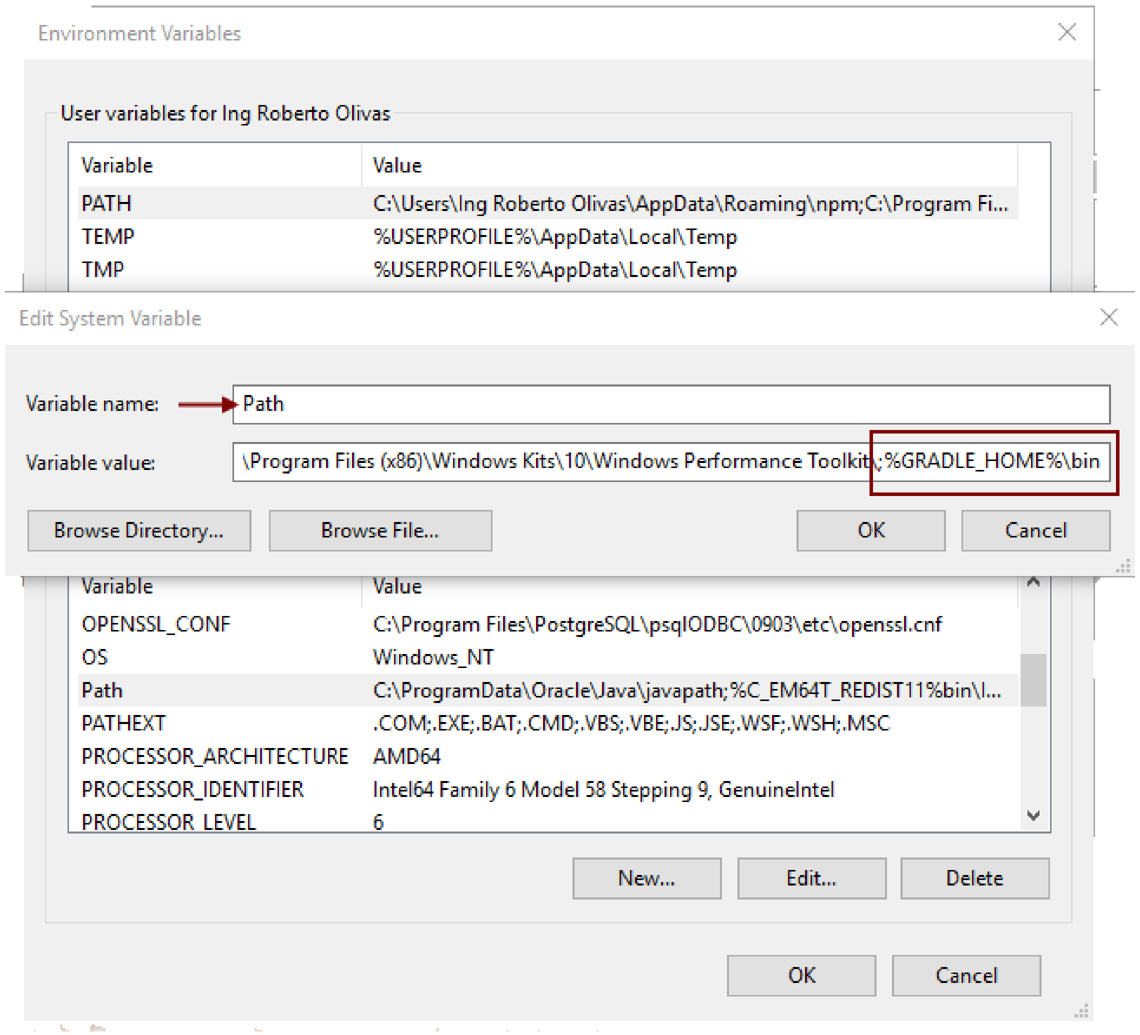
Variable name: GRADLE_HOME

Variable value: C:\gradle-2.12

ComSpec	C:\WINDOWS\system32\cmd.exe
DOTNET_HOME	C:\Program Files\dotnet\
FP_NO_HOST_CHECK	NO
GRADLE_HOME	C:\gradle-2.12
NUMBER_OF_PROCESSORS	4

Figure 5: Adding the GRADLE_HOME Environment Variable

6. Select the **Path** variable located under System Variables, in the same dialog box. Click **Edit** and add the text **;%GRADLE_HOME%\bin** at the end of the variable value list.


Environment Variables

User variables for Ing Roberto Olivas

Variable	Value
PATH	C:\Users\Ing Roberto Olivas\AppData\Roaming\npm;C:\Program Fi...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Edit System Variable

Variable name: Path

Variable value: \Program Files (x86)\Windows Kits\10\Windows Performance Toolkit\;%GRADLE_HOME%\bin

Variable	Value
OPENSSL_CONF	C:\Program Files\PostgreSQL\psqlODBC\0903\etc\openssl.cnf
OS	Windows_NT
Path	C:\ProgramData\Oracle\Java\javapath;%C_EM64T_REDIST11%bin\I...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 58 Stepping 9, GenuineIntel
PROCESSOR_LEVEL	6

Figure 6: Editing the Path Environment Variable

Note: The screenshots shown here are taken from Windows 10, so they might look different than your screen, depending on your Windows version.

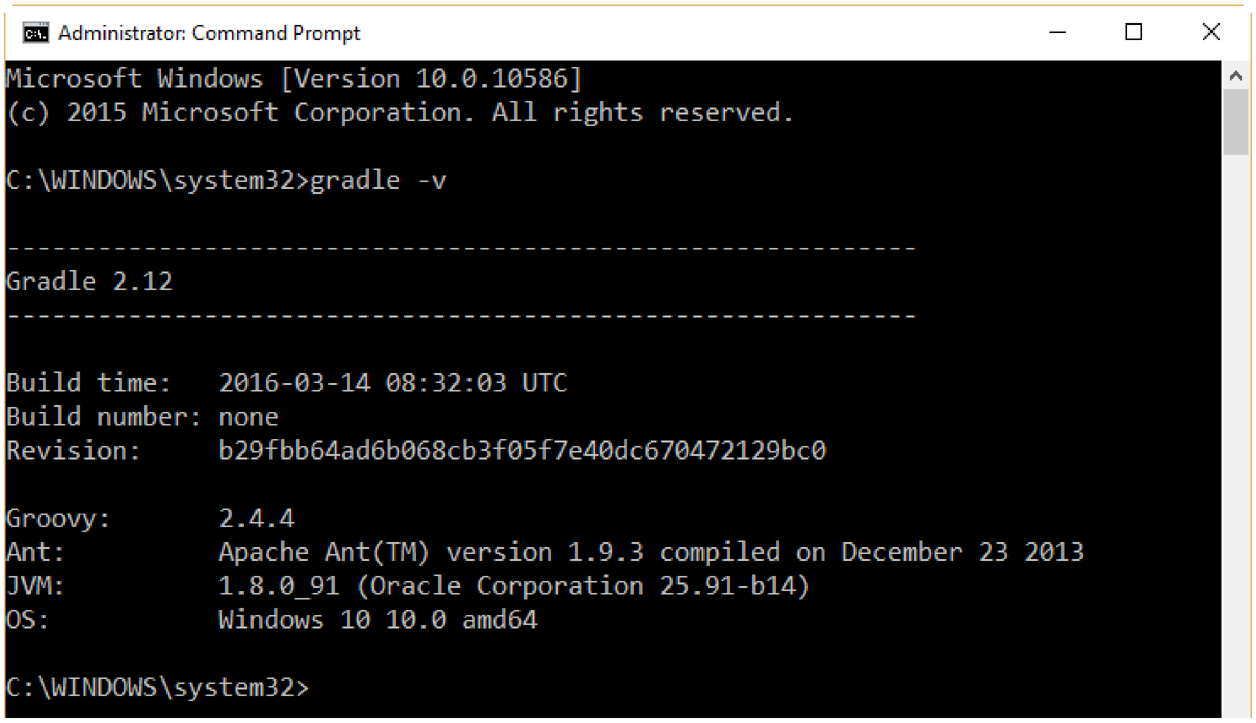
Testing the installation

After the installation process, you need to check whether Gradle is properly installed. Gradle runs via the **`gradle`** command, so open a command prompt window and type the following:

Code Listing 1

```
gradle -v
```

The output for this command shows the Gradle version and the local environment configuration (Groovy, JVM version, OS version, etc.). The displayed Gradle version should match the distribution downloaded by the user. For the purposes of this book, the Gradle version is 2.12.



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>gradle -v

------------------------------------------------------------
Gradle 2.12
------------------------------------------------------------

Build time:   2016-03-14 08:32:03 UTC
Build number: none
Revision:     b29fbb64ad6b068cb3f05f7e40dc670472129bc0

Groovy:       2.4.4
Ant:          Apache Ant(TM) version 1.9.3 compiled on December 23 2013
JVM:          1.8.0_91 (Oracle Corporation 25.91-b14)
OS:           Windows 10 10.0 amd64

C:\WINDOWS\system32>
```

Figure 7: The output for gradle -v command

Chapter summary

Java JDK or JRE version 7 or higher must be installed prior to Gradle installation. It's not necessary to install Groovy, because Gradle ships with its own Groovy library.

Gradle can be downloaded from here as a .zip file. This file contains all necessary files to install Gradle in the computer, including Gradle binary files, reference documents, and samples.

To install Gradle, the user needs to decompress the distribution .zip file into a folder. It is suggested to name this folder to something similar to **C:\ gradle-2.12**. After that, an Environment Variable named **GRADLE_HOME** must be added, and its value set to the name of the installation folder (**C:\ gradle-2.12**). Also, the Path Environment Variable must be edited to add the name of the **GRADLE_HOME** variable to its value, in the format: **;%GRADLE_HOME%.**

To test the installation process, the user needs to execute the command `gradle -v` from a command prompt window. The output for this command shows the Gradle version and the local environment configuration. The displayed Gradle version should match the distribution downloaded by the user. By the time you read this e-book, the current version of Gradle will be different, so you can use either the 2.12 version or a newer version.

Chapter 3 Beginning with Gradle

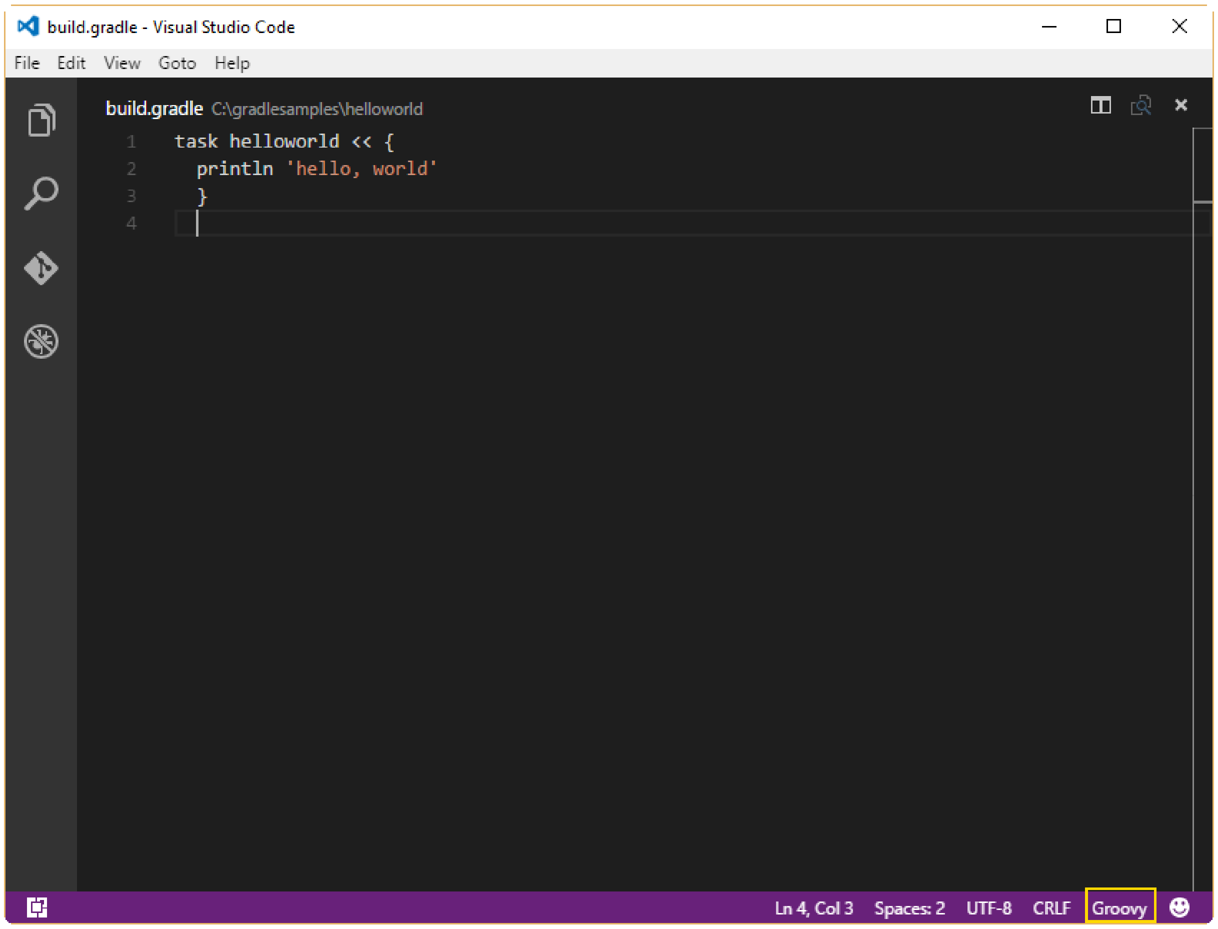
The ever-present Hello World

The first meeting with Gradle will take place by creating and running the traditional Hello World file. To do this, type the following sample using a text editor.

Code Listing 2

```
task helloworld << {
   println 'hello, world'
}
```

Now, to run the previous code, the user needs to save the file, naming it **build.gradle**. For the purposes of this book, all samples will be saved in a folder called **C:\gradlesamples**, and there will be a subfolder within for each one of the samples. For this case, a subfolder named **helloworld** will be created to store the file.


build.gradle - Visual Studio Code
File Edit View Goto Help
build.gradle C:\gradlesamples\helloworld
Ln 4, Col 3 Spaces: 2 UTF-8 CRLF Groovy

```
task helloworld << {
  println 'hello, world'
  }
```

Figure 8: The file C:\gradlesamples\helloworld\build.gradle being edited with Visual Studio Code

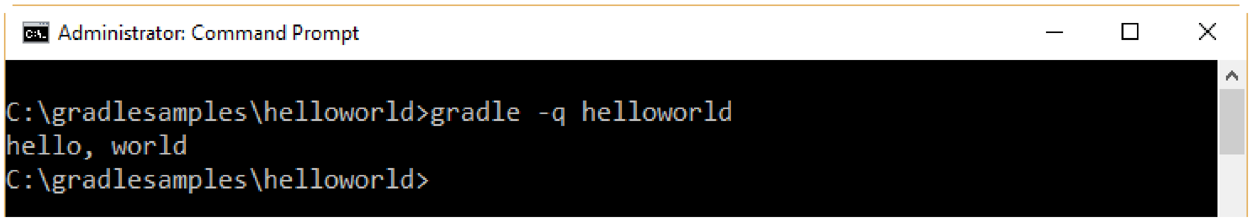
The previously mentioned file is known as a build script, and it is the container for all Gradle build statements. As mentioned in Chapter 1, all these statements correspond to the Groovy language. This can be noticed in the highlighted area of Figure 8.

In this example, the build script doesn't do much, other than printing out a sample message. To run the Hello World sample, open a command prompt window and get into the **C:\gradlesamples\helloworld** folder. After that, issue the following command.

Code Listing 3

```
gradle -q helloworld
```

The output produced by this command is shown in the following figure.



```
C:\gradlesamples\helloworld>gradle -q helloworld
hello, world
C:\gradlesamples\helloworld>
```

Figure 9: Output for Hello World sample

Build scripts and builds

For the purposes of this book, *build scripts* will refer to every **build.gradle** file used in the examples, and *build* will mean the execution process of those scripts.

Going a little further with Hello World

Looking into the Hello World example, you can see that the first line of code starts with a **`task`** statement. This statement will be explained in detail later in this book. For now, it's enough to know that **`task`** is one of the two basic concepts on which Gradle sits.

For the next example, you need to create a folder called **helloworldv2**. Then, the file **build.gradle** will be created in this folder with the following code.

Code Listing 4

```
// hello world v2
task hello << {
  print 'hello, '
}

task world(dependsOn: hello) << {
  println 'world'
}
```

To run this build script, execute the second task (**`world`**) by issuing the following command.

Code Listing 5

```
gradle -q world
```

The output is similar to the first example. The difference is that two tasks are executed to produce the same output.

When the **`world`** task is executed, the **`dependsOn`** statement (declared within the parentheses) tells Gradle that a task named **`hello`** should be executed before the **`world`** task. Then, the **`hello`** task produces the “hello” output. The **`print`** command leaves the cursor in the same line, after the white space position. Now, the **`world`** task is executed producing the “world” output and a line feed, followed by a carriage return.

The **`dependsOn`** statement declared in the code means that *dependencies* are being used for this build script. Dependencies will be explained in more detail later in this book.

Using the Gradle command line

The previous examples have been executed from a command prompt window. In other words, the **`gradle`** command line was used to run these build scripts. The **`gradle`** command line will be used in this book, and the options needed to be productive right away are explained in the following table.

Table 1: Gradle Command Line Options

Option	Description
`--help or -h`	Displays the help messages, which describe all command-line options.
`--info or -i`	Sets the Gradle log level to **`INFO`**, which causes a few more information messages to be displayed during the build script execution.
`-Dproperty=value`	Defines a system property; this is a mechanism for passing parameters into a build from the command line.

Option	Description
`--debug or -d`	Turns on debug logging for the build script. This generates a lot of output, but can be quite useful for troubleshooting build problems.
`--dry-run or -m`	Evaluates and runs the build script, but does not execute any task actions.
`--quiet or -q`	Supresses most output, showing error messages only.
`--gui`	Launches the Gradle GUI.
`properties`	Displays all the properties of the build's **`Project`** object. The **`Project`** object represents the structure and state of the current build.
`tasks --all`	Displays a list of all tasks available in the current build script. Plugins may introduce tasks of their own, so this list may be longer.
`--profile`	Writes a report that contains timing information about build script execution.

Displaying help messages

To show all help text associated with command-line options, issue the following command.

Code Listing 6

```
gradle -h
```

The output should look like this:

Code Listing 7

```
USAGE: gradle [option...] [task...]

-?, -h, --help          Shows this help message.
-a, --no-rebuild        Do not rebuild project dependencies.
-b, --build-file        Specifies the build file.
-c, --settings-file     Specifies the settings file.
```

```
--configure-on-demand   Only relevant projects are configured in this build
run. This means faster build for large multi-project builds. [incubating]
--console               Specifies which type of console output to generate.
Values are 'plain', 'auto' (default) or 'rich'.
--continue              Continues task execution after a task failure.
-D, --system-prop       Set system property of the JVM (e.g. -
Dmyprop=myvalue).
-d, --debug             Log in debug mode (includes normal stacktrace).
--daemon                Uses the Gradle daemon to run the build. Starts the
daemon if not running.
--foreground            Starts the Gradle daemon in the foreground.
[incubating]
-g, --gradle-user-home  Specifies the gradle user home directory.
--gui                   Launches the Gradle GUI.
-I, --init-script       Specifies an initialization script.
-i, --info              Set log level to info.
-m, --dry-run           Runs the builds with all task actions disabled.
--max-workers           Configure the number of concurrent workers Gradle
is allowed to use. [incubating]
--no-color              Do not use color in the console output. [deprecated
- use --console=plain instead]
--no-daemon             Do not use the Gradle daemon to run the build.
--offline               The build should operate without accessing network
resources.
-P, --project-prop      Set project property for the build script (e.g. -
Pmyprop=myvalue).
-p, --project-dir       Specifies the start directory for Gradle. Defaults
to current directory.
--parallel              Build projects in parallel. Gradle will attempt to
determine the optimal number of executor threads to use. [incubating]
--parallel-threads      Build projects in parallel, using the specified
number of executor threads. [deprecated - Please use --parallel, optionally
in conjunction with --max-workers.] [incubating]
--profile               Profiles build execution time and generates a
report in the <build_dir>/reports/profile directory.
--project-cache-dir     Specifies the project-specific cache directory.
Defaults to .gradle in the root project directory.
-q, --quiet             Log errors only.
--recompile-scripts     Force build script recompiling.
--refresh-dependencies  Refresh the state of dependencies.
--rerun-tasks           Ignore previously cached task results.
-S, --full-stacktrace   Print out the full (very verbose) stacktrace for
all exceptions.
-s, --stacktrace        Print out the stacktrace for all exceptions.
--stop                  Stops the Gradle daemon if it is running.
-t, --continuous        Enables continuous build. Gradle does not exit and
will re-execute tasks when task file inputs change. [incubating]
-u, --no-search-upward  Don't search in parent folders for a
settings.gradle file.
```

```
-v, --version            Print version info.
-x, --exclude-task       Specify a task to be excluded from execution.
```

Making it a little bit easier: task-name abbreviation

When tasks are specified in the command line, the user doesn't have to type the full name of those tasks. It's enough to provide a portion of the task name, which uniquely identifies this task within the build script.

For the helloworldv2 example, the following command can be used to run the build script.

Code Listing 8

```
gradle -q wo
```

If the abbreviation is not unique within the build script, Gradle will throw a warning. For example, consider a build script like the following.

Code Listing 9

```
task helloTask << {
   println 'helloTask output'
}

task hello << {
  print 'hello, '
}

task world(dependsOn: hello) << {
  println 'world'
}
```

If the user executes the following command.

Code Listing 10

```
gradle -q h
```

Gradle will throw the following output.

Code Listing 11

```
FAILURE: Build failed with an exception.

* What went wrong:
```

```
Task 'h' is ambiguous in root project 'helloworldv2'. Candidates are:
'hello', 'help'.

* Try:
Run gradle tasks to get a list of available tasks. Run with --stacktrace
option to get the stack trace. Run with --info or --debug option to get
more log output.
```

Tip: It's useful to name tasks in order to take advantage of task abbreviation.

Listing the tasks for a build script

You can get a report for all the tasks belonging to the build script, by executing the following command.

Code Listing 12

```
gradle -q tasks --all
```

The output should look like the following.

Code Listing 13

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

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

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root
project 'helloworldv2'.
components - Displays the components produced by root project
'helloworldv2'. [incubating]
dependencies - Displays all dependencies declared in root project
'helloworldv2'.
dependencyInsight - Displays the insight into a specific dependency in root
project 'helloworldv2'.
help - Displays a help message.
model - Displays the configuration model of root project 'helloworldv2'.
[incubating]
projects - Displays the sub-projects of root project 'helloworldv2'.
```

```
properties - Displays the properties of root project 'helloworldv2'.
tasks - Displays the tasks runnable from root project 'helloworldv2'.

Other tasks
-----------
helloTask
world
    hello
```

Getting detailed information about tasks

You can get detailed information about a specific task by using the **`help`** option combined with the **`--task`** switch. To get information about the task **`world`** of of the helloworldv2 example, execute the following command.

Code Listing 14

```
gradle help --task world
```

The output for this command should be the following.

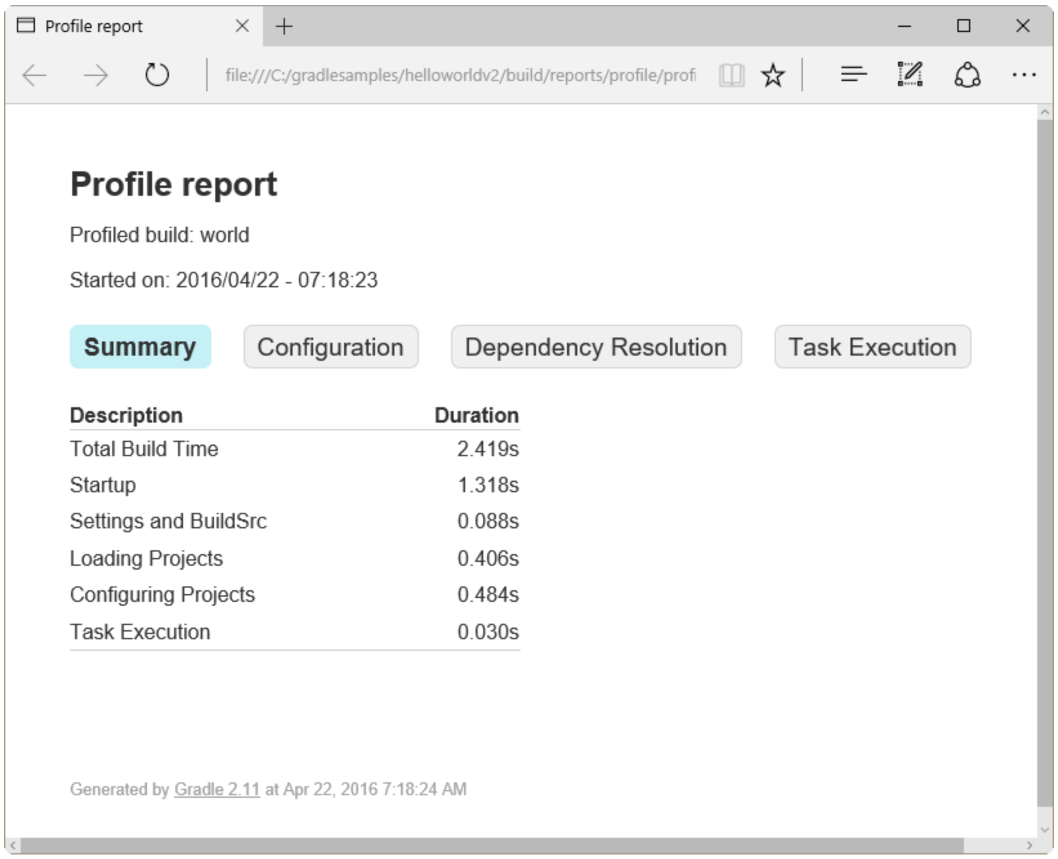
Code Listing 15

```
:help
Detailed task information for world

Path
     :world

Type
     Task (org.gradle.api.Task)

Description
     -

Group
     -

BUILD SUCCESSFUL

Total time: 2.863 secs
```

Getting a profile for the build script

The `--profile` command-line option allows you to get a report with information about the build. This report is written into the **build\reports\profile** folder. The report will be named using the date and time when the build script was run, and will be saved in HTML format.

This report lists summary times and detailed information for both the configuration phase and task execution. All operations are sorted, with the most expensive operation (in terms of consuming time) first. The results also indicate if any tasks were skipped, and the reason why this happened, and if tasks that were not skipped didn't work.



Profile report

Profiled build: world

Started on: 2016/04/22 - 07:18:23

Description	Duration
Total Build Time	2.419s
Startup	1.318s
Settings and BuildSrc	0.088s
Loading Projects	0.406s
Configuring Projects	0.484s
Task Execution	0.030s

Generated by Gradle 2.11 at Apr 22, 2016 7:18:24 AM

Figure 10: Profile report example for helloworldv2 build script

Chapter summary

This chapter has shown how to begin using Gradle. The traditional Hello World sample was used to understand the basic working concepts.

For the purposes of this book, all samples will be saved in a folder called **C:\gradlesamples**, and a subfolder for each example will be created within. For the Hello World sample, a subfolder named **helloworld** will be created, and a file named **build.gradle** will be created in the subfolder. This file is known as a build script, and it is the container for all Gradle build statements, which are Groovy language statements.

The first line of code for the **build.gradle** file starts with a **`task`** statement. **`Task`** is one of the two basic concepts on which Gradle sits.

A build script can contain several tasks, and a task can depend on another one to be executed. This can be accomplished by using the **`dependsOn`** clause in a **`task`** declaration, within parentheses. A declaration such as **`task world(dependsOn: hello)`** tells Gradle that a task named **`hello`** must be executed prior the task named **`world`**.

The **`gradle`** command line will be used throughout this book. There are options included that can be used to be productive right away. For example, the **`-h`** option is used to display all help text associated with command-line options, the **`tasks --all`** option is used to get a report of all tasks in the build script, the **`help --task`** option is used to get detailed information about a task, and the **`--profile`** option generates a report about build execution.

You don't have to type the full name of a task when executing it from the command-line—Gradle allows you to provide a portion of the task name, which uniquely identifies the task within the build script. This mechanism is known as abbreviation. It's recommended to name tasks in a way that avoids ambiguous references when using abbreviations, because Gradle will throw a warning during build if this happens.

Chapter 4 Build Script Basics

Project and task: the two basic concepts in Gradle

Everything in Gradle sits on top of two basic concepts: projects and tasks. A project is a collection of tasks, and each task performs some actions, such as compiling classes, or running unit tests. From now on, each folder containing the Gradle examples discussed in this book will be considered a project container.

Project lifecycle

A **build.gradle** file has a one-to-one relationship with a project. Every time a build script is executed, Gradle assembles a **`project`** object for each project that will participate in the build at build initialization. The **`project`** object is assembled in the following order:

1. Create a **`Settings`** instance for the build.
2. Evaluate the **`settings.gradle`** script, if present, against the **`Settings`** object to configure it.
3. Use the configured **`Settings`** object to create the hierarchy of **`Project`** instances.
4. Finally, each project is evaluated by executing its **build.gradle** file, if present, against the project. Every project in a build project is evaluated before its child projects.

Once these steps are complete, the build ends, and two folders named **build** and **.gradle** can be found as a result of the process.

Tasks

A task is considered the fundamental unit of build activity, and represents a basic piece of work that is performed by a build. This might be compiling some classes, creating a .jar file, generating Javadoc, or storing some archives to a repository.

This chapter will focus on a simple tasks definition for a one-project build. Multi-project builds and more about tasks will be discussed in later chapters.

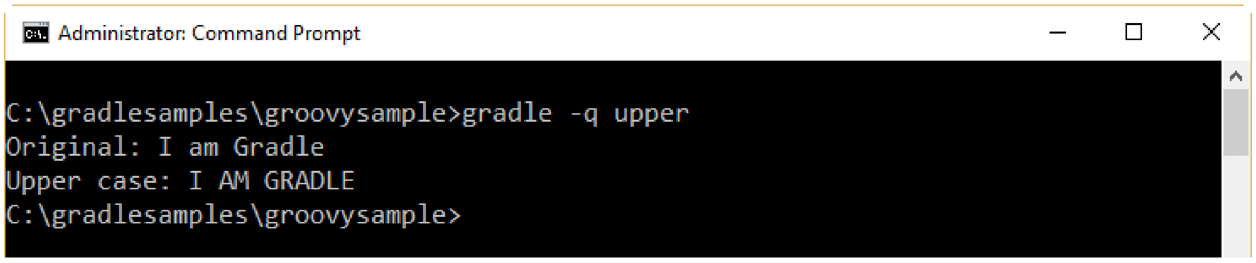
Build scripts are Groovy code

As mentioned in Chapter 1, Gradle’s build scripts are based on Groovy code, so the user can take advantage of the full power of Groovy. The following sample demonstrates that.

Code Listing 16

```
task upper << {
    String someString = 'I am Gradle'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
```

To execute this code, create a folder named **groovysample** in **C:\gradlesamples**. After that, save a **build.gradle** file with the code in the **groovysample** folder. The result of the build is shown in the following figure.



```
C:\gradlesamples\groovysample>gradle -q upper
Original: I am Gradle
Upper case: I AM GRADLE
C:\gradlesamples\groovysample>
```

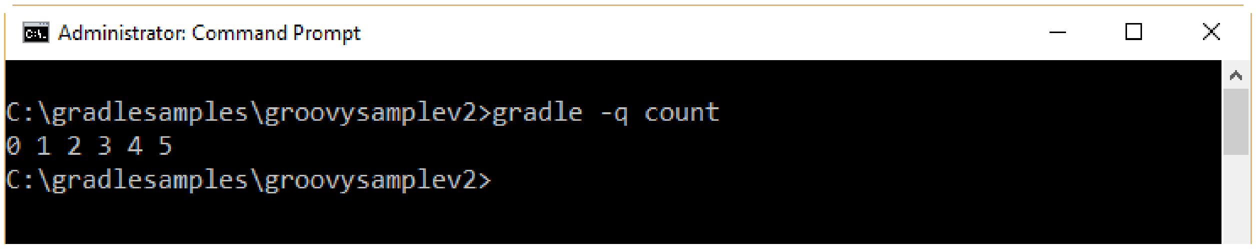
Figure 11: Groovy sample output

Another sample can use the Groovy **`times`** loop. The code for the **build.gradle** file should be created in a folder named **C:\gradlesamples\groovysamplev2**.

Code Listing 17

```
task count << {
  6.times { print "$it " }
  println ''
}
```

The result of the previous build is displayed in the following figure.



```
C:\gradlesamples\groovysamplev2>gradle -q count
0 1 2 3 4 5
C:\gradlesamples\groovysamplev2>
```

Figure 12: Groovy times loop sample output

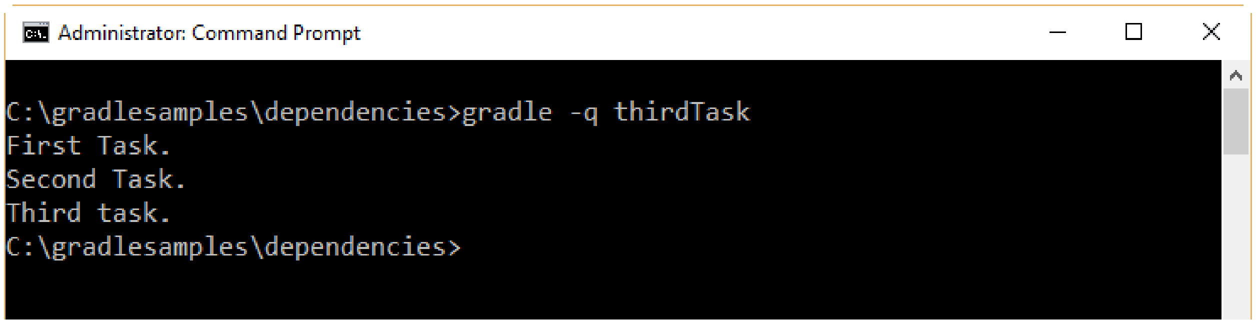
Task dependencies

Gradle allows you to declare tasks that depend on other tasks. This can be done by using the **`dependsOn:`** clause after the task name, and placing the name of the dependent task after the declaration. The following build script shows an example.

Code Listing 18

```
task firstTask << {
  println 'First Task.'
}

task secondTask(dependsOn: 'firstTask') << {
  println 'Second Task.'
}

task thirdTask(dependsOn: 'secondTask') << {
    println 'Third task.'
}
```

This example shows three tasks, two of which (**`secondTask`** and **`thirdTask`**) have dependencies declared. In this case, **`secondTask`** depends on **`firstTask`** to be executed before it, and **`thirdTask`** depends on **`secondTask`** to be executed first. The output for the build is displayed in the following figure.



Administrator: Command Prompt

```
C:\gradlesamples\dependencies>gradle -q thirdTask
First Task.
Second Task.
Third task.
C:\gradlesamples\dependencies>
```

Figure 13: Dependencies sample output

Dependencies can be declared with no strict order. Thus, the following code works in the same way as the example discussed at the beginning of this section.

Code Listing 19

```
task secondTask(dependsOn: 'firstTask') << {
  println 'Second Task.'
}

task thirdTask(dependsOn: 'secondTask') << {
    println 'Third task.'
}

task firstTask << {
  println 'First Task.'
}
```

Dynamic tasks

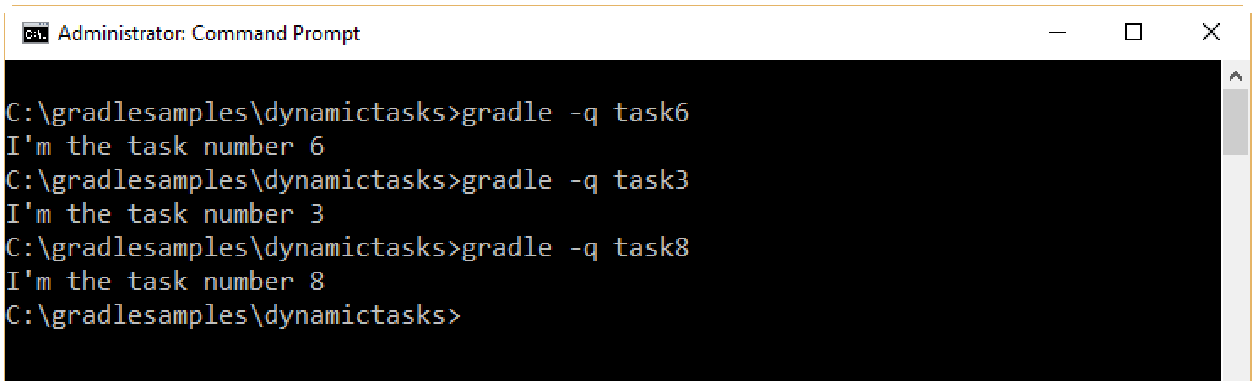
Due to the power of Groovy, Gradle allows you to create tasks dynamically. The following code illustrates this capability, using the Groovy **`times`** loop statement.

Code Listing 20

```
9.times { index ->
     task "task$index" << {
         println "I'm the task number $index"
     }
}
```

In this case, the **`times`** statement creates a loop that will iterate nine times. The index variable receives the iteration number (between 0 and 8) every time a loop iteration is executed, and a task is created within the loop. The name for each task is formed by the word **`task`** and the number stored in the index variable. So, the tasks created are **`task0`**, **`task1`**, **`task2`**, and so on.

Now, this build script can be executed by calling a task with a number between 0 and 8 (for example, **`task6`**). The output for the build script is presented in the following figure.



```
C:\gradlesamples\dynamictasks>gradle -q task6
I'm the task number 6
C:\gradlesamples\dynamictasks>gradle -q task3
I'm the task number 3
C:\gradlesamples\dynamictasks>gradle -q task8
I'm the task number 8
C:\gradlesamples\dynamictasks>
```

Figure 14: Dynamic tasks example output

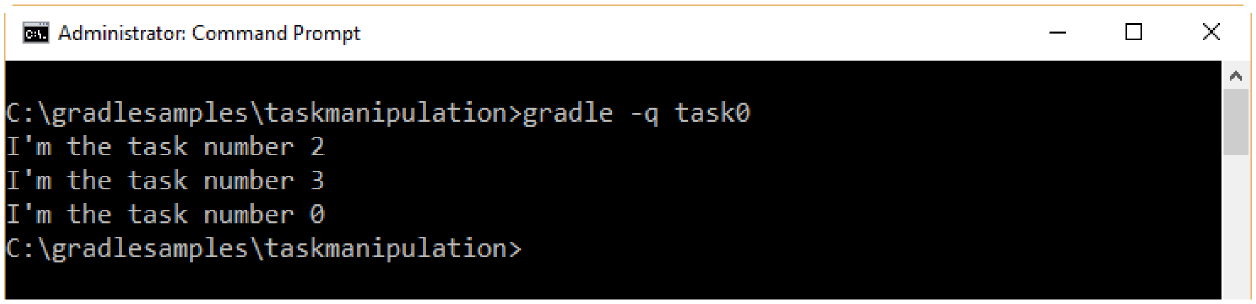
Task manipulation

Tasks can be accessed using an API after they're created. The following example adds dependencies to a task at runtime.

Code Listing 21

```
5.times { index ->
     task "task$index" << {
         println "I'm the task number $index"
     }
}
task0.dependsOn task2, task3
```

In the previous example, the dependencies for **`task0`** are declared using the **`dependsOn`** method, which is part of the **`task0`** object. Task objects will be discussed later in this book. The output for this code should look like this:



```
C:\gradlesamples\taskmanipulation>gradle -q task0
I'm the task number 2
I'm the task number 3
I'm the task number 0
C:\gradlesamples\taskmanipulation>
```

Figure 15: Task manipulation example output

Chapter summary

Everything in Gradle sits on top of two basic concepts: projects and tasks. A project is a collection of tasks, and each task performs some actions, such as compiling classes or running unit tests.

A **build.gradle** file has a one-to-one relationship with a project. Every time a build script is executed, Gradle assembles a **`Project`** object in the build

A task is considered the fundamental unit of build activity, and represents a basic piece of work that is performed by a build. This might be compiling some classes, creating a .jar file, generating Javadoc, or storing some archives to a repository.

Gradle allows you to declare tasks that depend on other tasks. This can be done by using the **`dependsOn:`** clause after the task name, and placing the name of the dependent task after the declaration. Dependencies can be declared with no strict order.

Since Gradle’s build scripts are based on Groovy code, you can take advantage of the full power of the Groovy language. This can be useful for creating tasks dynamically, or for task manipulation at runtime.

Chapter 5 Digging into Gradle Tasks

As explained in Chapter 4, a task is the fundamental unit of build activity, and it's identified within the build script with a name. Going a little further, a task is a named collection of build instructions that Gradle executes during a build process.

More about task declarations

All the samples discussed in earlier chapters of this book showed how to create a task and define its behavior at the same time. However, there's a simpler way to declare a task—all that's needed is a name. This way is displayed in the following sample.

Code Listing 22

```
task onlyname
```

If this task is executed with **`gradle -q onlyname`**, it will not produce any result. This is because this task hasn't had an action assigned yet.

Earlier in this book, a task was bound to an action using the left-shift (**`<<`**) operator and a set of statements within curly braces.

Code Listing 23

```
task nameandaction << {
   println 'An action was assigned to this task'
}
```

But Gradle also allows you to associate action to a task by referring to the task object created with the **`task`** declaration. This is displayed in the following sample.

Code Listing 24

```
task nameandaction

nameandaction << {
   println 'An action was assigned to this task'
}

nameandaction << {
   println 'A second action was assigned to this task'
}
```

The output for this example should look like this.

Code Listing 25

```
An action was assigned to this task
```

```
A second action was assigned to this task
```

Even though the previous example reflects a trivial build behavior, it exposes a powerful insight: tasks aren't one-off declarations of build activity, but first-class objects in Gradle programming.

Since build actions can be assigned to tasks over the course of the build file, there's more you can do with tasks. This chapter will go a little further in exploring the capabilities of tasks.

Note: In Groovy, operators like << (left-shift operator from Java) can be overloaded to have different meanings, depending on the types of the objects they operate on. Gradle has overloaded the << operator to append a code block to the list of actions a task performs. The equivalent of this is the doLast() method, which will be covered later in this book.

Configuration blocks

Considering the example discussed in the previous section, take a look at the following code:

Code Listing 26

```
task extractDatabase

extractDatabase << {
   println 'Connect to database'
}

extractDatabase << {
   println 'Extracting database data'
}

extractDatabase << {
   println 'Closing database connection'
}

extractDatabase {
   println 'Setting up database connection'
}
```

It is expected that all code blocks were build actions snippets. So, you might expect that the message for the fourth block will be printed at the end. But instead, the following output is displayed.

Code Listing 27

```
Setting up database connection
Connect to database
Extracting database data
Closing database connection
```

Why is that?

In Gradle, when a code block with no left-shift operator is added to a task name, this code block is treated as a configuration block. This means that the code block will be run during Gradle's configuration phase, according to the project lifecycle detailed in Chapter 4. This phase runs before the execution phase, when task actions are executed.

A configuration block can be additive just like action blocks, so the following code will work exactly as the example shown at the beginning of this section.

Code Listing 28

```
task extractDatabase

extractDatabase << {
   println 'Connect to database'
}

extractDatabase << {
   println 'Extracting database data'
}

extractDatabase << {
   println 'Closing database connection'
}

extractDatabase {
   print 'Setting up '
}

extractDatabase {
   println 'database connection'
}
```

A configuration block is useful to set up variables and data structures that will be needed by the task action when it runs later on in the build. This gives the user an opportunity to turn a build's tasks into a rich object model populated with information about the build.

All build configuration code runs every time Gradle executes a build script, regardless of whether any given task runs during execution.

Note: As discussed in Chapter 4, every time Gradle executes a build, it runs through three lifecycle phases: initialization, configuration, and execution. Build tasks are executed in the execution phase, following the order required by their dependency relationships. Those task objects are assembled into an internal object model during the configuration phase. This internal object model is usually called the

DAG (Directed Acyclic Graph). During initialization phase, Gradle decides which projects are going to participate in the build.

Task object model basics

Every time Gradle executes a build, it creates an internal object model beforehand, so every task declared is a **`task`** object contained within the overall project. Like any object, a **`task`** object has properties and methods.

The default type for each new task is **`DefaultTask`**. Every Gradle task descends from this object type. The **`DefaultTask`** contains the functionality required for them to interface with the Gradle project model.

An overview of **`task`**’s properties and methods will be discussed in the following sections.

Methods of DefaultTask

dependsOn(task)

This method adds a task as a dependency of the calling task. This depended-on task will always run before the task that depends on it. The following sample shows several ways to use this technique.

Code Listing 29

```
task setUpConnection {
  println 'Setting up database connection'
}

//Declares dependency using quotes (usually optional)
task connectToDatabase {
  dependsOn 'setUpConnection'
}

//Declares dependency using the task name with no quotes
task extractDatabase {
 dependsOn connectToDatabase
}

//Declares dependency using the left-shift operator
task closeConnection {
  dependsOn << extractDatabase
}
```

Another way to write the previous code with the same results is:

Code Listing 30

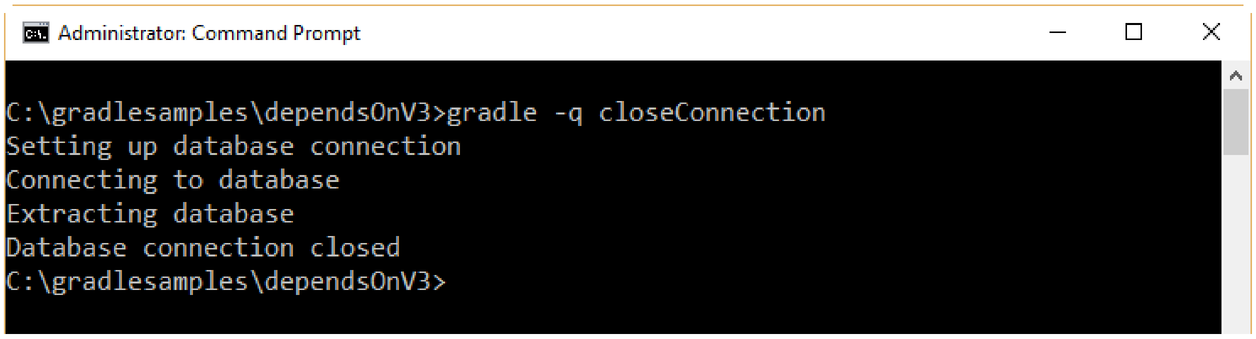
```
task setUpConnection {
  println 'Setting up database connection'
}

//An explicit call on the task object
task connectToDatabase
connectToDatabase.dependsOn setUpConnection

//Declares dependency using the task name with no quotes
task extractDatabase {
 dependsOn connectToDatabase
}

//A shortcut for declaring dependencies (discussed in Chapter 4)
task closeConnection(dependsOn: extractDatabase)
```

Multiple dependencies can be used. In this way, a task can depend on more than one task. The following sample shows how to achieve this.

Code Listing 31

```
task setUpConnection {
  println 'Setting up database connection'
}

task connectToDatabase
connectToDatabase
{
    println 'Connecting to database'
}

//Declares multi-dependency by listing tasks in a comma separated list
task extractDatabase {
 dependsOn setUpConnection, connectToDatabase
 println 'Extracting database'
}

//A shortcut for declaring dependencies (discussed in Chapter 4)
task closeConnection(dependsOn: extractDatabase) {
    println 'Database connection closed'
}
```

All depended-on tasks are executed following the order in which they appear in the list, starting from the left. The following figure shows the output for the previous code.



```
C:\gradlesamples\dependsOnV3>gradle -q closeConnection
Setting up database connection
Connecting to database
Extracting database
Database connection closed
C:\gradlesamples\dependsOnV3>
```

Figure 16: Displayed output when closeConnection task is executed.

doFirst(closure)

This method adds a block of executable code to the beginning of a task's action.

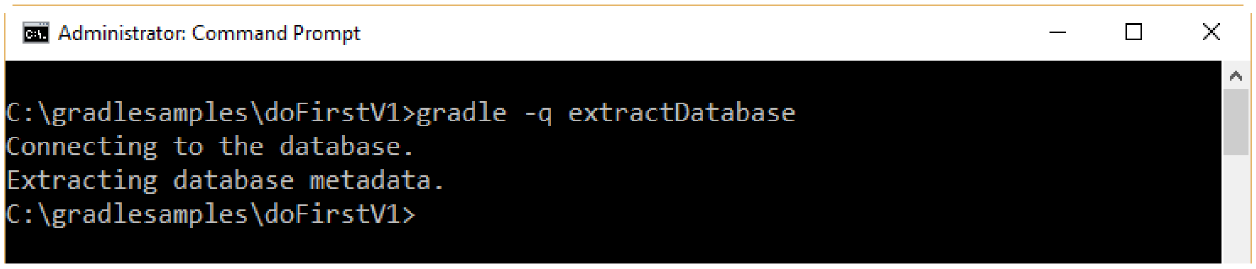
Notice that the term **`closure`** appears within the parentheses. This term is inherited from Groovy, and it refers to a block of code between two curly braces. A closure can function like an object, and it can be passed as a parameter to a method or assigned to a variable, and then executed later.

The **`doFirst`** method can be invoked directly on the **`task`** object, passing a closure to the method. This closure contains the code to run before the **`task`** action.

Code Listing 32

```
task extractDatabase << {
    //This is the task action
    println 'Extracting database metadata.'
}

extractDatabase.doFirst {
    println 'Connecting to the database.'
}
```

The output for the previous example should look like this.



```
C:\gradlesamples\doFirstV1>gradle -q extractDatabase
Connecting to the database.
Extracting database metadata.
C:\gradlesamples\doFirstV1>
```

Figure 17: The doFirst execution sample output

The user can invoke **`doFirst`** from within the task's configuration block. As discussed earlier in this chapter, all code located in those configuration blocks runs before any task action occurs, during the configuration phase of the build.

Code Listing 33

```
task extractDatabase << {
    //This is the task action
    println 'Extracting database metadata.'
}

extractDatabase{
    doFirst {
    println 'Connecting to the database.'
    }
}
```

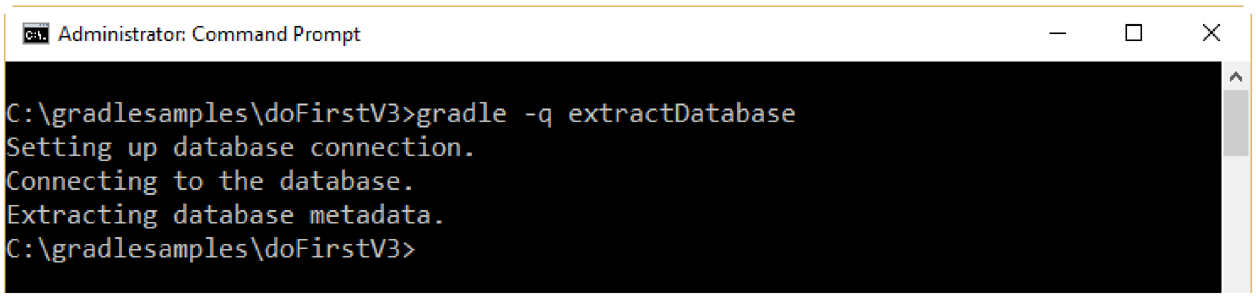
In this example, the **`doFirst`** method is invoked within the **`extractDatabase`** configuration block. So when Gradle executes the build script, the code associated to the **`doFirst`** method is executed in the configuration phase, prior to any task action.

Repeated calls to the **`doFirst`** method can be made. These calls are additive, meaning that each previous call is retained, and the new closure is appended to the start of the list. As a result, all calls to the **`doFirst`** method will be executed in reverse order (starting from the last call and ending with the first one).

Code Listing 34

```
task extractDatabase << {
    //This is the task action
    println 'Extracting database metadata.'
}

extractDatabase{
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}
```

The output for the previous code is displayed in the following figure.


Administrator: Command Prompt

```
C:\gradlesamples\doFirstV3>gradle -q extractDatabase
Setting up database connection.
Connecting to the database.
Extracting database metadata.
C:\gradlesamples\doFirstV3>
```

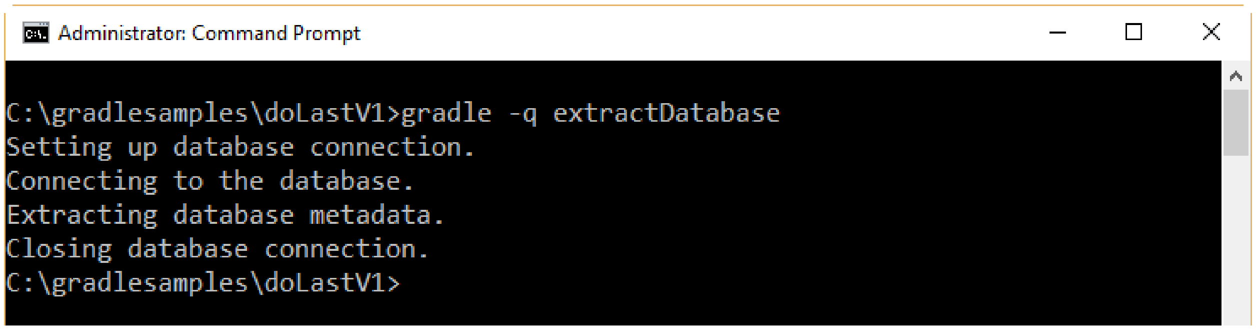
Figure 18: Sample output for doFirst repeated calls

doLast(closure)

This method is similar to the **doFirst** method, except that the closure is appended to the end of an action. This is useful when it's needed to execute a code block after all of a task's activities.

Code Listing 35

```
task extractDatabase << {
    //This is the task action
    println 'Extracting database metadata.'
}

extractDatabase.doLast {
   println 'Closing database connection.'
}

extractDatabase{
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}
```



```
C:\gradlesamples\doLastV1>gradle -q extractDatabase
Setting up database connection.
Connecting to the database.
Extracting database metadata.
Closing database connection.
C:\gradlesamples\doLastV1>
```

Figure 19: The output for the doLast example

The output displayed in Figure 19 shows that the code assigned to the **doLast** method of the **extractDatabase** task is executed after all of the task's activities. In this case, it is supposed that a database connection must be closed after extracting all data from the database itself.

The **doLast** method also is additive. In this way, all additive calls append their closure to the end of the execution's list, so the calls will be executed starting from the first one, and ending with the last closure added.

Code Listing 36

```
task extractDatabase << {
    //This is the task action
    println 'Extracting database metadata.'
}
```

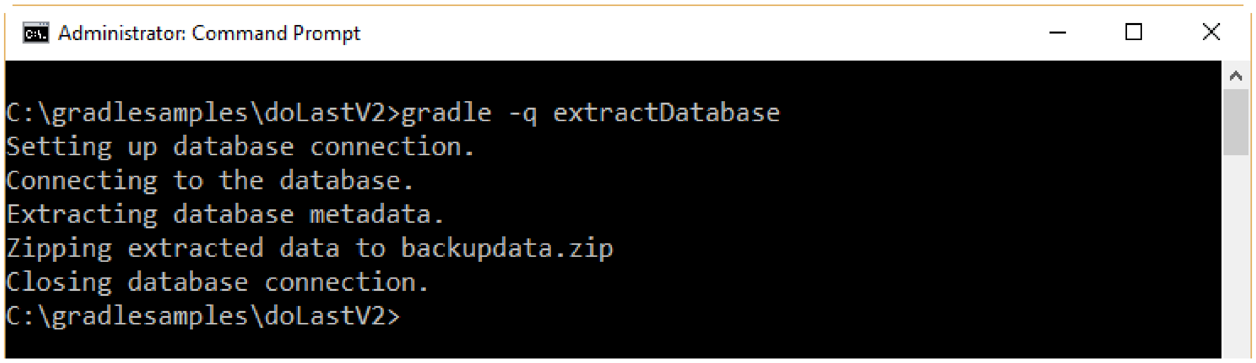
```
extractDatabase.doLast {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase.doLast {
   println 'Closing database connection.'
}

extractDatabase{
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}
```

As explained in the “Configuration blocks” section of this chapter, another way to express a call to the **`doLast`** method is by using the left-shift (**`<<`**) operator.

Code Listing 37

```
task extractDatabase << {
    //This is the task action
    println 'Extracting database metadata.'
}

extractDatabase << {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase << {
   println 'Closing database connection.'
}

extractDatabase{
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}
```

The output for both previous examples should look like this.



```
C:\gradlesamples\doLastV2>gradle -q extractDatabase
Setting up database connection.
Connecting to the database.
Extracting database metadata.
Zipping extracted data to backupdata.zip
Closing database connection.
C:\gradlesamples\doLastV2>
```

Figure 20: Repeated calls to the doLast method

onlyIf (closure)

This method allows you to express a predicate that determines whether a task should be executed. The predicate corresponds to a closure with a returning value, which is the value that will be assigned to the predicate. Gradle uses a switch logic (on or off) to execute the task, or to ignore it. So, the value assigned to the predicate should be a Boolean value.

Code Listing 38

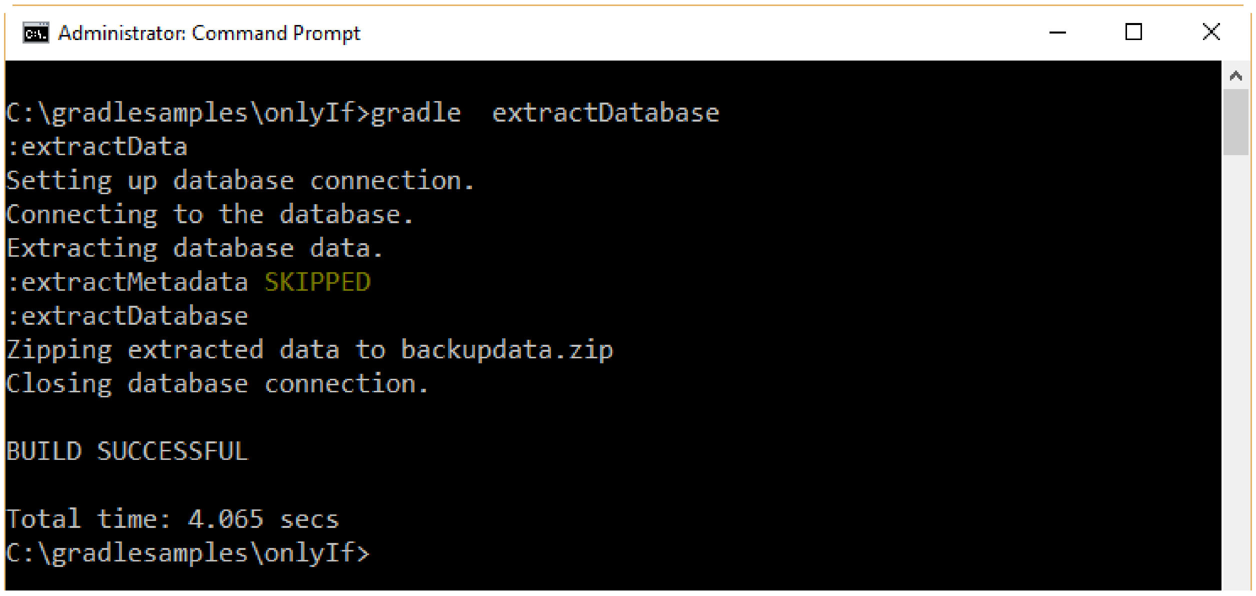
```
/* This is the task that receives the predicate. */
task extractMetadata << {
        println 'Extacting metadata.'
}

/* The predicate is assigned to extractMetadata. */
/* The task will be executed if the property extract.metadata is equal to
true. */
extractMetadata.onlyIf {
    System.properties['extract.metadata'] == 'true'
}

task extractData << {
    println 'Extracting database data.'
}

/* This is the main task. */
task extractDatabase << {
    //This is the task action
}

/* Code that will be executed after extractDatabase action ends. */
extractDatabase << {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase << {
   println 'Closing database connection.'
}

/* extractDatabase will depend on extractData and extractMetadata. */
/* extractData is executed first, then extractMetadata will be executed
depending on the predicate. */
extractDatabase {
    dependsOn extractData, extractMetadata
}

/* Code that will be executed before extractData task action begins is set
up at configuration phase. */
extractData {
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}
```

The previous example can be called in two ways. For the purposes of the **`onlyIf`** property explanation, the **`-q`** (quiet mode) option will be omitted in both calls. The first one looks like this:

Code Listing 39T

```
gradle extractDatabase
```

The following figure shows the output displayed.



```
C:\gradlesamples\onlyIf>gradle  extractDatabase
:extractData
Setting up database connection.
Connecting to the database.
Extracting database data.
:extractMetadata SKIPPED
:extractDatabase
Zipping extracted data to backupdata.zip
Closing database connection.

BUILD SUCCESSFUL

Total time: 4.065 secs
C:\gradlesamples\onlyIf>
```

Figure 21: A call to extractDatabase without a value for extract.metadata property

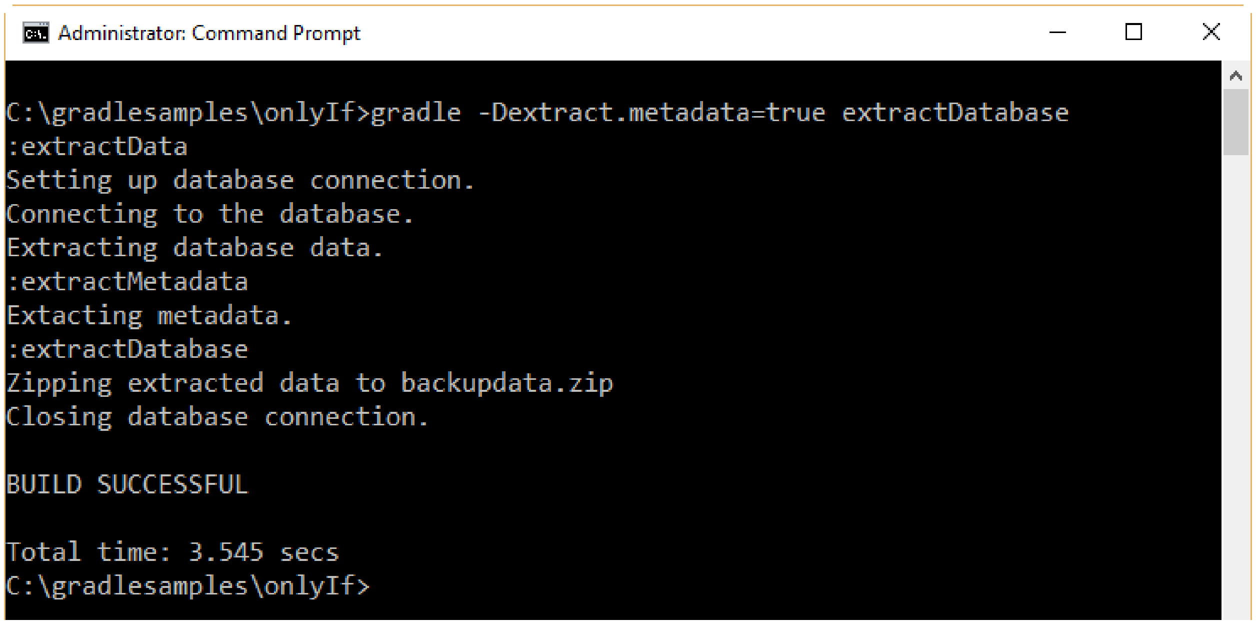
Notice that the **extractMetadata** task was skipped for this call. The reason for this relies on the absence of a value for the **extract.metadata** property defined in the predicate closure. Gradle assumes that the property has a value of **false**, and omits the execution of the **extractMetadata** task.

To assign a value to the **extract.metadata** property, the **-D** option of Gradle's command line should be used. The call to **extractDatabase** task should look like the following sample.

Code Listing 40

```
gradle -Dextract.metadata=true extractDatabase
```

The output for this call is shown in the following figure.



```
C:\gradlesamples\onlyIf>gradle -Dextract.metadata=true extractDatabase
:extractData
Setting up database connection.
Connecting to the database.
Extracting database data.
:extractMetadata
Extacting metadata.
:extractDatabase
Zipping extracted data to backupdata.zip
Closing database connection.

BUILD SUCCESSFUL

Total time: 3.545 secs
C:\gradlesamples\onlyIf>
```

Figure 22: The result of calling extractDatabase with a given value of true for extract.metadata

The predicate for the **onlyIf** method can use any kind of code in order to return a **true** or a **false** value. So, this is not limited to System properties to make the test—you can read files, call a web service, check credentials, or anything else.

Properties of DefaultTask

didWork

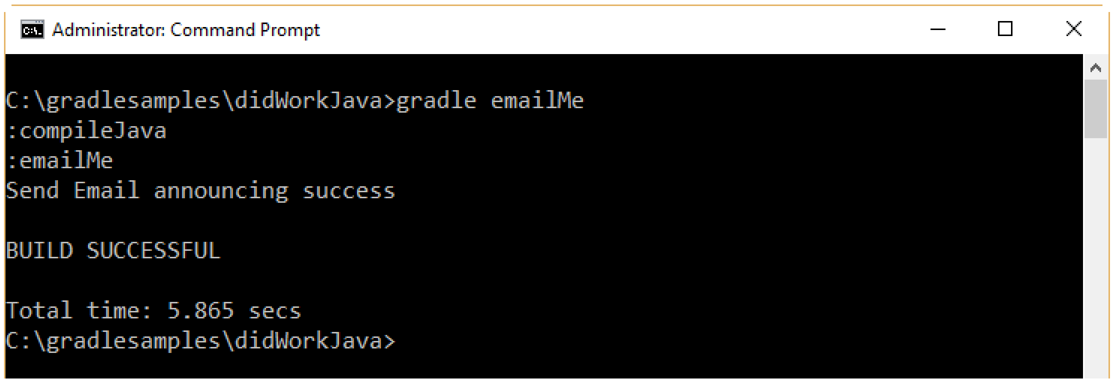
This Boolean property indicates whether a task was completed successfully. You can set the **didWork** property in its own task actions, in order to reflect the results of build code.

In the following example, an implementation of the Java Compiler will return a **didWork** value of **true** if at least one file was successfully compiled.

Code Listing 41

```
/* Applying the Java plugin for this Gradle build. */
apply plugin: 'java'

/* Making emailMe task depend on Java compilation.
 The task compileJava will return true for didWork
 property if at least one file was successfully
 compiled. */
task emailMe(dependsOn: compileJava) << {
    if (tasks.compileJava.didWork) {
        println 'Send Email announcing success'
    }

}
```

The output for this build should look like this.



```
C:\gradlesamples\didWorkJava>gradle emailMe
:compileJava
:emailMe
Send Email announcing success

BUILD SUCCESSFUL

Total time: 5.865 secs
C:\gradlesamples\didWorkJava>
```

Figure 23: Using didWork property.

enabled

This Boolean property indicates whether the task will be executed. You can set any task's **`enabled`** property value to **`false`**, which will prevent Gradle from executing the task. If there are any dependencies, they're still going to execute as if the task were enabled.

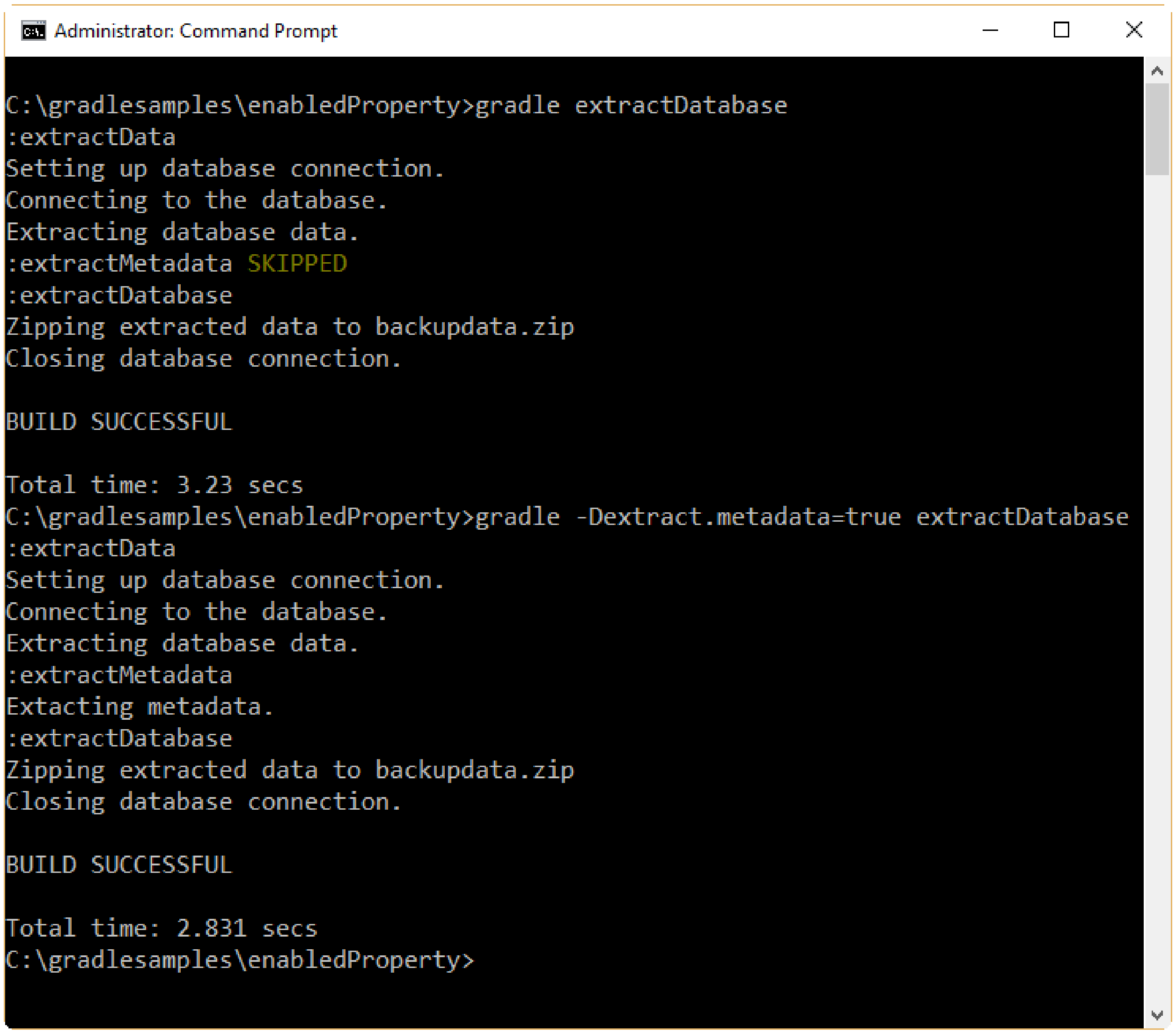
The example for the **`onlyIf`** method, explained in the "Methods" section of this chapter, will be taken to demonstrate the enabled property. This example will be modified with a few changes.

Code Listing 42

```
/* This is the task that will be enabled
   on demand. */
task extractMetadata << {
        println 'Extacting metadata.'
}

task extractData << {
    println 'Extracting database data.'
}
```

```
/* Code that will be executed before extractData task action begins, is set
up at configuration phase. */
extractData {
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}

/* This is the main task. */
task extractDatabase (dependsOn: [extractMetadata, extractData]) << {
    //This is the task action
}

/* A closure is assigned to the extractDatabase task
   at configuration phase, so the code will be executed
   before Gradle enters to the execution phase.

   The tasks collection is used to access the extractMetadata
   task object, in order to set the value for enabled property.

   The value for enabled property depends on the extract.metadata
   System property. If this propety is set to true, the extractMetadata
   task will be executed.
*/
extractDatabase {
    tasks['extractMetadata'].enabled =
(System.properties['extract.metadata'] == 'true')
}

/* Code that will be executed after extractDatabase action ends. */
extractDatabase << {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase << {
   println 'Closing database connection.'
}
```

Like the **`onlyIf`** example, this build script can be called in two ways—one of them assigning the value of **`true`** to the **`extractMetadata`** property. The output for both ways is displayed in the following figure.


Administrator: Command Prompt

```
C:\gradlesamples\enabledProperty>gradle extractDatabase
:extractData
Setting up database connection.
Connecting to the database.
Extracting database data.
:extractMetadata SKIPPED
:extractDatabase
Zipping extracted data to backupdata.zip
Closing database connection.

BUILD SUCCESSFUL

Total time: 3.23 secs
C:\gradlesamples\enabledProperty>gradle -Dextract.metadata=true extractDatabase
:extractData
Setting up database connection.
Connecting to the database.
Extracting database data.
:extractMetadata
Extacting metadata.
:extractDatabase
Zipping extracted data to backupdata.zip
Closing database connection.

BUILD SUCCESSFUL

Total time: 2.831 secs
C:\gradlesamples\enabledProperty>
```

Figure 24: Results for enabled property example

path

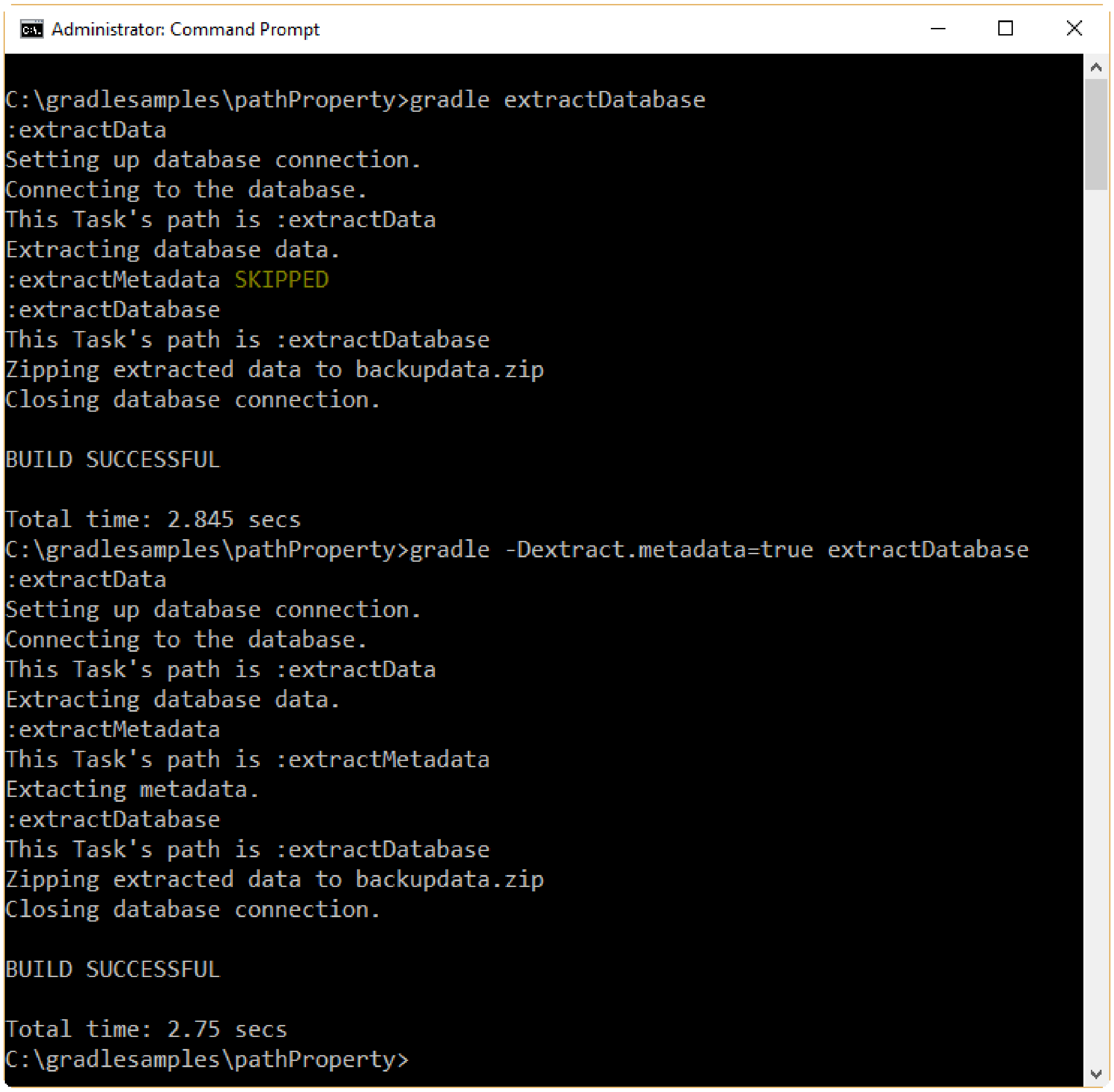
This string property contains the fully qualified path of the task object. A task's path is, by default, simply the name of the task with a leading colon. The leading colon indicates that the task is located in the top-level build script file. Since Gradle supports dependent subprojects or nested builds, if the task existed in a nested build, then the path would be **`:nestedBuild:taskName`**.

The enabled property example was modified to display all tasks' paths along the build execution.

Code Listing 43

```
/* This is the task that will be enabled
   on demand. */
task extractMetadata << {
        println "This Task's path is ${path}"  //Displaying task's path
        println 'Extacting metadata.'
}

task extractData << {
    println "This Task's path is ${path}" //Displaying task's path
    println 'Extracting database data.'
}

/* Code that will be executed before extractData task action begins, is set
up at configuration phase. */
extractData {
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}

/* This is the main task. */
task extractDatabase (dependsOn: [extractMetadata, extractData]) << {
    println "This Task's path is ${path}" //Displaying task's path
    //This is the task action
}

/* A closure is assigned to the extractDatabase task
   at configuration phase, so the code will be executed
   before Gradle enters to the execution phase.

   The tasks collection is used to access the extractMetadata
   task object, in order to set the value for enabled property.

   The value for enabled property depends on the extract.metadata
   System property. If this propety is set to true, the extractMetadata
   task will be executed.
*/
extractDatabase {
    tasks['extractMetadata'].enabled =
(System.properties['extract.metadata'] == 'true')
}

/* Code that will be executed after extractDatabase action ends. */
extractDatabase << {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase << {
   println 'Closing database connection.'
}
```

The output should look like this:



```
C:\gradlesamples\pathProperty>gradle extractDatabase
:extractData
Setting up database connection.
Connecting to the database.
This Task's path is :extractData
Extracting database data.
:extractMetadata SKIPPED
:extractDatabase
This Task's path is :extractDatabase
Zipping extracted data to backupdata.zip
Closing database connection.

BUILD SUCCESSFUL

Total time: 2.845 secs
C:\gradlesamples\pathProperty>gradle -Dextract.metadata=true extractDatabase
:extractData
Setting up database connection.
Connecting to the database.
This Task's path is :extractData
Extracting database data.
:extractMetadata
This Task's path is :extractMetadata
Extacting metadata.
:extractDatabase
This Task's path is :extractDatabase
Zipping extracted data to backupdata.zip
Closing database connection.

BUILD SUCCESSFUL

Total time: 2.75 secs
C:\gradlesamples\pathProperty>
```

Figure 25: Displaying the task's path

description

This string property is a small piece of human-readable metadata, used to document the purpose of a task. The following sample shows some ways to set a description.

Code Listing 44

```
task extractMetadata(description: 'Extracts database metadata') << {
        println "This Task's path is ${path}"  //Displaying task's path
        println 'Extacting metadata.'
```

```
}

task extractData << {
    println "This Task's path is ${path}" //Displaying task's path
    println 'Extracting database data.'
}

/* Code that will be executed before extractData task action begins is set
up at configuration phase. */
extractData {
    description = 'Extracts data from the database'
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}

/* This is the main task. */
task extractDatabase (dependsOn: [extractMetadata, extractData]) << {
    println "This Task's path is ${path}" //Displaying task's path
    //This is the task action
}

extractDatabase.description = 'Entry point for extractDatabase build'

/* A closure is assigned to the extractDatabase task
   at configuration phase, so the code will be executed
   before Gradle enters the execution phase.

   The tasks collection is used to access the extractMetadata
   task object, in order to set the value for enabled property.

   The value for enabled property depends on the extract.metadata
   System property. If this property is set to true, the extractMetadata
   task will be executed.
*/
extractDatabase {
    tasks['extractMetadata'].enabled =
(System.properties['extract.metadata'] == 'true')
}

/* Code that will be executed after extractDatabase action ends. */
extractDatabase << {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase << {
   println 'Closing database connection.'
```

```
}
```

Since the description is used for documentation purposes only, the output is the same as was shown in the section on the **path** property, previously discussed in this chapter.

About task types

According to the “Task object model basics” section of this chapter, every task has a type. The first type for a task is the **DefaultTask**, but there are also task types for copying, archiving, executing programs, and many more.

A complete task reference is beyond the scope of this book, but a few important types will be discussed in this section.

Copy

A **copy** task copies files from one place into another. Its most basic form copies files from one directory to another. There are optional restrictions for performing this action, like file patterns to be included or to be excluded.

Code Listing 45

```
task copyFiles(type: Copy){
    from 'source'
    into 'target'
    include '**/*.txt'
}

task backupFiles(dependsOn: copyFiles) << {
    println 'Backup completed'
}
```

This example creates a copy of all text files located in a directory named **source**, and places that copy in a directory named **target**. If the destination directory (**target**, in this case) doesn’t exist, the **copy** task will create that directory.

This example should be executed with the following command.

Code Listing 46

```
gradle backupFiles
```

The **from**, **into**, and **include** methods are inherited from **Copy**.

Zip

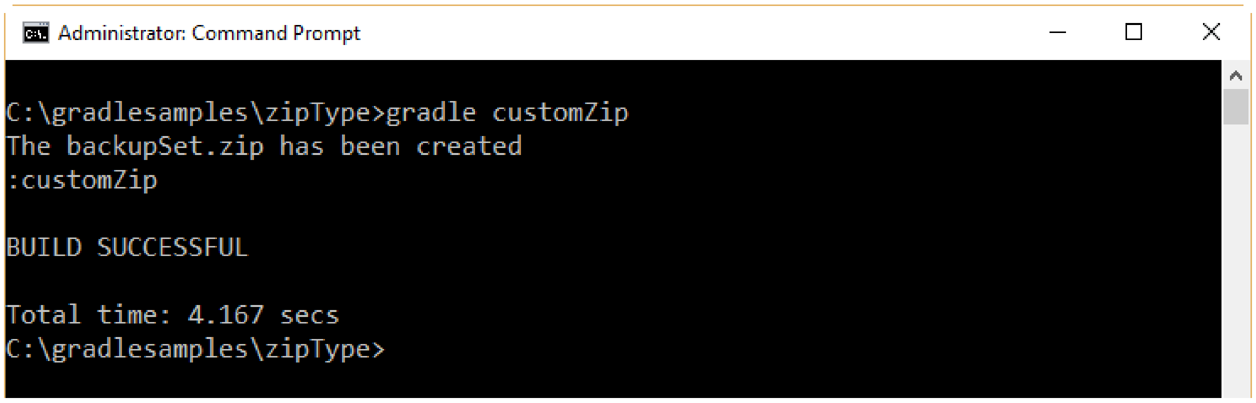
A **zip** task creates a compressed file in .zip format. The **from** method can be used to specify the folder where the files to be compressed are saved.

Code Listing 47

```
task customZip(type: Zip){
    from 'source'
    baseName = 'backupSet'
}
/* Using Groovy string interpolation to display zip file name. */
println "The ${customZip.archiveName} has been created"
```

This code creates a .zip file from the contents of source folder, which is within the folder where **build.gradle** is located. The name for the .zip file will be **backupSet.zip**, and will be placed in the same folder as **build.gradle**.

Before **`build`** execution ends, the .zip filename will be displayed in a message indicating that the file was created.



```
C:\gradlesamples\zipType>gradle customZip
The backupSet.zip has been created
:customZip

BUILD SUCCESSFUL

Total time: 4.167 secs
C:\gradlesamples\zipType>
```

Figure 26: The output for customZip build

Assigning a version number to the .zip file

Sometimes it’s useful to assign a version number to a .zip file, especially if the content of the file corresponds to an application’s source code, which is continuously modified.

To accomplish this task, a **`System`** property can be used in conjunction with the version property of the **`Zip`** type.

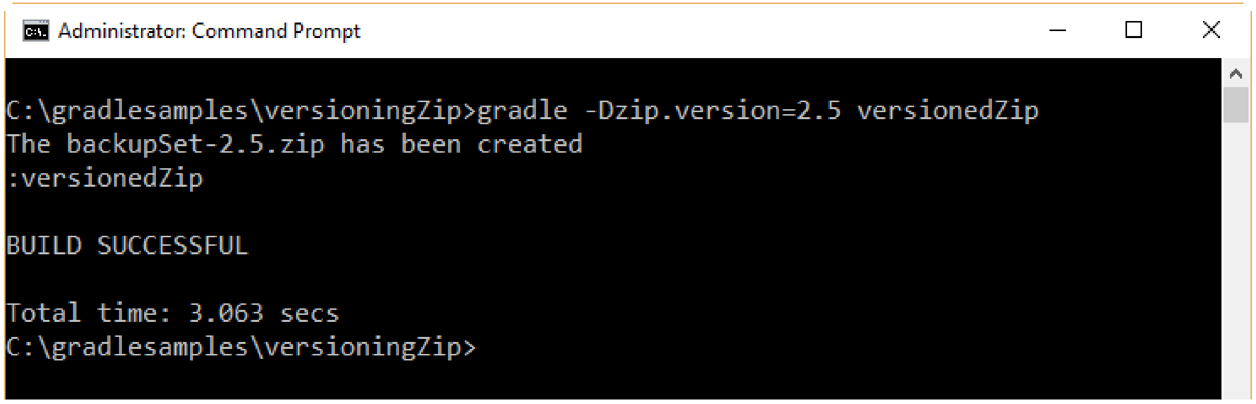
Code Listing 48

```
task versionedZip(type: Zip){
    from 'source'
    baseName = 'backupSet'
    version = System.properties['zip.version']
}
/* Using Groovy string interpolation to display zip file name. */
println "The ${versionedZip.archiveName} has been created"
```

The build should be executed with the following command.

Code Listing 49

```
gradle -Dzip.version=2.5 versionedZip
```



```
C:\gradlesamples\versioningZip>gradle -Dzip.version=2.5 versionedZip
The backupSet-2.5.zip has been created
:versionedZip

BUILD SUCCESSFUL

Total time: 3.063 secs
C:\gradlesamples\versioningZip>
```

Figure 27: The zip file with a version number

Note: The expressions surrounded with the ${} placeholder in the double-quoted strings are part of the string interpolation action. This action evaluates the expression within the placeholder, and then transforms the result to its string representation upon evaluation of the entire string.

Chapter summary

As explained in Chapter 4, a task is the fundamental unit of build activity, and it's identified with a name within the build script. This name is all the user needs to declare a task, but won't produce any result because no action was assigned to the task itself.

The left-shift (`<<`) operator is used to bind a task to an action. An action consists of a set of statements within curly braces. Gradle also allows you to combine action code in the task by referring to the task object created with the task declaration. So, tasks are objects in Gradle programming.

When a code block is added to a task name with no left-shift operator (**`<<`**), this code block is treated as a configuration block. The code block will be run during Gradle's configuration phase. This phase runs before the execution phase, when tasks' actions are executed.

Every time Gradle executes a build, it creates an internal object model beforehand. Every task declared is a task object with properties and methods. The default type for each new task is **`DefaultTask`**, from which every Gradle task inherits. The control methods **`dependsOn()`**, **`doFirst()`**, **`doLast()`**, and **`onlyIf()`** were explained in this chapter, as well as the properties **`didWork`**, **`enabled`**, **`path`**, and **`description`**.

Besides the **`DefaultTask`** type, there are other special task types: the **`Copy`** and **`Zip`** types.

Finally, the term *closure* appeared in this chapter. This term is inherited from Groovy, and it refers to a block of code between two curly braces. You can think of a closure as an anonymous function. This block can work like an object, so it can be used as a parameter, or can be assigned to a variable to be executed later.

Chapter 6 More on Writing Build Scripts

Using a different build script file

So far, every build script has been saved in a file named **build.gradle**, and each time Gradle was executed, it looked for this file by default.

A build script can be saved in a file with different name, however. For the next example, **another.bld** will be used as the filename.

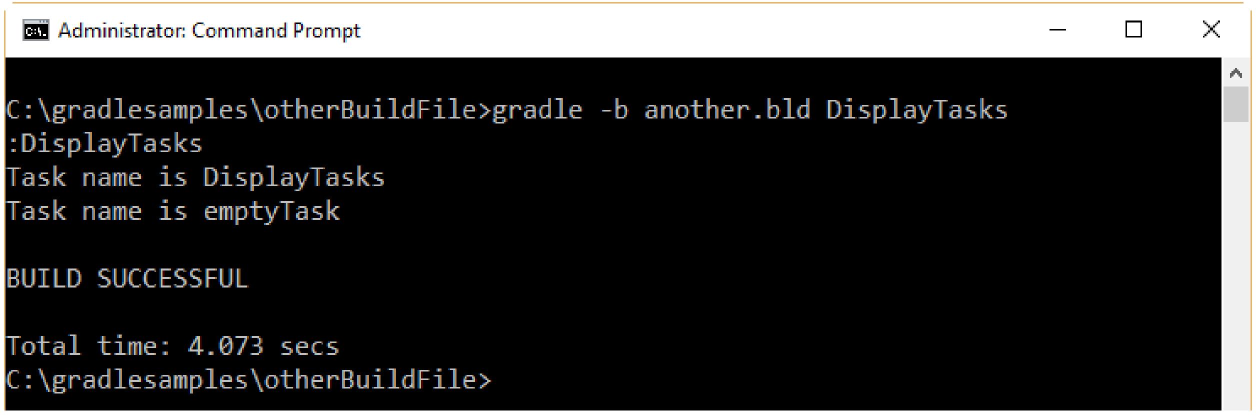
Code Listing 50

```
task emptyTask << {
}

task DisplayTasks << {
   tasks.each {
       println "Task name is ${it.name}"
   }
}
```

This example displays the names of all tasks that are present in the build. The following command is used to execute the build script.

Code Listing 51

```
gradle -b another.bld DisplayTasks
```



```
C:\gradlesamples\otherBuildFile>gradle -b another.bld DisplayTasks
:DisplayTasks
Task name is DisplayTasks
Task name is emptyTask

BUILD SUCCESSFUL

Total time: 4.073 secs
C:\gradlesamples\otherBuildFile>
```

Figure 28: Output for DisplayTasks example

Specifying a default task

Build scripts allow you to specify a series of default tasks, which will be executed if no task is specified in the command line (when the user calls **`gradle`**).

Code Listing 52

```
defaultTasks 'DisplayTasks'

task emptyTask << {
}

task DisplayTasks << {
   tasks.each {
       println "Task name is ${it.name}"
   }
}
```

The previous code defines **`DisplayTasks`** as the default task. Assuming that the build script is saved in the **build.gradle** file, if the user executes **`gradle`** in the following way, **`DisplayTasks`** will be executed automatically, because no task name was specified in the command line.

Code Listing 53

```
gradle
```

Another way **`defaultTasks`** can be used is displayed in the following sample.

Code Listing 54

```
/* A list of default tasks is supplied. Each task will be executed
   in order, starting from the left.
*/
defaultTasks 'SetEnvironment','extractDatabase','ResetEnvironment'

task SetEnvironment << {
    println 'Setting operating system environment'
}

task extractMetadata << {
        println 'Extacting metadata.'
}

task extractData << {
    println 'Extracting database data.'
}

task extractDatabase << {

}

task ResetEnvironment << {
    println 'Resetting operating system environment'
}

/* Code that will be executed after extractDatabase action ends. */
extractDatabase << {
    println 'Zipping extracted data to backupdata.zip'
}

extractDatabase << {
   println 'Closing database connection.'
}

/* extractDatabase will depend on extractData and extractMetadata.       */
/* extractData is executed first, then extractMetadata will be executed. */
extractDatabase {
    dependsOn extractData, extractMetadata
}

/* Code that will be executed before extractData task action begins is set
up at configuration phase. */
extractData {
    doFirst {
    println 'Connecting to the database.'
    }
    doFirst {
        println 'Setting up database connection.'
    }
}
```

The previous code sets three tasks as default tasks. It will be assumed that the code is saved in a file named **severaldefaults.gradle**, so when this build script is called with the following command, **`SetEnvironment`**, **`extractDatabase`**, and **`ResetEnvironment`** will be executed, in that order.

Code Listing 55

```
gradle -b severaldefaults.gradle
```

Taking a Java quick start

At this point, it has been stated that Gradle is a general-purpose build tool. It can build anything you want to implement in a build script. But, simply out-of-the-box, Gradle doesn't build anything unless you add code to the build script to do so.

This section is intended to explain how to build a Java program. Most Java projects are pretty similar: a set of Java source files needs to be compiled, some unit tests need to be run, and a .jar file needs to be created to contain all classes. Coding all these steps for every project could be annoying, and could result in a huge waste of time. Fortunately, Gradle solves this problem with the use of the Java plugin. A plugin is an extension to Gradle that configures the project in some way, typically by adding some preconfigured tasks that together do something useful.

To use the Java plugin, a series of conventions must be used. That means that the plugin defines default values for many aspects of the project. These aspects include Java sources, production resources, and tests location. Following these conventions in a project, you don't need to do much in the build script to get a useful build.

Creating a Java project

To create a Java project in Gradle, the Java plugin needs to be applied in the build script, such as in the following example.

Code Listing 56

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

The previous code is all you need to define a Java project. The Java plugin will be applied to the project, and will add a number of tasks to it.

Project layout

The Java plugin is convention based. This was explained in the "Taking a Java quick start" section of this chapter. So, the Java plugin assumes a certain layout for the project. This layout is shown in the following table.

Table 2: Java plugin project layout

Directory	Meaning
src\main\java	Production Java source code
src\main\resources	Production resources
src\test\java	Test Java source
src\test\resources	Test resources

When the build script ends its execution, the result of the build is saved in a directory named **build**. This directory contains the following relevant subdirectories.

Table 3: Build directory subdirectories

Subdirectory	Meaning
build\classes	Contains the compiled .class files.

Subdirectory	Meaning
build\libs	Contains the .jar or .war files created by the build.

Creating Java code

In order to build the Java project, a subdirectory named **hello** should be created in the **src\main\java** directory. Next, the file **HelloWorld.java** should be saved with the following code.

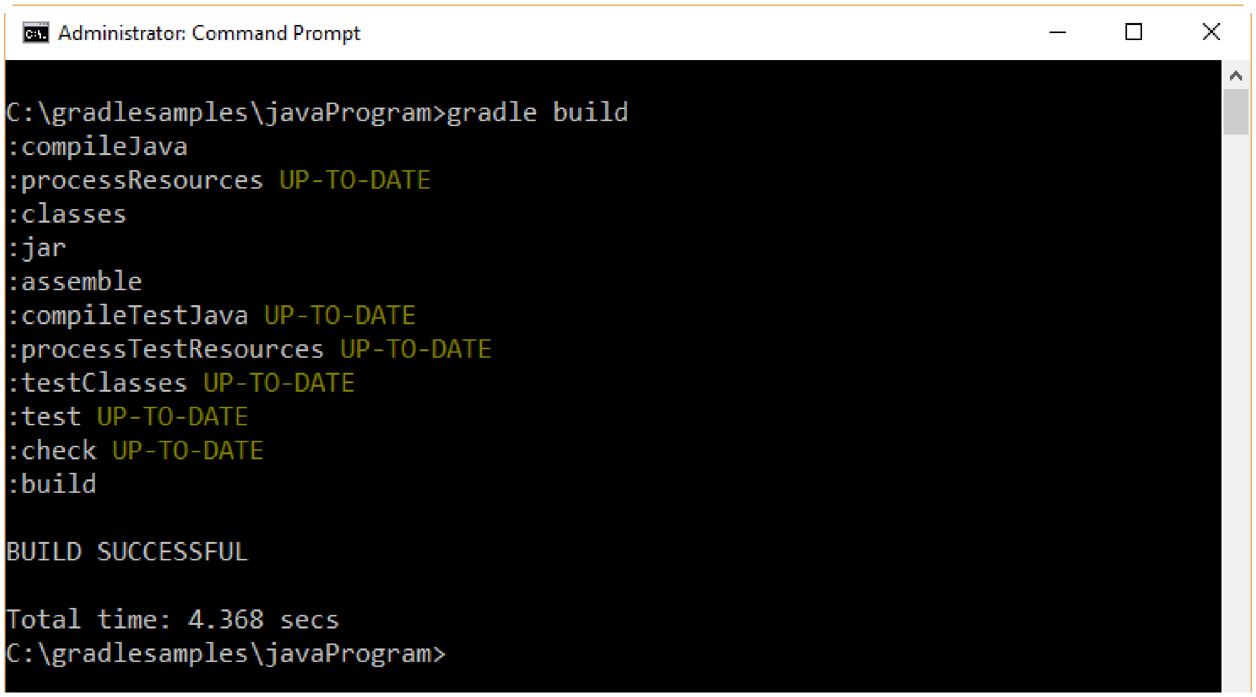
Code Listing 57

```
package hello;

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

Now, the following command will build the Java executable .jar file in the **build\libs** directory. The file will be saved as **javaProgram.jar**.

Code Listing 58

```
gradle build
```



```
C:\gradlesamples\javaProgram>gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL

Total time: 4.368 secs
C:\gradlesamples\javaProgram>
```

Figure 29: Build task output

Running **javaProgram.jar** with the following command...

Code Listing 59

```
java -jar build\libs\javaProgram.jar
```

...will display the following output.

Code Listing 60

```
no main manifest attribute, in build\libs\javaProgram.jar
```

This problem is the result of not configuring the main class of the .jar file. This **`main`** class can be configured in a file known as the manifest file.

Configuring the main class of a .jar file

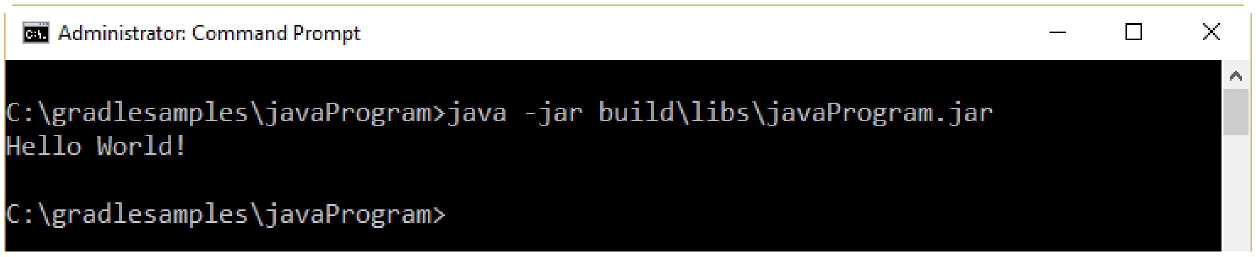
The Java plugin exposes a **`jar`** task into the project. Every **`jar`** object has a **`manifest`** property, which is an instance of the **`manifest`** class. A detailed explanation for these classes is out of the scope of this book, but in this section, we will explain how to use the **`jar`** object to configure the **`main`** class for the .jar executable.

The **`attributes`** method of the **`manifest`** instance is used to set the entry point of an application. This can be accomplished by using a map that contains key-value pairs, and these attributes will be added to the manifest file. The value of the **`Main-Class`** attribute must be set with the name of the class, which acts as the entry point of the program. The changes to the **build.gradle** file should look like this.

Code Listing 61

```
apply plugin: 'java'

jar {
    manifest {
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}
```

A new .jar file will be created by issuing the **`gradle build`** command. After that, the user can run the .jar file. The output displayed by the program is shown in the following figure.



```
C:\gradlesamples\javaProgram>java -jar build\libs\javaProgram.jar
Hello World!

C:\gradlesamples\javaProgram>
```

Figure 30: The Hello World class in action

Learning about the tasks of a Java project

The Java plugin adds many tasks to the build. The most relevant tasks for the purposes of this book are the following:

- The **`assemble`** task: compiles the source code of the application and packages it into a .jar file, but doesn't run the unit tests.
- The **`build`** task: performs a full build of the project, including code compilation, unit tests and .jar file packaging.
- The **`clean`** task: deletes the build directory.
- The **`compileJava`** task: compiles the source code of the project.
- The **`check`** task: compiles and tests the code.

It's a good idea to get an overview of the project without reading the build script. To do so, the following command should be used.

Code Listing 62

```
gradle tasks
```

The output should look like this.

Code Listing 63

```
:tasks

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

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that
depend on it.
buildNeeded - Assembles and tests this project and all projects it depends
on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.

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

Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.
```

```
Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root
project 'javaProgramv2'.
components - Displays the components produced by root project
'javaProgramv2'. [incubating]
dependencies - Displays all dependencies declared in root project
'javaProgramv2'.
dependencyInsight - Displays the insight into a specific dependency in root
project 'javaProgramv2'.
help - Displays a help message.
model - Displays the configuration model of root project 'javaProgramv2'.
[incubating]
projects - Displays the sub-projects of root project 'javaProgramv2'.
properties - Displays the properties of root project 'javaProgramv2'.
tasks - Displays the tasks runnable from root project 'javaProgramv2'.

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.

Rules
-----
Pattern: clean<TaskName>: Cleans the output files of a task.
Pattern: build<ConfigurationName>: Assembles the artifacts of a
configuration.
Pattern: upload<ConfigurationName>: Assembles and uploads the artifacts
belonging to a configuration.

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

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

BUILD SUCCESSFUL

Total time: 4.637 secs
```

The previous example shows the full list of runnable tasks and their descriptions. As can be seen at the bottom of the previous example, if the user needs more detailed information about a particular task, the following command should be used.

Code Listing 64

```
gradle help --task <task>

Where <task> is the name of the task to be inquired.
```

The following example gets detailed information about the **`clean`** task.

Code Listing 65

```
gradle help --task clean
```

The output should look like this.

Code Listing 66

```
:help
Detailed task information for clean

Path
     :clean

Type
     Delete (org.gradle.api.tasks.Delete)

Description
     Deletes the build directory.

Group
     build

BUILD SUCCESSFUL

Total time: 3.652 secs
```

The attributes displayed for a task by the **`help`** command are the following:

- **`Path`**: the task's path (as discussed in the "Properties of DefaultTask" section in Chapter 5).
- **`Type`**: the task type (as discussed in the "About task types" section in Chapter 5).
- **`Description`**: a brief description about the purpose of the task.
- **`Group`**: the name of the group to which the task belongs (in the example, the **`clean`** task is part of the **`build`** group).

Setting package version information for the project

Sometimes, the user may need to include package version information in the build. This can be accomplished by setting a set of headers, using the **`manifest`** method of the **`jar`** object. These headers are summarized in the following table.

Table 4: Headers in the manifest file

Header	Definition
`Name`	The name of the package.

Header	Definition
`Specification-Title`	The title for describing the package.
`Specification-Version`	The version number of the package.
`Specification-Vendor`	The vendor name.
`Implementation-Title`	The title of the implementation that is being packaged.
`Implementation-Version`	The build number of the implementation.
`Implementation-Vendor`	The vendor of the implementation.

The build script should look like the following example.

Code Listing 67

```
apply plugin: 'java'

jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}
```

The .jar file can be built as usual. But now, the manifest file will have all package information after the build process.

Creating a distribution for being shipped

The following build script will create a .zip file from the **`jar`** program, in order to be shipped to any customer.

Code Listing 68

```
/* Declaring default tasks will ensure the zip packaging after the build.
*/
```

```
defaultTasks 'build', 'packageDistribution'

/* Execute gradle with no tasks, in order to succeed. */

apply plugin: 'java'

/* Adding the needed attributes to the manifest file. */
jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}

/* This task will be executed after the build task
   and will create a zip file with the contents of
   the build\libs directory (the jar program).

   The zip file will be saved in the
   build\distributions folder, and will be named
   distPackage-1.0.zip                             */
task packageDistribution(type: Zip){
    from 'build\\libs'
    baseName = 'distPackage'
    version = '1.0'
}
```

This build script takes advantage of **`defaultTasks`** and the **`Zip`** task type, in order to create a distribution package in a .zip compressed format.

As explained within the code, issuing the **`gradle`** command with no task name will automatically execute all tasks declared by the **`defaultTasks`** statement. In this case, the **`build`** task will be executed first, and after the build process ends, the **`packageDistribution`** task will create the .zip file with the **`jar`** program in it.

Chapter summary

This chapter has gone beyond the simple **build.gradle** file used for saving build scripts. Different names can be used for a build script file, and you can look for this named file by using the **`-b`** option from the command line.

You can also execute a build script without specifying a task name. In this case, the use of the **defaultTasks** statement allows you to indicate which tasks, given in a quoted and comma-separated list, are going to be executed. The order of execution is always from left to right.

In order to demonstrate that Gradle can build something, the rest of the chapter describes how to create a Java project using the java plugin. A plugin is an extension to Gradle that configures the project in some way, typically by adding some pre-configured tasks.

The Java plugin is convention-based. That means that the plugin defines default values for many aspects of the project, including Java sources, production resources, and tests location.

For creating a Java project, a specific directory layout must exist within the project's directory, since Gradle searches for that layout during the build process. All results of a build are saved in the **build** directory. This directory is created automatically if it doesn't exist.

The relevant tasks added by the Java plugin, for the purposes of this book, are: **assemble**, **clean**, **compileJava**, **build**, and **check**. Assemble compiles Java code and creates the .jar executable file, but doesn't run the unit tests; **clean** deletes the build directory; **compileJava** compiles the Java code only; **build** performs a full build for the project, including code compilation, unit tests, and .jar packaging; and **check** compiles and tests the code.

A Java project built with Gradle needs to be configured around which class is going to be the entry point for the application. The Java plugin exposes a **jar** task into the project, which has **manifest** property. The **attributes** method of this property is used to set the entry point by assigning to the **Main-class** attribute of this method the name of the class that will act as the entry point.

The **manifest** property can also be used to add detailed information about a jar package. The attributes that can hold this information are **Name**, **Specification-Title**, **Specification-Title**, **Specification-Version**, **Specification-Vendor**, **Implementation-Title**, **Implementation-Version**, and **Implementation-Vendor**.

A build script can take advantage of the **defaultTasks** statement and the **Zip** task type in order to create a .zip distribution package automatically, after the build process.

Chapter 7 Build Hooks

Overview

Gradle is a highly customizable tool for creating custom build software. It exposes a rich set of APIs for introducing novel functionality into a build. But the ability to customize a build doesn't end with plugins.

Gradle also offers the ability to modify the execution of a build by hooking a variety of events that occur during the configuration and execution of a build. These hooks can run when tasks are added, when projects are created, and at other times during configuration sequence.

Reviewing the Gradle lifecycle

As explained in Chapter 4, a Gradle build always proceeds through three phases exactly in the same order: initialization, configuration, and execution.

The initialization phase lets Gradle start up and locates all the build files it must process. In this phase, Gradle also determines if the build is a single-project or multi-project build. For a single-project build, Gradle identifies a single build file and continues to the next phase. For a multi-project build, Gradle locates all the build files needed for processing and continues to the next phase.

The next phase is configuration. In this phase, Gradle executes each build file as a Groovy script, but this is not the actual execution of build actions. This is the creation of a directed acyclic graph (DAG) of task objects (explained in the "Configuration blocks" section of Chapter 5). Many hooks can run during the construction of this graph.

The last phase is execution. In this phase, Gradle identifies the tasks that must be executed by looking into the DAG. Then, Gradle executes them in dependency order (as explained in the "Task dependencies" section of Chapter 4). All build activities occur during the execution phase and, as in the configuration phase, some hooks can run during execution.

The "advice" terminology

A paradigm for managing the complexity of enterprise Java software became marginally popular in the early 2000s. This paradigm was known as aspect-oriented programming (AOP). AOP focused on stating that programs often contain individual functional units that need to be enhanced with code, with a purpose not directly related to the unit in question. For example, a method that writes into a database might have to set up a database context first, and a transaction committed after.

AOP developed a particular vocabulary and some Java frameworks emerged with an AOP implementation. A special name was given to the code that ran before and after the original method: an *advice* action. The advice code relies on this principle: the code block that runs before the original method has no correlation with the code that runs after, even though the execution sequence does.

During the explanation of build hooks, the “advice” special name will be used.

Advices to a project evaluation

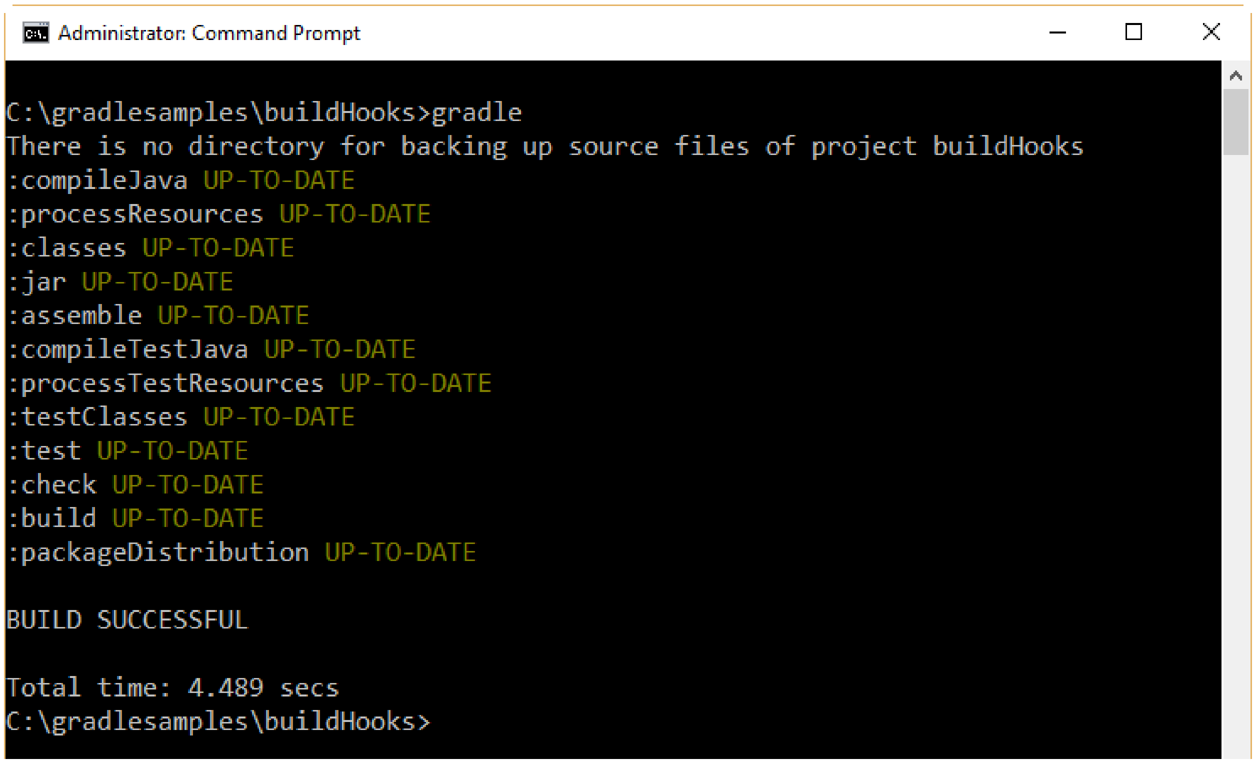
An individual Gradle build file sets up a project with all of the settings and tasks it needs in order to give Gradle useful work to do during the execution phase. This setup occurs during the configuration phase, and notifications can be received about a project as it’s evaluated.

The **`project.afterEvaluate`** method is used to execute code after the project has been evaluated.

Code Listing 69

```
/* Declaring default tasks will ensure the zip packaging after the build.
*/
defaultTasks 'build', 'packageDistribution'

/* Execute gradle with no tasks, in order to succeed. */

apply plugin: 'java'

/* Adding the needed attributes to the manifest file. */
jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}

/* This task will be executed after the build task
   and will create a zip file with the contents of
   the build\libs directory (the jar program).

   The zip file will be saved in the
   build\distributions folder, and will be named
   distPackage-1.0.zip                             */
task packageDistribution(type: Zip){
    from 'build\\libs'
    baseName = 'distPackage'
    version = '1.0'
}

/* In this case, and for testing purposes only, a directory
   named backuprepo is searched within the project's
   tree structure. If such directory doesn't exist, a message
   is displayed.                                             */
afterEvaluate {
    if (!file('backuprepo').isDirectory())
    {
       println "There is no directory for backing up source files of
project  {project.name}"
    }
}
```

In the previous example, the **`project.afterEvaluate`** method searches for a given directory within the project's directory tree structure. In this case, a simple message is displayed when that directory isn't found. As shown in code, there is a **`project`** object, which is available at project's evaluation. In this case, the code uses the **`project`** object to display the project's name in the warning message. This is trivial, of course, but this advice can have many applications, such as setting up a proper directory structure to be used by the execution phase, sending notifications to the project managers, etc.



```
C:\gradlesamples\buildHooks>gradle
There is no directory for backing up source files of project buildHooks
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar UP-TO-DATE
:assemble UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build UP-TO-DATE
:packageDistribution UP-TO-DATE

BUILD SUCCESSFUL

Total time: 4.489 secs
C:\gradlesamples\buildHooks>
```

Figure 31: Project.afterEvaluate example output

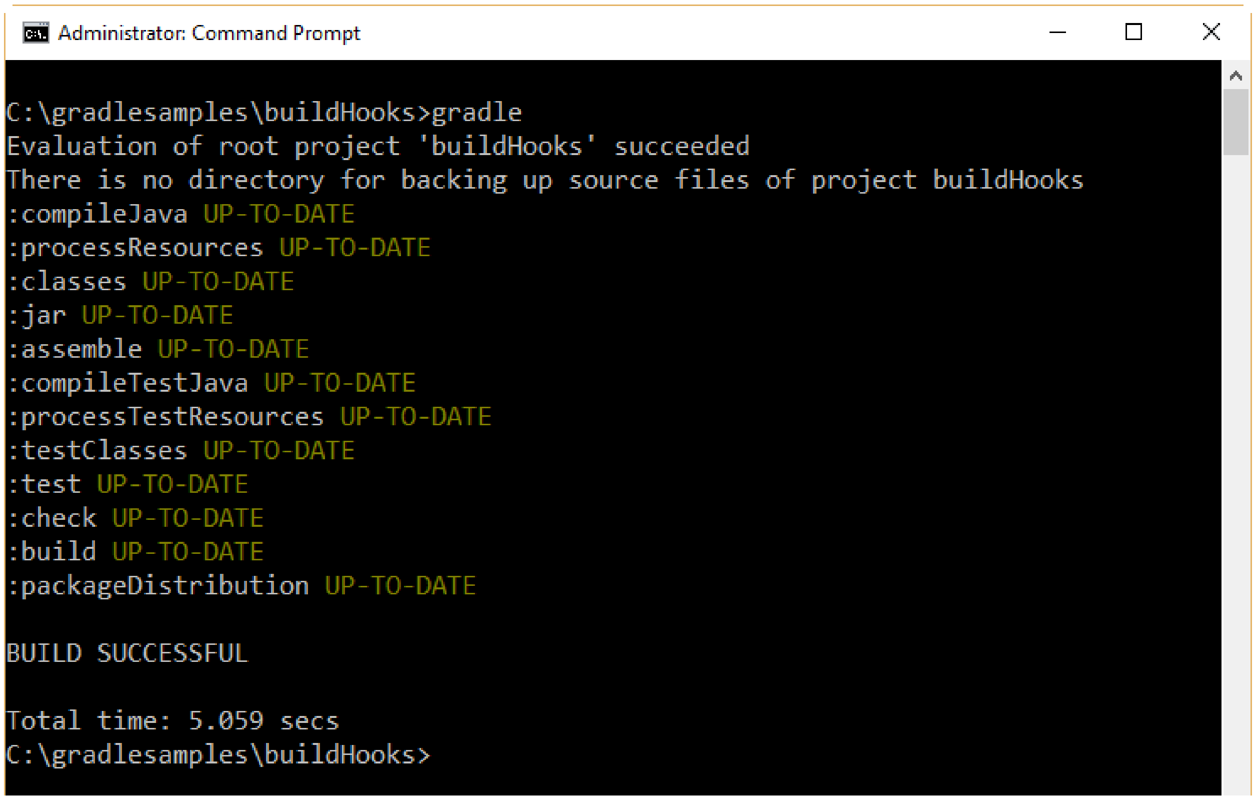
The result of a project's evaluation can be inquired by means of an advice. To do this, we use the **`afterProject`** method of the **`gradle`** object. This method takes two parameters in order to handle the project. The first one is the **`project`** object itself, and the second parameter is the state of the project after its evaluation.

Code Listing 70

```
/* Declaring default tasks will ensure the zip packaging after the build.
*/
defaultTasks 'build', 'packageDistribution'

/* Execute gradle with no tasks, in order to succeed. */

apply plugin: 'java'

/* Adding the needed attributes to the manifest file. */
jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}

/* This task will be executed after the build task
   and will create a zip file with the contents of
   the build\libs directory (the jar program).

   The zip file will be saved in the
   build\distributions folder, and will be named
   distPackage-1.0.zip                             */
task packageDistribution(type: Zip){
    from 'build\\libs'
    baseName = 'distPackage'
    version = '1.0'
}

/* In this case, and for testing purposes only, a directory
   named backuprepo is searched within the project's
   tree structure. If such directory doesn't exist, a message
   is displayed.                                           */
afterEvaluate {
    if (!file('backuprepo').isDirectory())
    {
        println "There is no directory for backing up source files of
project ${project.name}"
```

```
    }
}

/* The result of the evaluation for a project can be
   trapped with this code.                              */
gradle.afterProject {project, projectState ->
   if (projectState.failure) {
        println "Evaluation of $project FAILED"
   }
   else
   {
       println "Evaluation of $project succeeded"
   }
}
```

The output for the previous code is displayed in the following figure.



Administrator: Command Prompt

```
C:\gradlesamples\buildHooks>gradle
Evaluation of root project 'buildHooks' succeeded
There is no directory for backing up source files of project buildHooks
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar UP-TO-DATE
:assemble UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build UP-TO-DATE
:packageDistribution UP-TO-DATE

BUILD SUCCESSFUL

Total time: 5.059 secs
C:\gradlesamples\buildHooks>
```

Figure 32: AfterProject sample output

Note the sequence of how both **`afterProject`** and **`afterEvaluate`** advices are fired in Figure 32. In this case, **`afterProject`** takes place just after project's evaluation is ended. Then, the code associated to this advice is executed. Once this execution ends, the **`afterEvaluate`** advice code is executed.

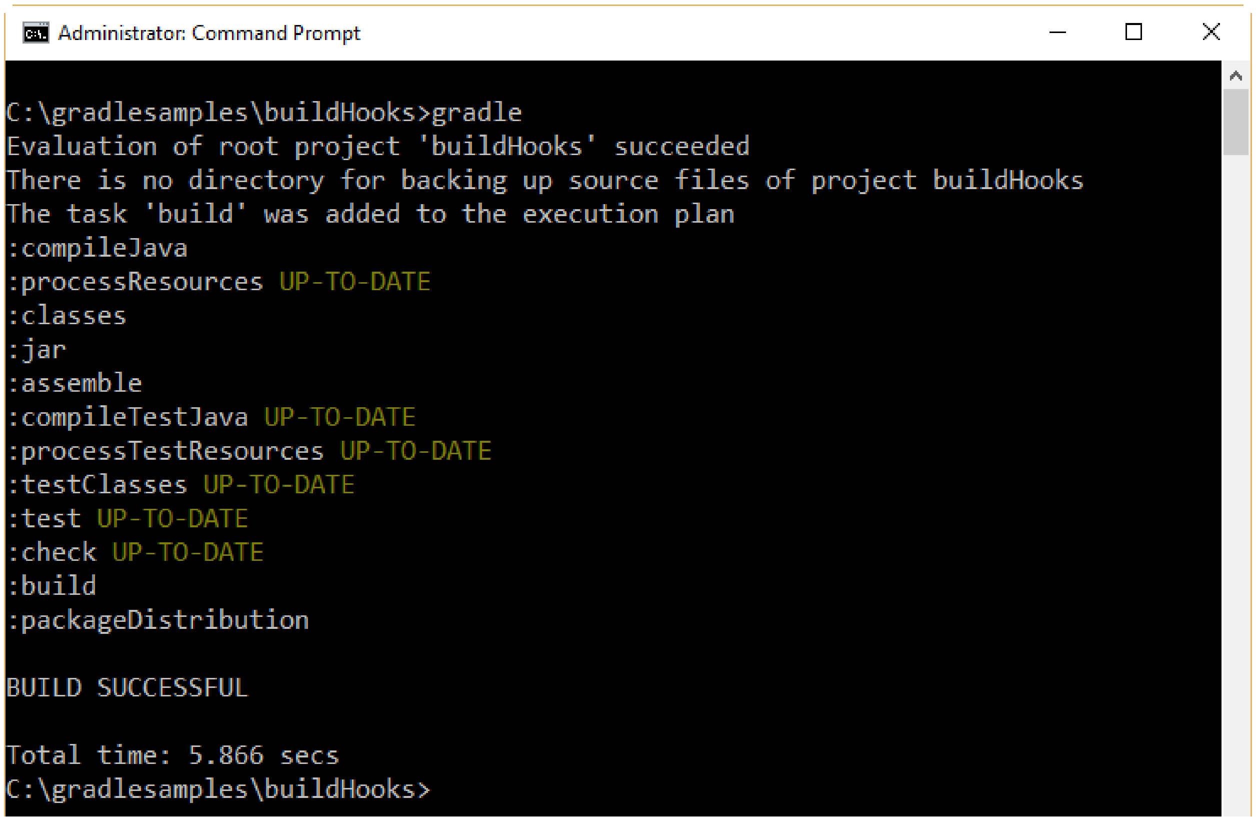
Advices to tasks creation

An advice can be attached to notify you when a task is added to a project. This can be useful to set some values that are needed before the task is made available for the build. The **`whenTaskAdded`** method of the **`tasks`** object is used for that purpose. This method receives the **`task`** object as a parameter.

Code Listing 71

```
/* Declaring default tasks will ensure the zip packaging after the build.
*/
defaultTasks 'packageDistribution'

/* Execute gradle with no tasks, in order to succeed. */

apply plugin: 'java'

/* Adding the needed attributes to the manifest file. */
jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}

/* This task will be executed after the build task
   and will create a zip file with the contents of
   the build\libs directory (the jar program).

   The zip file will be saved in the
   build\distributions folder, and will be named
   distPackage-1.0.zip                             */
task packageDistribution(type: Zip){
    from 'build\\libs'
    baseName = 'distPackage'
    version = '1.0'
}

packageDistribution {
    dependsOn 'build'
}

/* In this case, and for testing purposes only, a directory
   named backuprepo is searched within the project's
   tree structure. If such directory doesn't exist, a message
   is displayed.                                             */
afterEvaluate {
    if (!file('backuprepo').isDirectory())
    {
        println "There is no directory for backing up source files of
project ${project.name}"
    }
}

/* The result of the evaluation for a project can be
   trapped with this code.                          */
gradle.afterProject {project, projectState ->
   if (projectState.failure) {
        println "Evaluation of $project FAILED"
   }
   else
   {
       println "Evaluation of $project succeeded"
   }
  }

tasks.whenTaskAdded { task ->
    println "The task '${task.name}' was added to the execution plan"
}
```

The previous code simply displays the task’s name when it is added to the project. The output for this build is displayed in the following figure.



```
C:\gradlesamples\buildHooks>gradle
Evaluation of root project 'buildHooks' succeeded
There is no directory for backing up source files of project buildHooks
The task 'build' was added to the execution plan
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build
:packageDistribution

BUILD SUCCESSFUL

Total time: 5.866 secs
C:\gradlesamples\buildHooks>
```

Figure 33: Task creation advice sample output

According to the output shown in Figure 33, task creation happens after the project's evaluation.

Advices to execution phase

Task execution

Advices can be used before and after any task is executed. This can be done by using the **`beforeTask`** and **`afterTask`** methods of the **`gradle.taskGraph`** object.

The **`beforeTask`** method receives the **`task`** object as a parameter, so the user can query any property of a task in order to make a behavior change, before the task's execution.

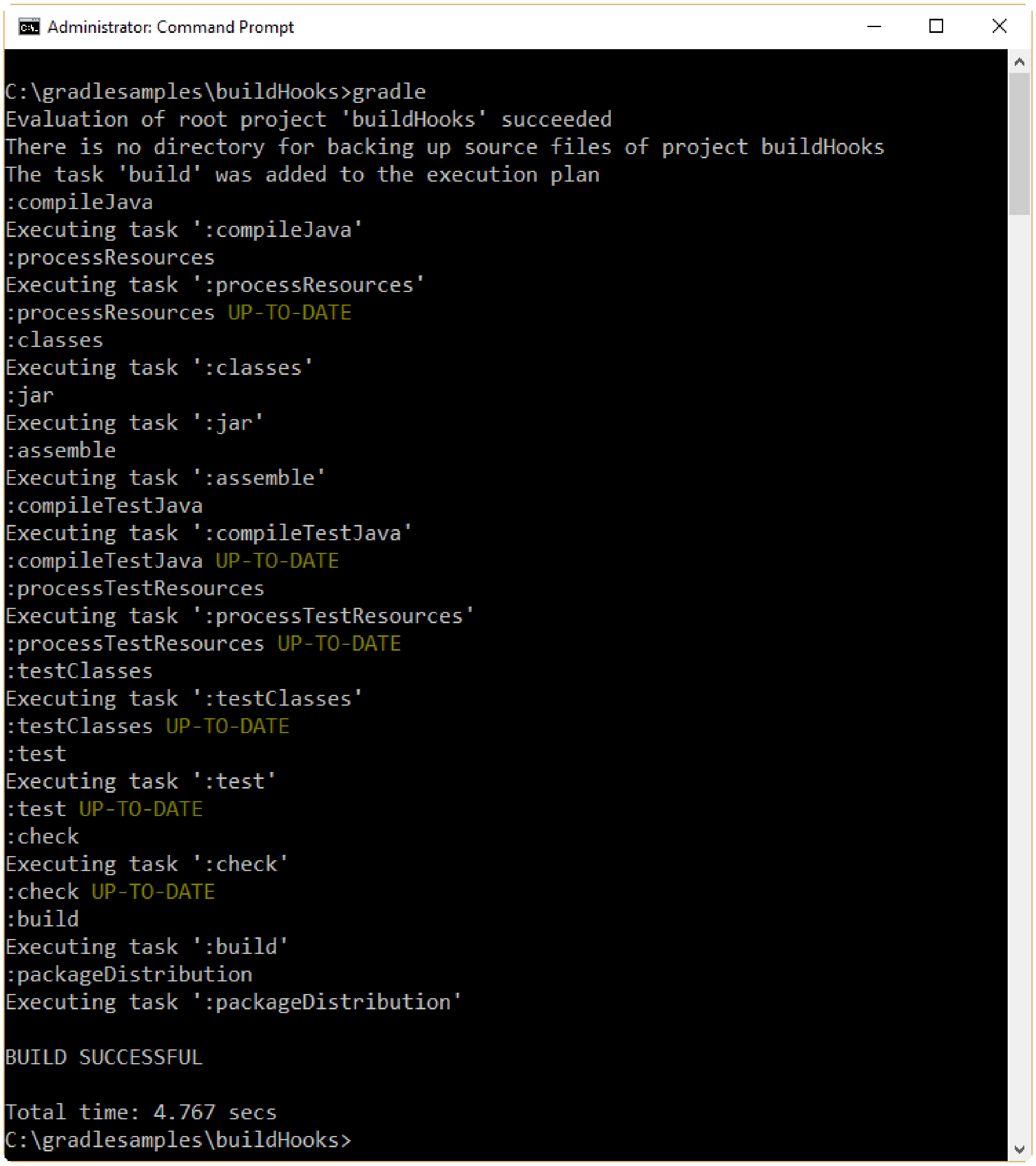
Code Listing 72

```
/* Declaring default tasks will ensure the zip packaging after the build.
*/
defaultTasks 'packageDistribution'

/* Execute gradle with no tasks, in order to succeed. */

apply plugin: 'java'
```

```
/* Adding the needed attributes to the manifest file. */
jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}

/* This task will be executed after the build task
   and will create a zip file with the contents of
   the build\libs directory (the jar program).

   The zip file will be saved in the
   build\distributions folder, and will be named
   distPackage-1.0.zip                             */
task packageDistribution(type: Zip){
    from 'build\\libs'
    baseName = 'distPackage'
    version = '1.0'
}

packageDistribution {
    dependsOn 'build'
}

/* In this case, and for testing purposes only, a directory
   named backuprepo is searched within the project's
   tree structure. If such directory doesn't exist, a message
   is displayed.                                            */
afterEvaluate {
    if (!file('backuprepo').isDirectory())
    {
        println "There is no directory for backing up source files of
project ${project.name}"
    }
}

/* Every time a task is added its name is displayed */
tasks.whenTaskAdded { task ->
    println "The task '${task.name}' was added to the execution plan"
}

/* The result of the evaluation for a project can be
   trapped with this code.                          */
```

```
gradle.afterProject {project, projectState ->
   if (projectState.failure) {
        println "Evaluation of $project FAILED"
   }
   else
   {
       println "Evaluation of $project succeeded"
   }
  }

/* Every task name is displayed before task's execution begins */
gradle.taskGraph.beforeTask {Task task ->
      println "Executing $task"
  }
```

The previous code adds the **`beforeTask`** method to display every task name, before execution begins.



```
C:\gradlesamples\buildHooks>gradle
Evaluation of root project 'buildHooks' succeeded
There is no directory for backing up source files of project buildHooks
The task 'build' was added to the execution plan
:compileJava
Executing task ':compileJava'
:processResources
Executing task ':processResources'
:processResources UP-TO-DATE
:classes
Executing task ':classes'
:jar
Executing task ':jar'
:assemble
Executing task ':assemble'
:compileTestJava
Executing task ':compileTestJava'
:compileTestJava UP-TO-DATE
:processTestResources
Executing task ':processTestResources'
:processTestResources UP-TO-DATE
:testClasses
Executing task ':testClasses'
:testClasses UP-TO-DATE
:test
Executing task ':test'
:test UP-TO-DATE
:check
Executing task ':check'
:check UP-TO-DATE
:build
Executing task ':build'
:packageDistribution
Executing task ':packageDistribution'

BUILD SUCCESSFUL

Total time: 4.767 secs
C:\gradlesamples\buildHooks>
```

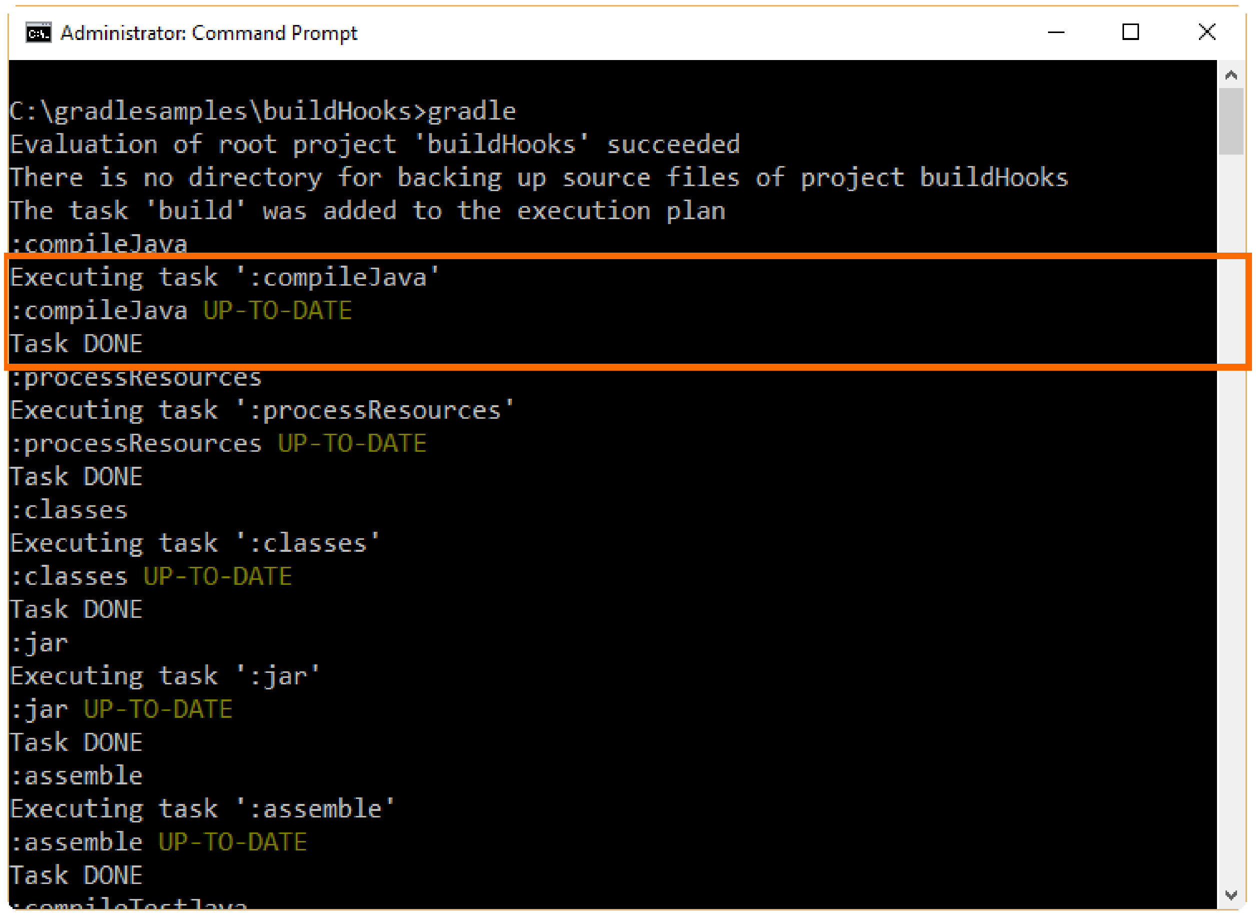
Figure 34: The output generated by the beforeTask method

The **afterTask** method can be used to know the state of a task when the task's execution finishes. This method receives the task object and the **TaskState** as parameters. The following code snippet shows its implementation.

Code Listing 73

```
/* The state for every task is displayed just after its execution. */
```

```
gradle.taskGraph.afterTask {Task task, TaskState state ->
      if (state.failure) {
          println 'Task FAILED'
      }
      else
      {
          println 'Task DONE'
      }
}
```



```
C:\gradlesamples\buildHooks>gradle
Evaluation of root project 'buildHooks' succeeded
There is no directory for backing up source files of project buildHooks
The task 'build' was added to the execution plan
:compileJava
Executing task ':compileJava'
:compileJava UP-TO-DATE
Task DONE
:processResources
Executing task ':processResources'
:processResources UP-TO-DATE
Task DONE
:classes
Executing task ':classes'
:classes UP-TO-DATE
Task DONE
:jar
Executing task ':jar'
:jar UP-TO-DATE
Task DONE
:assemble
Executing task ':assemble'
:assemble UP-TO-DATE
Task DONE
```

Figure 35: The output for the build when afterTask is added

Note: The afterTask method is executed regardless of whether the task is successfully completed or fails with an exception.

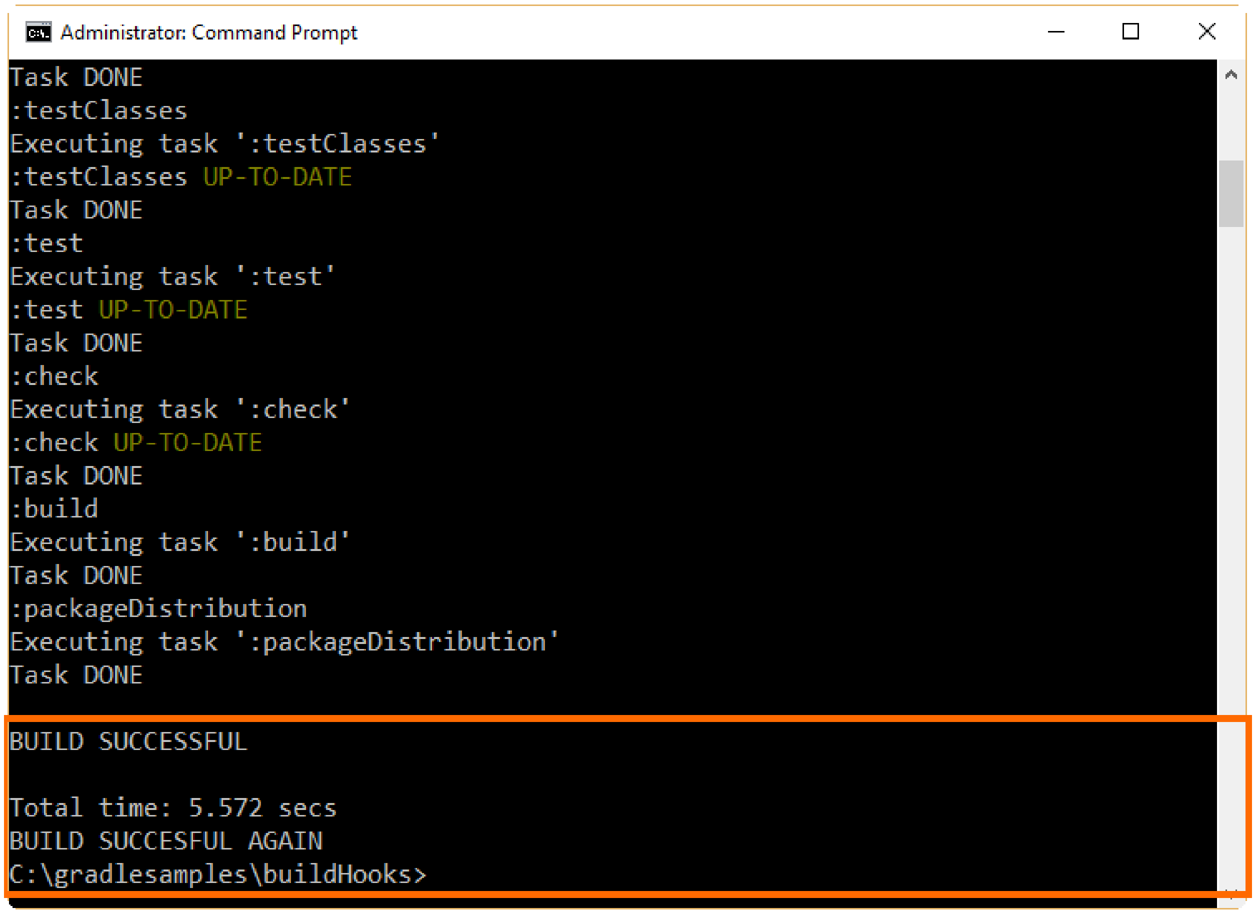
Build finished

Sometimes the user may want to know when a build is finished, and whether it ended successfully or with an error. This can be useful to write a sort of log that keeps track of all build executions.

The method **buildFinished** of the **gradle** object is used to trap a build finished notification. The following code snippet shows the implementation.

Code Listing 74

```
/* If the build fails, the reason why it fails is displayed. */
gradle.buildFinished {buildResult ->
    if (buildResult.failure != null)
    {
        println "Build has failed - ${buildResult.failure}"
    }
    else
    {
        println "BUILD SUCCESFUL AGAIN"
    }
}
```



```
Task DONE
:testClasses
Executing task ':testClasses'
:testClasses UP-TO-DATE
Task DONE
:test
Executing task ':test'
:test UP-TO-DATE
Task DONE
:check
Executing task ':check'
:check UP-TO-DATE
Task DONE
:build
Executing task ':build'
Task DONE
:packageDistribution
Executing task ':packageDistribution'
Task DONE

BUILD SUCCESSFUL

Total time: 5.572 secs
BUILD SUCCESFUL AGAIN
C:\gradlesamples\buildHooks>
```

Figure 36: The output for the buildFinished method

Chapter summary

Gradle is a highly customizable tool for creating custom build software. It exposes a rich set of APIs for introducing novel functionality into a build. Gradle also offers the ability to modify the execution of a build by hooking a variety of events that occur during the configuration and execution of a build.

As explained in Chapter 4, a Gradle build always proceeds through three phases exactly in the same order: initialization, configuration, and execution. The initialization phase lets Gradle start up and locates all the build files it must process. The configuration phase executes each build file as a Groovy script and creates a directed acyclic graph (DAG) of task objects. The execution phase identifies the tasks that must be executed by looking into the DAG, and then executes them in dependency order. All build activities occur during the execution phase.

In the early 2000s, a paradigm known as aspect-oriented programming (AOP) became marginally popular. AOP focused on stating that individual functional units of programs need to be enhanced with code not directly related to the unit in question. AOP developed a particular vocabulary and some Java frameworks emerged with an AOP implementation. A special name was given to the code that ran before and after the original method: *advice*.

The user can create advices for Project Evaluation and tasks creation in the configuration phase. In the execution phase, advices for task execution and **`buildFinished`** can be created.

The **`project.afterEvaluate`** method is used to execute code after the project has been evaluated. Also, the result of a project's evaluation can be found using the **`afterProject`** method of the **`gradle`** object. This method receives two parameters: the project object itself, and the state of the project after its evaluation.

An advice can be attached to notify you when a task is added to a project by using the **`whenTaskAdded`** method of the **`tasks`** object. This method receives the **`task`** object as a parameter.

Advices can be used before and after any task is executed. This can be done by using the **`beforeTask`** and **`afterTask`** methods of the **`gradle.taskGraph`** object. The **`beforeTask`** method receives the **`task`** object as a parameter. The **`afterTask`** method can be used when a task's execution just finishes. This method receives the **`task`** object and the **`TaskState`** as parameters.

Sometimes, the user may want to know when a build is finished, and whether it ended successfully or with an error. The method **`buildFinished`** of the **`gradle`** object is used for this purpose.

Chapter 8 Multi-Project Builds

In Gradle, a multi-project build consists of a root project, and one or more subprojects that may also have subprojects. This scenario is useful in order to organize and understand a project as a set of smaller, interdependent modules. In this case, all the modules are linked together.

Structure of a multi-project build

A multi-project build should follow a consistent structure, which has the following characteristics:

- A **settings.gradle** file in the root project's directory.
- A **build.gradle** file in the root project's directory.
- Child directories who have their own **build.gradle** files (some multi-project builds may omit child projects' build scripts).

The **settings.gradle** file tells Gradle how the project and subprojects are structured. The following command displays a project's structure.

Code Listing 75

```
gradle -q projects
```

The output should look like this.

Code Listing 76

```

------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'multiProject'
\--- Project ':secondproject'

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

```

This tells us that **`multiProject`** has one immediate child project, named **`secondproject`**. The name of the subproject is displayed in a project's path format (as explained in Chapter 5). If a subproject has child projects, these are displayed under its name in a tree structure way. This output maps to the directory structure for all projects. So, **`secondproject`** can be found at the **multiproject\secondproject** directory.

Usually, each project has its own **build.gradle** file. This is not necessary in some cases (for example, when a subproject is only a container for other subprojects), but the root project has one.

The root project's **build.gradle** file is often used to share common configuration between the child projects. For example, it might be needed to apply the same sets of plugins to all of them. It also can be used to configure individual subprojects, if it's preferable to have all projects' configuration in one place. This means that looking into the root build file is a good way to discover how a particular subproject is being configured.

Executing a multi-project build

Multi-project builds are still collections of tasks that Gradle can run. The difference relies on the fact that the user may want to control which project's tasks should be executed. The following options apply to a multi-project build execution.

- You can change to the directory that corresponds to the project you want to run, and then execute **`gradle <taskname>`** as usual.
- You can specify a qualified task name from the directory of the root project. For example, **`gradle :ftputils:build`** will build the **`fptutils`** subproject and any subprojects it depends on.
- You can specify a task name that can be found in the root project, or in any subprojects it depends on, by issuing **`gradle <taskname>`** command.

The first and third options are similar to the single-project use case, but Gradle will work slightly differently in a multi-project case. The following command…

Code Listing 77

```
gradle hello
```

…will execute the **`hello`** task in any subprojects that have the task, relative to the current directory.

The second option gives more control over what gets executed. The following command…

Code Listing 78

```
gradle :secondproject:subProjectTask
```

…will execute the **`subProjectTask`** task of the **`secondproject`** subproject, and all its dependencies.

Paying attention to the following project structure…

Code Listing 79

```

------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'multiProject'
+--- Project ':secondproject'
\--- Project ':secondproject/thirdproject'
```

```

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

…if the user executes the following command…

Code Listing 80

```
gradle subProjectTask
```

…the **`subProjectTask`** task will be executed in every project where it is defined. Gradle will look for this task starting from the root project, and ending in the last project included in the **settings.gradle** file.

The code for the root project is the following.

Code Listing 81

```
task hello << {
    println "I'm $project.name"
}
```

This is the code for **`secondproject`** project:

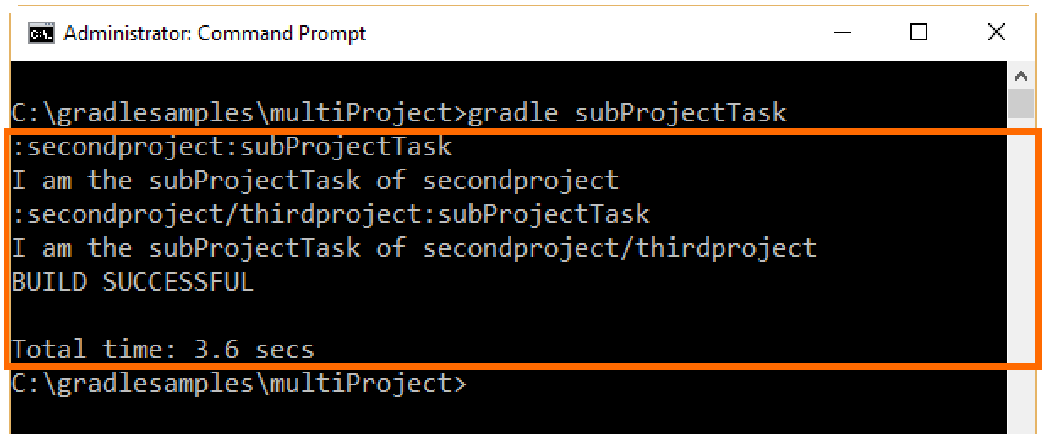
Code Listing 82

```
task hello << {
    println "I'm $project.name"
}

task subProjectTask << {
    println 'I am the subProjectTask of secondproject'
}
```

The code for **`secondproject/thirdproject`** is:

Code Listing 83

```
task hello << {
    println "I'm $project.name"
}

task subProjectTask << {
    print "I am the subProjectTask of $project.name"
}
```


Administrator: Command Prompt

```
C:\gradlesamples\multiProject>gradle subProjectTask
:secondproject:subProjectTask
I am the subProjectTask of secondproject
:secondproject/thirdproject:subProjectTask
I am the subProjectTask of secondproject/thirdproject
BUILD SUCCESSFUL

Total time: 3.6 secs
C:\gradlesamples\multiProject>
```

Figure 37: The output for gradle subProjectTask command

Defining common behavior

As explained previously in this chapter, in a multi-project build it is not necessary for the subprojects to have their own **build.gradle** file. That is, build scripts are optional in a multi-project build. But this doesn't mean that you cannot define a specific behavior for each one of the dependent subprojects. Considering the following project structure…

Code Listing 84

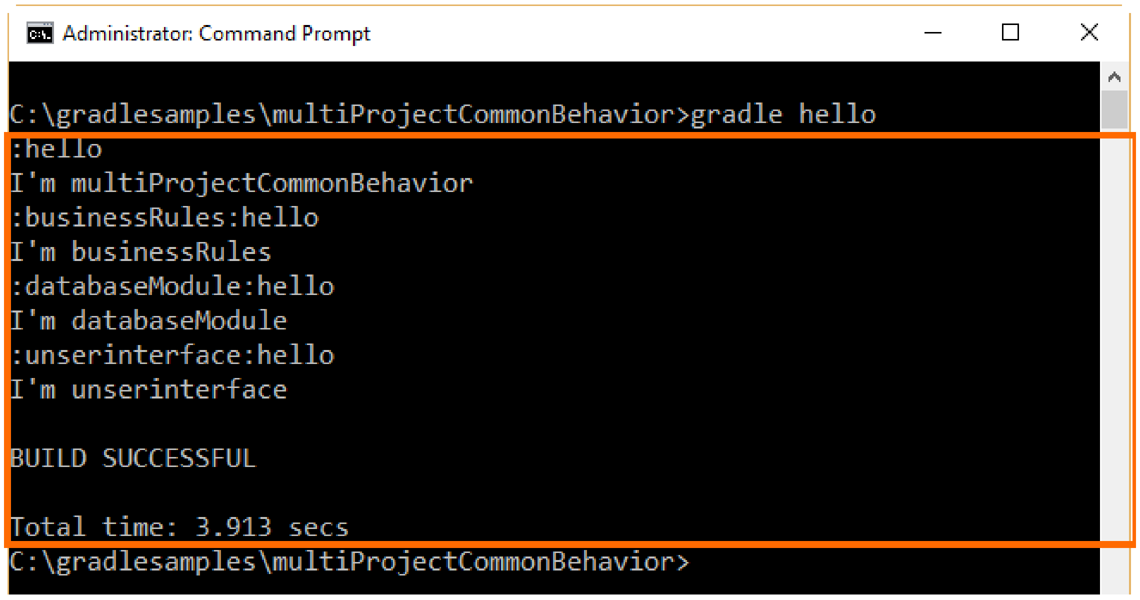
```
------------------------------------------------------------
Root project
------------------------------------------------------------

Root project 'multiProjectCommonBehavior'
+--- Project ':businessRules'
+--- Project ':databaseModule'
\--- Project ':unserinterface'

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

…and the following build script in the **build.gradle** file of the root project…

Code Listing 85

```
Closure cl = { task -> println "I'm $task.project.name" }
allprojects {
    task hello << cl
}
```

…the output for the previous build script is displayed in the following figure.



```
C:\gradlesamples\multiProjectCommonBehavior>gradle hello
:hello
I'm multiProjectCommonBehavior
:businessRules:hello
I'm businessRules
:databaseModule:hello
I'm databaseModule
:unserinterface:hello
I'm unserinterface

BUILD SUCCESSFUL

Total time: 3.913 secs
C:\gradlesamples\multiProjectCommonBehavior>
```

Figure 38: The output for the common behavior example

Gradle provides in its API a property called **`allprojects`**. This property returns a list with the current project and all its subprojects underneath it. If the user calls **`allprojects`** with a closure, all the statements of the closure are delegated to the projects associated with this property. So, despite that there's no **build.gradle** file for the subprojects, the task **`hello`** of the example is delegated for each one of them from the root project's **`build.gradle`** file.

An iteration via **`allprojects.each`** can be done, like in the following example.

Code Listing 86

```
Closure cl = {task -> println "I am the task $task.name of project
$task.project.name"}

allprojects.each {

    project("$it.path") {
        task hello << cl
    }

}
```

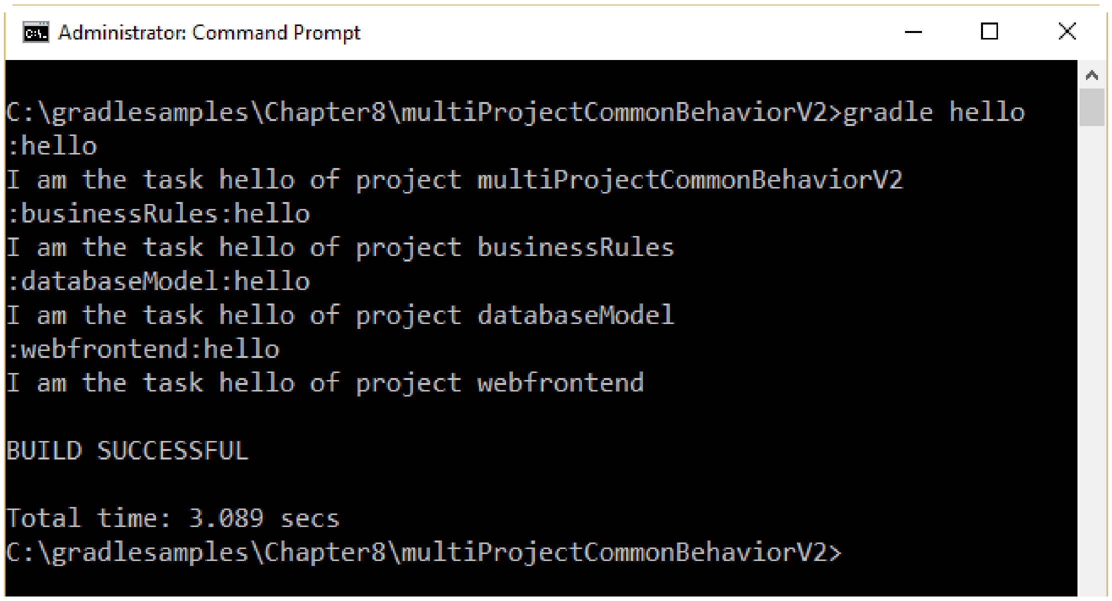
The previous code assigns a closure to an object variable. Then, when iteration via **`allprojects.each`** is done, the closure is assigned to a task named **`hello`**, and the task is assigned to each project using the **`project`** method, which receives the project's path as a parameter.

Now, when the user executes the following command…

Code Listing 87

```
gradle hello
```

…the output displayed looks like the following figure.



```
C:\gradlesamples\Chapter8\multiProjectCommonBehaviorV2>gradle hello
:hello
I am the task hello of project multiProjectCommonBehaviorV2
:businessRules:hello
I am the task hello of project businessRules
:databaseModel:hello
I am the task hello of project databaseModel
:webfrontend:hello
I am the task hello of project webfrontend

BUILD SUCCESSFUL

Total time: 3.089 secs
C:\gradlesamples\Chapter8\multiProjectCommonBehaviorV2>
```

Figure 39: Output for allprojects.each iteration

Defining a specific behavior

The **`allprojects.each`** iteration can be useful for assigning a different closure to a project, depending on its path. Assuming a project structure like the following…

Code Listing 88

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

Root project 'multiProjectCustomBehavior'
+--- Project ':businessRules'
+--- Project ':databaseModel'
\--- Project ':webfrontend'

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

...the code shown in the following snippet should be used to accomplish this goal.

Code Listing 89

```
/* Four closures are defined. */
Closure clbr = {task -> println "I am the task $task.name of project
$task.project.name. I'm in charge of Business Rules."}
Closure cldm = {task -> println "I am the task $task.name of project
$task.project.name. I'm in charge of Database Model."}
```

```
Closure clwf = {task -> println "I am the task $task.name of project
$task.project.name. I'm in charge of Web Front End displaying."}
Closure clroot = {task -> println "I am the task $task.name of project
$task.project.name. I'm the entry point for all process."}

/* Depending on the project's path, a different closure is assigned to the
task hello.
   This produces a specific behavior for the same task, depending on which
project is
   executed.
*/
allprojects.each {

    project("$it.path") {
        switch (it.path) {
            case ":businessRules" : task hello << clbr ; break;
            case ":databaseModel" : task hello << cldm ; break;
            case ":webfrontend" : task hello << clwf ; break ;
            default: task hello << clroot;
        }
    }
}
```

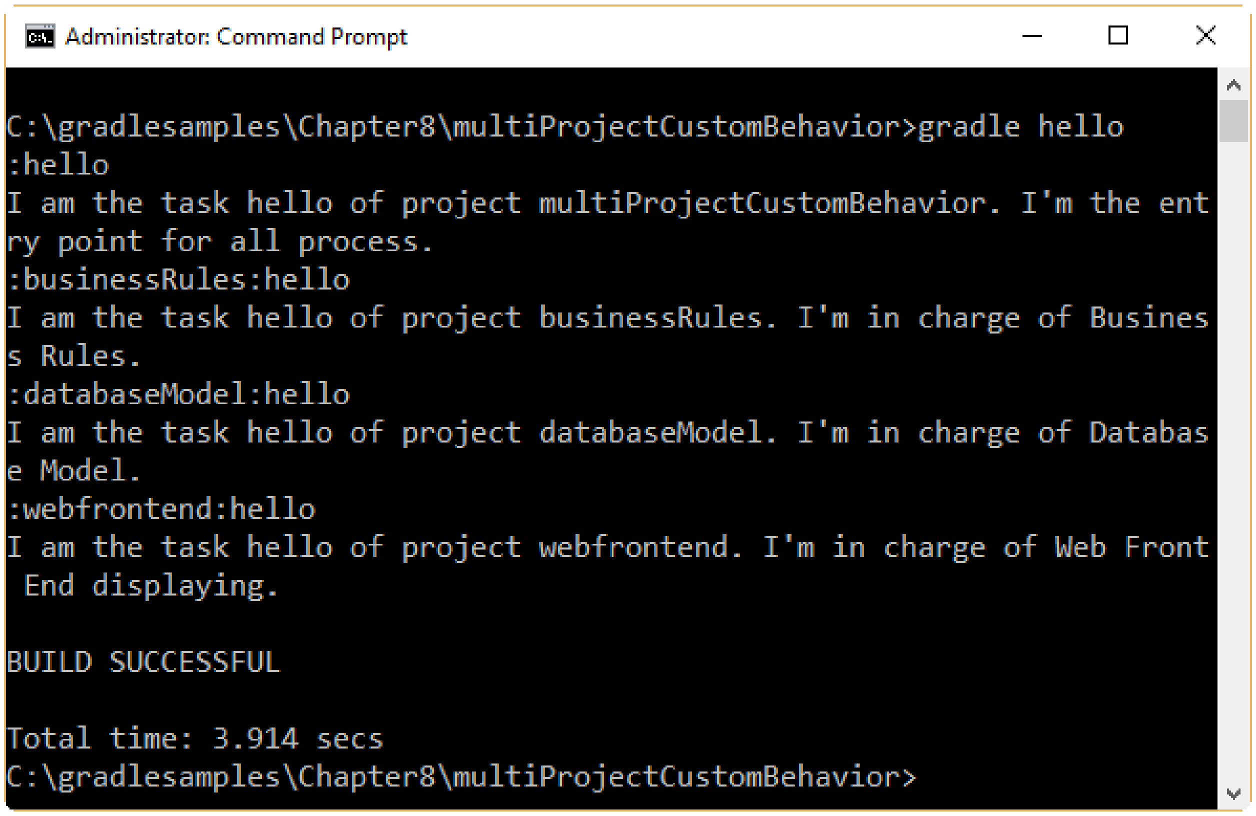
The code assigns one of the four closures declared at the beginning of the build script to the **`hello`** task. This assignment depends on which project is being executed. The build script evaluates the name of the project using the **`path`** property of the **`project`** object, which is stored in the implicit **`it`** variable created by the **`each`** loop. A **`switch`** statement, which is placed within the curly braces of the **`project`** method, makes all comparisons needed to assign the proper closure to the task. In some way, the previous code uses the polymorphism ability of object-oriented programming.

If you execute the following command…

Code Listing 90

```
gradle hello
```

…the output should look like the following figure.



```
C:\gradlesamples\Chapter8\multiProjectCustomBehavior>gradle hello
:hello
I am the task hello of project multiProjectCustomBehavior. I'm the ent
ry point for all process.
:businessRules:hello
I am the task hello of project businessRules. I'm in charge of Busines
s Rules.
:databaseModel:hello
I am the task hello of project databaseModel. I'm in charge of Databas
e Model.
:webfrontend:hello
I am the task hello of project webfrontend. I'm in charge of Web Front
 End displaying.

BUILD SUCCESSFUL

Total time: 3.914 secs
C:\gradlesamples\Chapter8\multiProjectCustomBehavior>
```

Figure 40: Output for multiProjectCustomBehavior

Subproject configuration

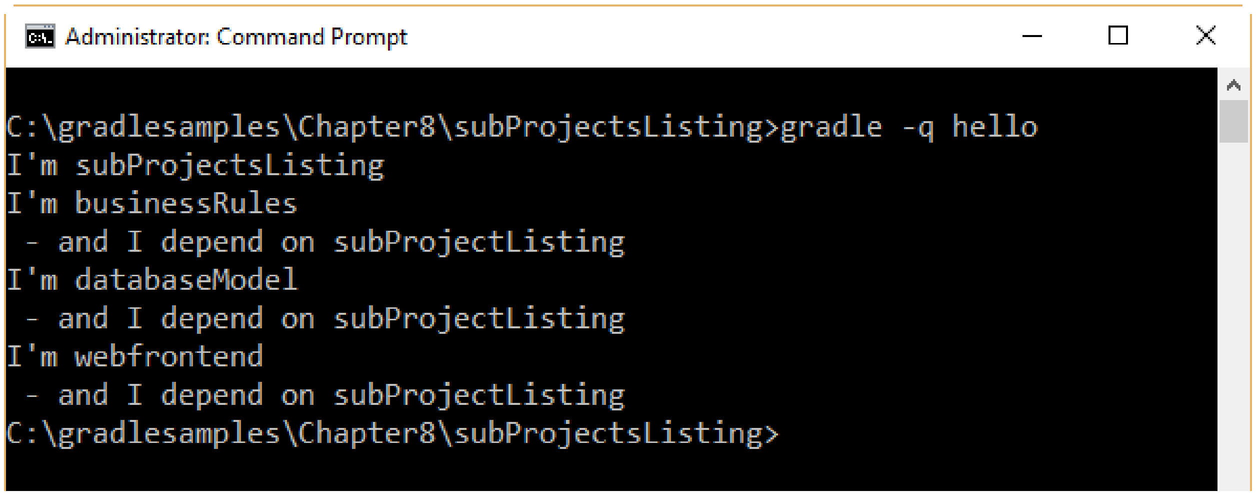
The subprojects property

There's also a way to access the subprojects only. This can be done by using the **`subprojects`** property. This property performs an iteration through the subprojects of the build only. The following code shows how to perform this iteration.

Code Listing 91

```
allprojects {
    task hello << {task -> println "I'm $task.project.name"}
}

subprojects {
    hello << {println " - and I depend on subProjectListing"}
}
```

The output for this build is shown in the following figure.


Administrator: Command Prompt

```
C:\gradlesamples\Chapter8\subProjectsListing>gradle -q hello
I'm subProjectsListing
I'm businessRules
 - and I depend on subProjectListing
I'm databaseModel
 - and I depend on subProjectListing
I'm webfrontend
 - and I depend on subProjectListing
C:\gradlesamples\Chapter8\subProjectsListing>
```

Figure 41: SubProjectsListing output

The projects' execution order can be determined by looking at the output displayed in Figure 41. First, at the configuration phase, the **`allprojects`** loop assigns the **`hello`** task to the root project and its dependents. When this action finishes, the **`subprojects`** loop assigns a new action for every **`hello`** task belonging to each one of them. Since **`allprojects`** assigned an action in the first place, the action added to each subproject will be executed after the first one assigned. Then, the execution phase produced the results shown in the figure.

Adding specific behavior to a given subproject

Like the **`allprojects`** property, **`subprojects`** has an **`each`** iteration, which allows users to access subprojects individually. This iteration can be useful if you want to add a specific behavior to any of the dependent subprojects.

Code Listing 92

```
allprojects {
    task hello << {task -> println "I'm $task.project.name"}
}

subprojects.each {
   project("$it.path").hello << {
        println " - and I depend on subProjectsBehavior"
    }

    if (it.path == ":webfrontend")
    {
        project("$it.path").hello << {
            println " - and I manage Web User Interface, too."
        }
    }
}
```

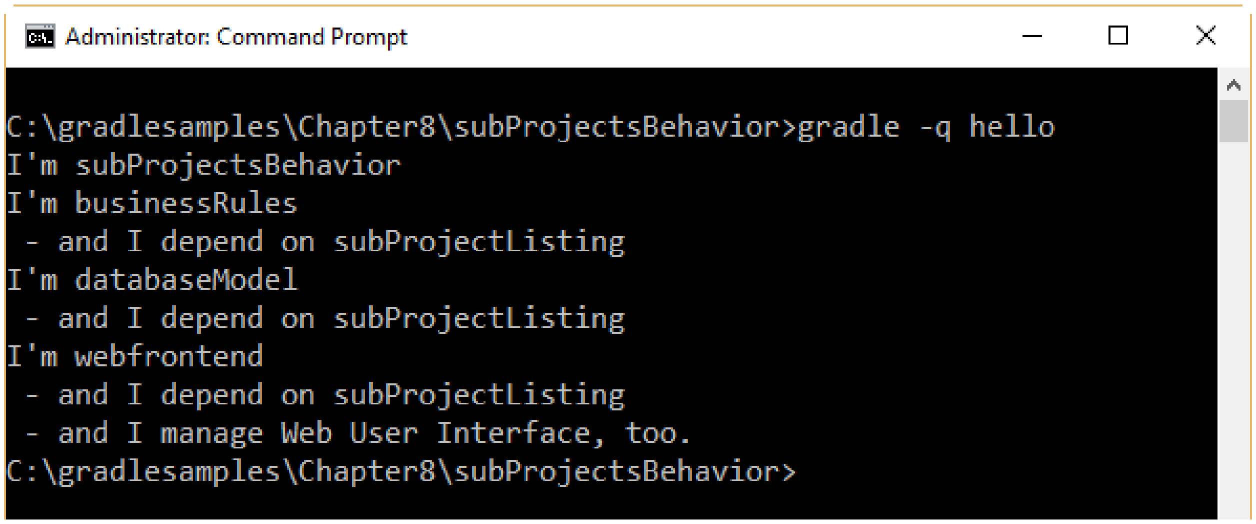
This code is derived from the example discussed in the previous section. The **`each`** iteration for the **`subproject`** property is used in order to learn the name of each dependent subproject. The **`hello`** task for every subproject receives a closure, which displays the phrase “ - and I depend on subProjectsBehavior,” and when the **`:webfrontend`** subproject is evaluated, only its **`hello`** task receives a new closure, which displays the “- and I manage Web User Interface, too." statement.

When the following command is executed…

Code Listing 93

```
gradle -q hello
```

…the following output will be displayed.



```
C:\gradlesamples\Chapter8\subProjectsBehavior>gradle -q hello
I'm subProjectsBehavior
I'm businessRules
 - and I depend on subProjectListing
I'm databaseModel
 - and I depend on subProjectListing
I'm webfrontend
 - and I depend on subProjectListing
 - and I manage Web User Interface, too.
C:\gradlesamples\Chapter8\subProjectsBehavior>
```

Figure 42: Output for subProjectsBehavior build script

Adding specific behavior using build.gradle files

Another way to add specific behavior to a subproject is to create its own **build.gradle** build script file. Assuming a project structure like the following…

Code Listing 94

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

Root project 'subProjectsBehaviorV2'
+--- Project ':businessRules'
+--- Project ':databaseModel'
\--- Project ':webfrontend'

To see a list of the tasks of a project, run gradle <project-path>:tasks
```

```
For example, try running gradle :businessRules:tasks
```

…a **build.gradle** file will be created for every subproject.

Code Listing 95

```
/* businessRules\build.gradle */
hello.doLast {
    println " - and I'm in charge of Business Rules along the entire
application."
}
```

Code Listing 96

```
/* databaseModel\build.gradle */
hello.doLast {
    println " - and I'm in charge of database management."
}
```

Code Listing 97

```
/* webfrontend\build.gradle */
hello.doLast {
    println " - and I'm in charge of Web Responsive User Interface."
}
```

Code Listing 98

```
/* build.gradle */
allprojects {
    task hello << {task -> println "I'm $task.project.name"}
}

subprojects.each {
   project("$it.path").hello << {
        println " - and I depend on subProjectsBehavior"
    }
}
```

The **`doLast`** method is used in every subproject build script. This method adds a closure for the **`hello`** task in each subproject, which will be executed after all previous closures assigned to the same task, in the build script of the root project. Remember, the root project is the first one evaluated in the configuration phase. Then, every subproject enters in configuration phase following the order established in the **settings.gradle** file. After that, every subproject's **build.file** will enter configuration mode—that's why the closure declared in each subproject's **build.gradle** file is added at the end of the execution chain.

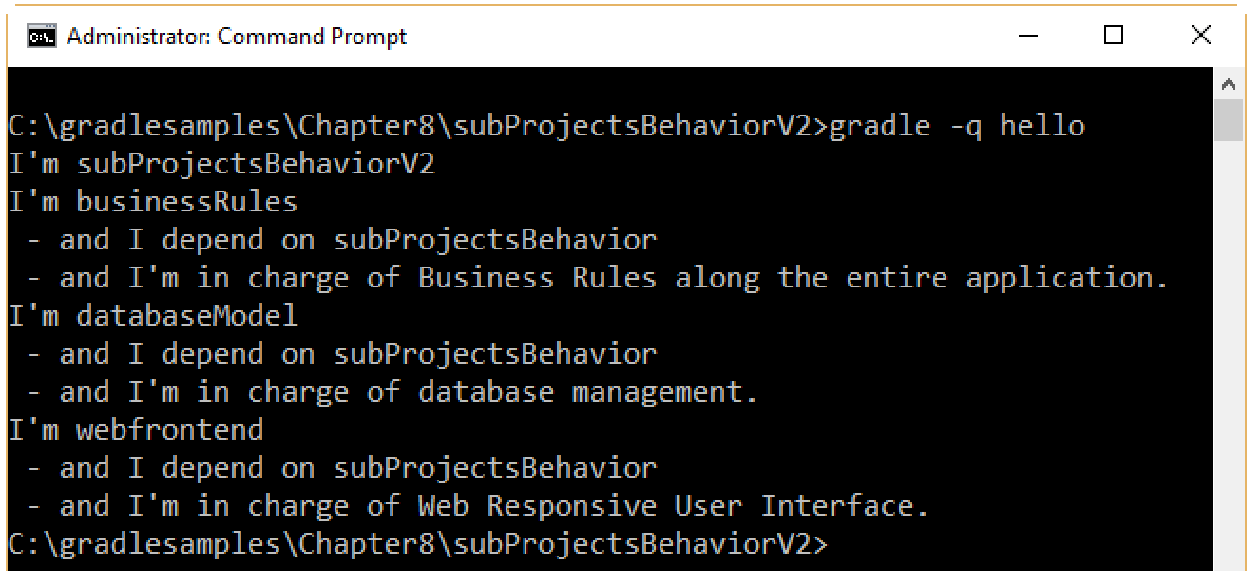
Even if the user has access to all subprojects from the root's build script, it is recommended to put project-specific behavior in a separate **build.gradle** file for each subproject. This makes project maintenance less complex.

Now, running the following command…

Code Listing 99

```
gradle -q hello
```

…the following output will be displayed.



Administrator: Command Prompt

```
C:\gradlesamples\Chapter8\subProjectsBehaviorV2>gradle -q hello
I'm subProjectsBehaviorV2
I'm businessRules
 - and I depend on subProjectsBehavior
 - and I'm in charge of Business Rules along the entire application.
I'm databaseModel
 - and I depend on subProjectsBehavior
 - and I'm in charge of database management.
I'm webfrontend
 - and I depend on subProjectsBehavior
 - and I'm in charge of Web Responsive User Interface.
C:\gradlesamples\Chapter8\subProjectsBehaviorV2>
```

Figure 43: Output for subProjectsBehaviorV2

Filtering by name

Another way to control a project's behavior is by using the **`configure`** method in the build script of the root project. This method takes a list with a series of project objects as an argument. Then, the method applies the configuration to the projects in the list.

Code Listing 100

```
allprojects {
    task hello << {task -> println "I'm $task.project.name"}
}

subprojects.each {
   project("$it.path").hello << {
         println " - and I depend on subProjectsFiltering"
    }
}

configure(subprojects.findAll {it.name != 'zipPackage'}) {
    hello << {println " - and I'm part of the application."}
}
```

This code and the project structure for this build script are derived from the sample project of the previous section. The difference consists of a new subproject called **`zipPackage`**, which has its own **build.gradle** file, too.

The **build.gradle** file for the **`zipPackage`** project is displayed in the following code snippet.

Code Listing 101

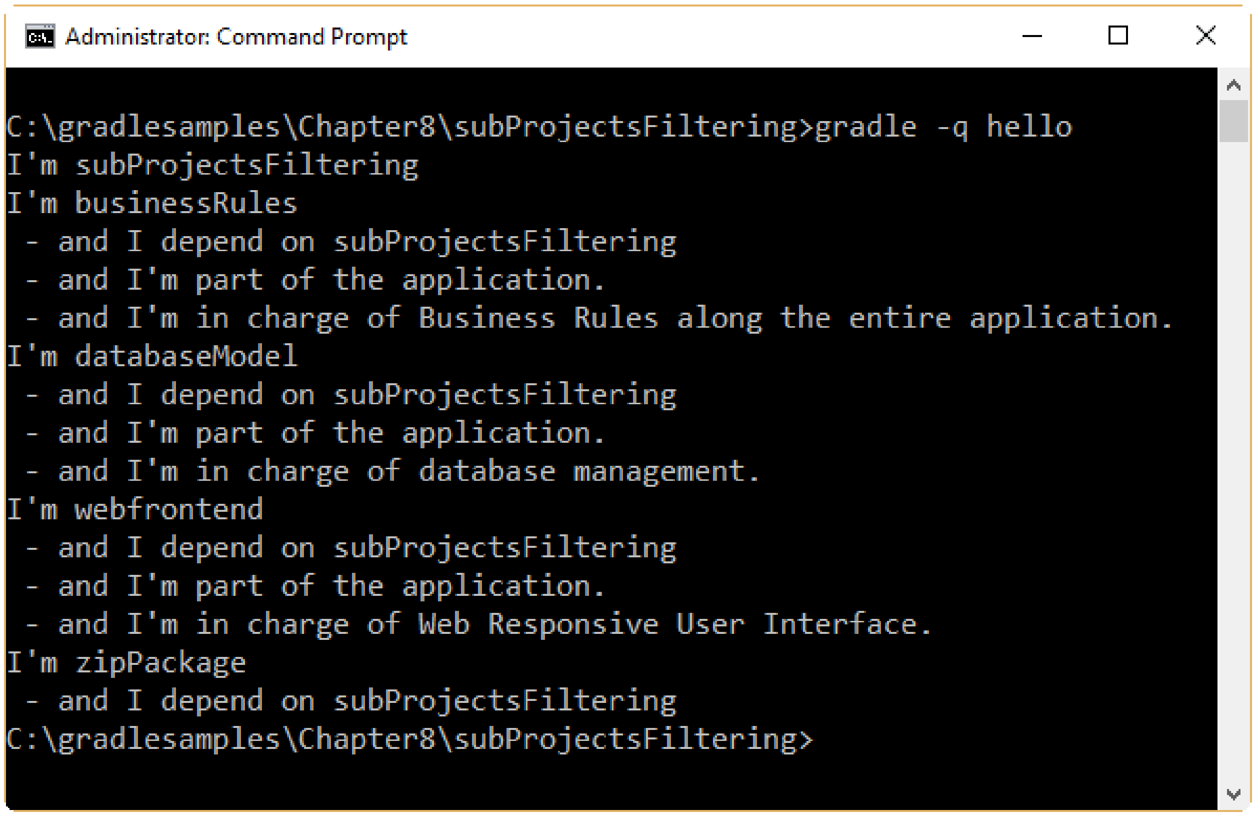
```
/* zipPackage\build.gradle */
hello.doLast {
    println " - and I'm in charge of creating a ZIP file for the 
distribution package."
}
```

Once the following command is executed…

Code Listing 102

```
gradle -q hello
```

…the output displayed is the following.



```
C:\gradlesamples\Chapter8\subProjectsFiltering>gradle -q hello
I'm subProjectsFiltering
I'm businessRules
 - and I depend on subProjectsFiltering
 - and I'm part of the application.
 - and I'm in charge of Business Rules along the entire application.
I'm databaseModel
 - and I depend on subProjectsFiltering
 - and I'm part of the application.
 - and I'm in charge of database management.
I'm webfrontend
 - and I depend on subProjectsFiltering
 - and I'm part of the application.
 - and I'm in charge of Web Responsive User Interface.
I'm zipPackage
 - and I depend on subProjectsFiltering
C:\gradlesamples\Chapter8\subProjectsFiltering>
```

Figure 44: Output for subProjectsFiltering

In Figure 44, you can see that the closure declared in **zipPackage**’s build script is not executed. Why? The **configure** method is responsible for that. Again, every subproject’s build script enters the configuration phase after the **allprojects** and **subprojects** methods are configured. But in this build script, the **configure** method has been declared. So, this method is in charge of dealing with configuration phases for all subprojects. The method receives a parameter, which is a list with all projects to be configured, and then executes the statements within the curly braces for each project in the list, before executing the statements contained in the project’s **build.gradle** file, if there are any.

The **findAll** method of the **subprojects** object creates a sublist with all the projects, which complies with the condition declared within the curly braces. In this case, the condition tells the **findAll** method that a project should be added to this sublist when it has a name different from **zipPackage**. Once the sublist is created, it is passed as a parameter to the **configure** method. Due to the condition specified to the **findAll** method, the **zipPackage** project is not included. Therefore, no configuration is applied to the project, and that includes the configuration of its own **build.gradle** file.

Filtering by properties

A filter can be established using extra project properties. To define an extra property, this one should be declared in the project’s build script with its name preceded by the keyword **ext**, followed by a dot. An extra property should look like this:

Code Listing 103

```
ext.software = true
```

In the previous code, an extra property named **software** is declared, and the value of **true** is assigned to it.

Now, this extra property should be declared in all build scripts within the multi-project directory structure (the example of the previous section is going to be used, with a few changes). The value for this property will always be **true**, except for the **zipPackage’s** subproject, in which this property will contain a **false** value.

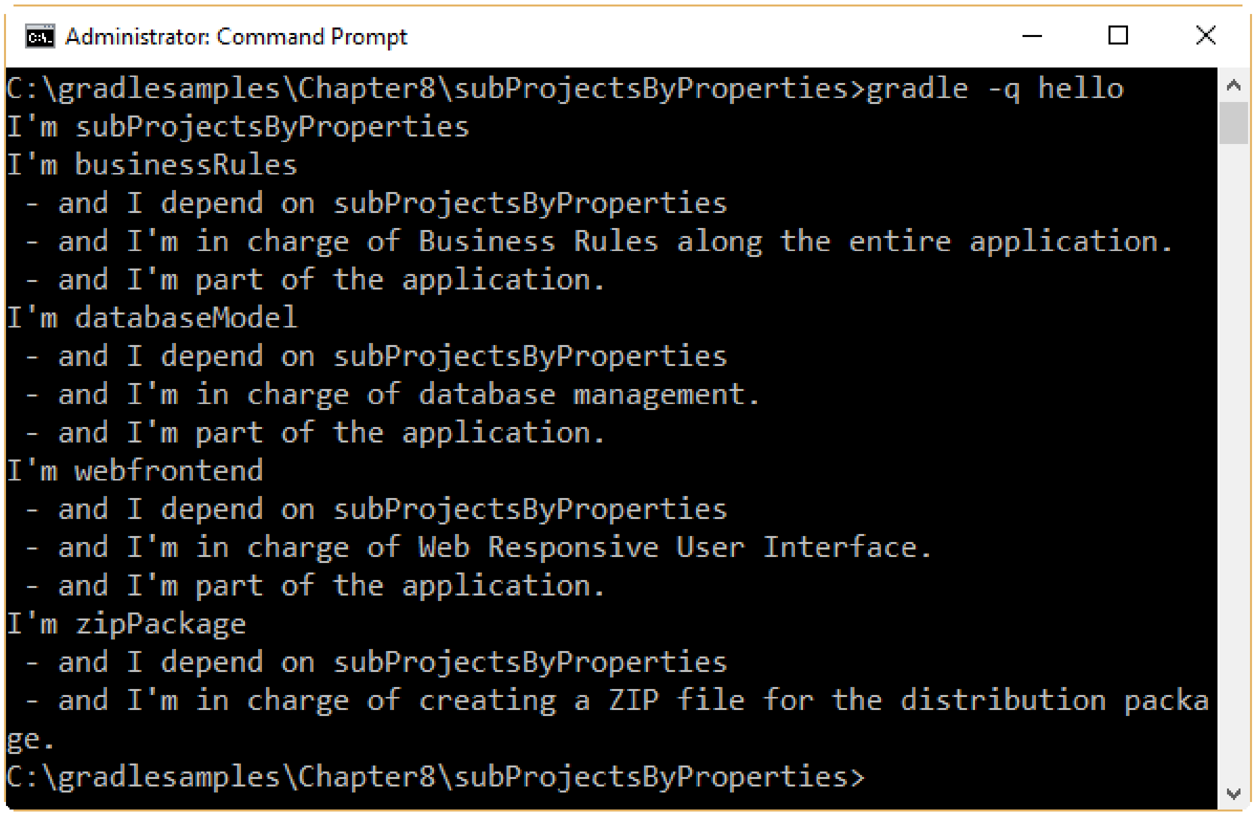
The **build.gradle** file for the root project should look like this:

Code Listing 104

```
allprojects {
    task hello << {task -> println "I'm $task.project.name"}
}

subprojects {
   hello {
           doLast {println " - and I depend on $rootProject.name"}
         }
   afterEvaluate {Project project ->
         if (project.software) { hello.doLast {println " - and I'm part of
the application."} }
```

```
    }
}
```

In the previous code, when the **`subprojects`** method enters the configuration phase, a closure is added to the **`hello`** task of each subproject, one by one. This closure displays the name of the project on which the subproject depends, using the **`name`** property of the **`rootProject`** object. After that, the **build.gradle** file for the subproject is also configured, and the **`doLast`** method adds another closure to the **`hello`** task. Then, the **`afterEvaluate`** advice action is fired for each subproject that is being configured, passing the **`subproject`** object as a parameter. This advice action evaluates the **`software`** extended property declared for every project in the directory structure, and adds a closure to the **`hello`** task if the value for that property is **`true`**. The suggestion made for the **`zipPackage`**'s build script was to set the **`software`** property's value to **`false`**. So, no closure will be added to **`zipPackage`**'s **`hello`** task by the **`afterEvaluate`** advice. Finally, Gradle enters into the execution phase.



```
C:\gradlesamples\Chapter8\subProjectsByProperties>gradle -q hello
I'm subProjectsByProperties
I'm businessRules
 - and I depend on subProjectsByProperties
 - and I'm in charge of Business Rules along the entire application.
 - and I'm part of the application.
I'm databaseModel
 - and I depend on subProjectsByProperties
 - and I'm in charge of database management.
 - and I'm part of the application.
I'm webfrontend
 - and I depend on subProjectsByProperties
 - and I'm in charge of Web Responsive User Interface.
 - and I'm part of the application.
I'm zipPackage
 - and I depend on subProjectsByProperties
 - and I'm in charge of creating a ZIP file for the distribution packa
ge.
C:\gradlesamples\Chapter8\subProjectsByProperties>
```

Figure 45: Output for subProjectsByProperties example

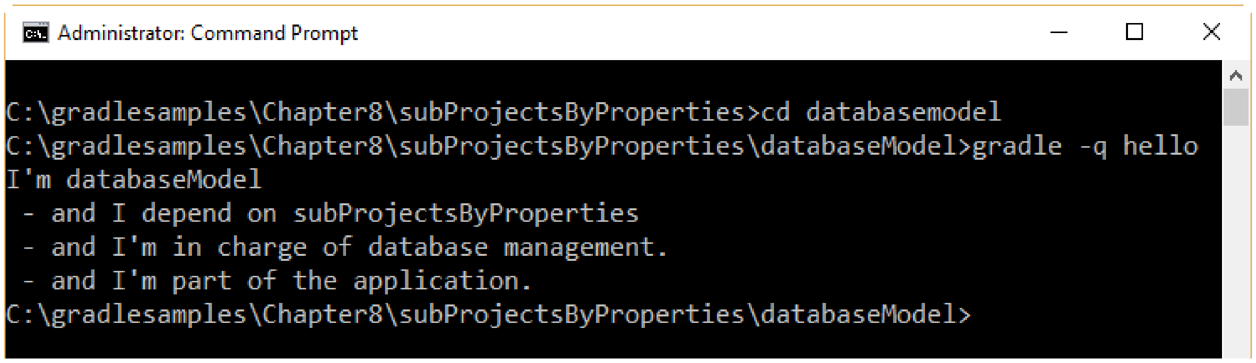
Execution rules for multi-project builds

When the **`hello`** task from the root project directory is executed, an intuitive behavior happens automatically: all **`hello`** tasks of the different projects are executed. If the user switches to the **`databaseModel`** directory and executes…

Code Listing 105

```
gradle -q hello
```

…the following output will be displayed.



```
C:\gradlesamples\Chapter8\subProjectsByProperties>cd databasemodel
C:\gradlesamples\Chapter8\subProjectsByProperties\databaseModel>gradle -q hello
I'm databaseModel
 - and I depend on subProjectsByProperties
 - and I'm in charge of database management.
 - and I'm part of the application.
C:\gradlesamples\Chapter8\subProjectsByProperties\databaseModel>
```

Figure 46: Executing hello task from databaseModel directory

There's a basic rule behind Gradle's behavior. Gradle looks down the hierarchy, starting with the current directory, for all tasks with the name hello and executes them. One thing should be noted: Gradle always evaluates every project in the multi-project build, and creates all existing task objects. Then, Gradle filters the tasks that should be executed, according to the task name argument and the current directory. Every project has to be evaluated before any task gets executed.

A real-life Java example

The following example consists of a parent project that builds two Java applications, and then creates a distribution including the two applications.

Code Listing 106

```
defaultTasks 'build', 'copyDist', 'packageDistribution'

allprojects {
    apply plugin: 'java'
    group = 'hello'
    version = '1.0'
}

project(':helloWorld').jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.HelloWorld'
    }
}

project(':sendMessage').jar {
    manifest {
        attributes 'Specification-Title'    : 'Gradle Succinctly'
        attributes 'Specification-Version'  : '1.0'
        attributes 'Specification-Vendor'   : 'Syncfusion, Inc.'
        attributes 'Implementation-Title'   : 'hello.HelloWorld'
        attributes 'Implementation-Version' : 'build02'
        attributes 'Implementation-Vendor'  : 'Syncfusion, Inc.'
        attributes 'Main-Class': 'hello.SendMessage'
    }
}

task copyDist(type: Copy) {
    into "$buildDir\\libs"
    subprojects {
        from tasks.withType(Jar)
    }
}

/* This task will be executed after the copyDist task
   and will create a zip file with the contents of
   the build\libs directory (the jar programs).

   The zip file will be saved in the
   build\distributions folder, and will be named
   distPackage-1.0.zip                             */
task packageDistribution(type: Zip){
    from 'build\\libs'
    baseName = 'distPackage'
    version = '1.0'
}
```

First, the build script declares three default tasks that will be executed in order, starting from the left. For the purposes of this example, the **`build`** task should be executed first. Then, the **`copyDist`** task will be executed after Gradle finishes building the two Java applications. This task will copy the two applications to the root project's **build\libs** folder. Finally, the **`packageDistribution`** tasks will take the contents of the root project's **build\libs** folder and create a compressed .zip file with the two applications. This file will be saved in the **build\distributions** folder of the root project.

In the previous example, the multi-project build consists of two subprojects: **`helloWorld`** and **`sendMessage`**. The **`allprojects`** method applies the Java plugin for all projects included in the build. It also assigns a version number (**`1.0`**) and a common group name for the two applications (**`hello`**). After that, the **`jar`** object is used for both projects in order to declare the application's entry point and set the package information. When the user issues the **`gradle`** command with no task name, the **`build`** task is executed first. Then, when Java applications are built, the **`copyDist`** takes all artifacts of **`Jar`** type created by the build (in this case, the two Java applications) and copies them to the root project's **build\libs** directory. Now, the `packageDistribution` task takes all files saved in the **build\libs** directory and creates a .zip compressed file in the root project's **build\distributions** directory.

Chapter summary

In Gradle, a multi-project build consists of a root project, and one or more subprojects that may also have subprojects. In this case, all the projects are linked together.

A multi-project build should follow a consistent structure, which has a main directory known as the root project's directory. Every subproject consists of a subdirectory within the root. A **settings.gradle** file and a **build.gradle** file should be in the project's root directory. The **settings.gradle** file tells Gradle how the project and subprojects are structured.

Multi-project builds are still collections of tasks Gradle can run. The difference relies on the fact that the user may want to control which project's tasks should be executed. The following options apply to a multi-project build execution:

- You can change to the directory that corresponds to the project you want to run, and then execute `gradle <taskname>` as usual.
- You can specify a qualified task name from the directory of the root project. For example, `gradle :ftputils:build` will build the **fptutils** subproject and any subprojects it depends on.

You can specify a task name that can be found in the root project, or in any subprojects it depends on, by issuing the `gradle <taskname>` command.

In a multi-project build, sometimes it is not necessary for the subprojects to have their own **build.gradle** file. That is, build scripts are optional in a multi-project build. Gradle provides in its API a property called `allprojects`, which returns a list with the current project and all its subprojects underneath it. If a closure is used when `allprojects` is called, all the statements of the closure are delegated to the projects associated with this property.

The `allprojects.each` iteration can be useful for assigning a different closure to a project, depending on its path.

There's also a way to access the subprojects only. This can be done by using the `subprojects` property. Like the `allprojects` property, `subprojects` has an `each` iteration, which allows you to access subprojects individually.

Another way to add specific behavior to a subproject is by creating a **build.gradle** file in the subproject directory. Also, a project's behavior can be controlled using the `configure` method in the build script of the root project. This method takes a list with a series of `project` objects as an argument. Then, the method applies the configuration to the projects in the list.

A filter for the subprojects can be established using extra properties. To define an extra property, it should be declared in the project's build script with its name preceded by the keyword `ext` followed by a dot (for example, `ext.software`).

When a task from the root project directory is executed, an intuitive behavior happens automatically: all tasks of the different projects with the same task name are executed. There's a basic rule behind Gradle's behavior. Gradle looks down the hierarchy, starting with the current directory, for all tasks with the same name and executes them. It should be noted that Gradle always evaluates every project in the multi-project build, and creates all existing **`task`** objects. Then, Gradle filters the tasks that should be executed according to the task **`name`** argument and the current directory. Every project has to be evaluated before any task gets executed.

Chapter 9 Running Gradle from Visual Studio Code

All code examples of this book were created using Visual Studio Code® (VS Code), a free, open-source code editor from Microsoft Corporation. Besides the Windows platform, the editor is available for Linux and Mac OS X platforms. It's extensible and customizable by installing extensions for several programming languages, adding debuggers, and connecting to additional services.

This chapter won't go deep into Visual Studio Code, as it's out of the scope of this book, but I will give a brief explanation on how Gradle builds can be executed from VS Code.

Loading a Gradle project into VS code

The first thing you should do is create the project's directory in which the build script will be edited. Then, launch VS Code.

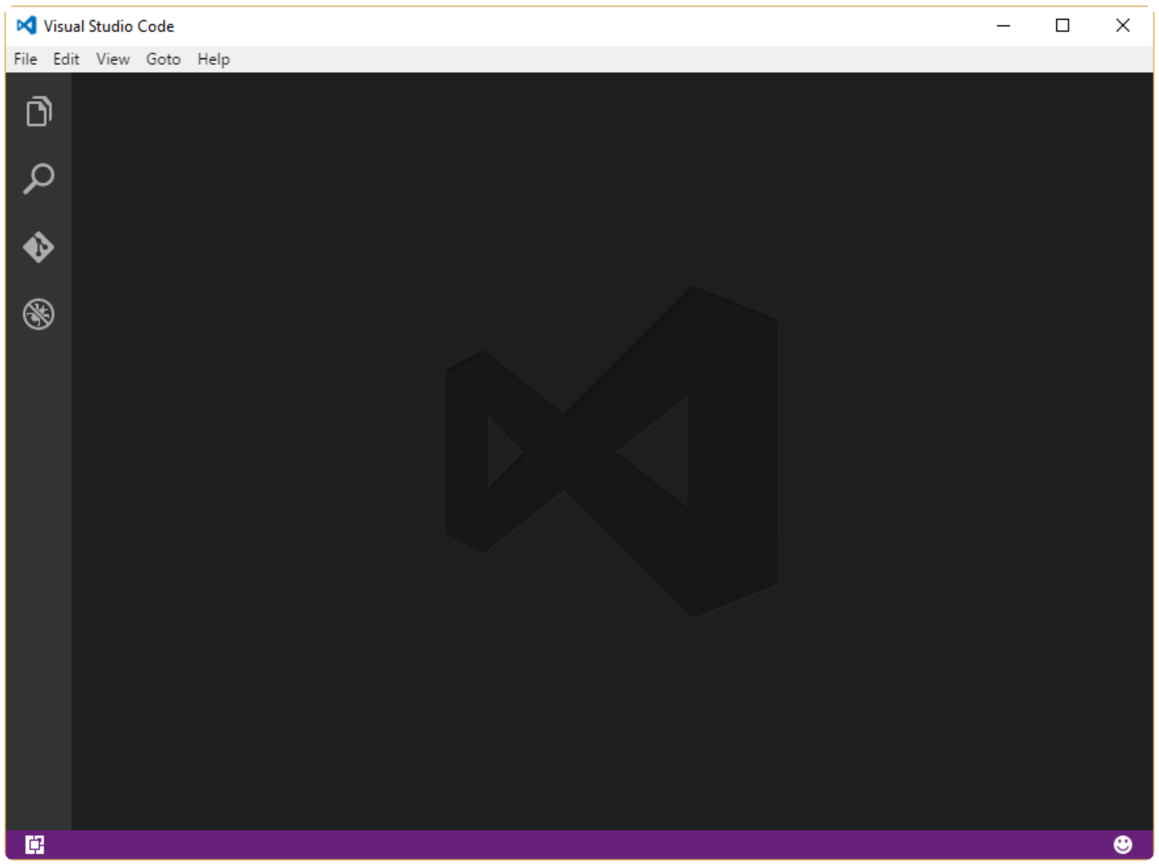


Figure 47: VS Code main window

Now, the Gradle project's directory needs to be opened by VS Code. To do so, choose the **Open Folder** option from the **File** menu.

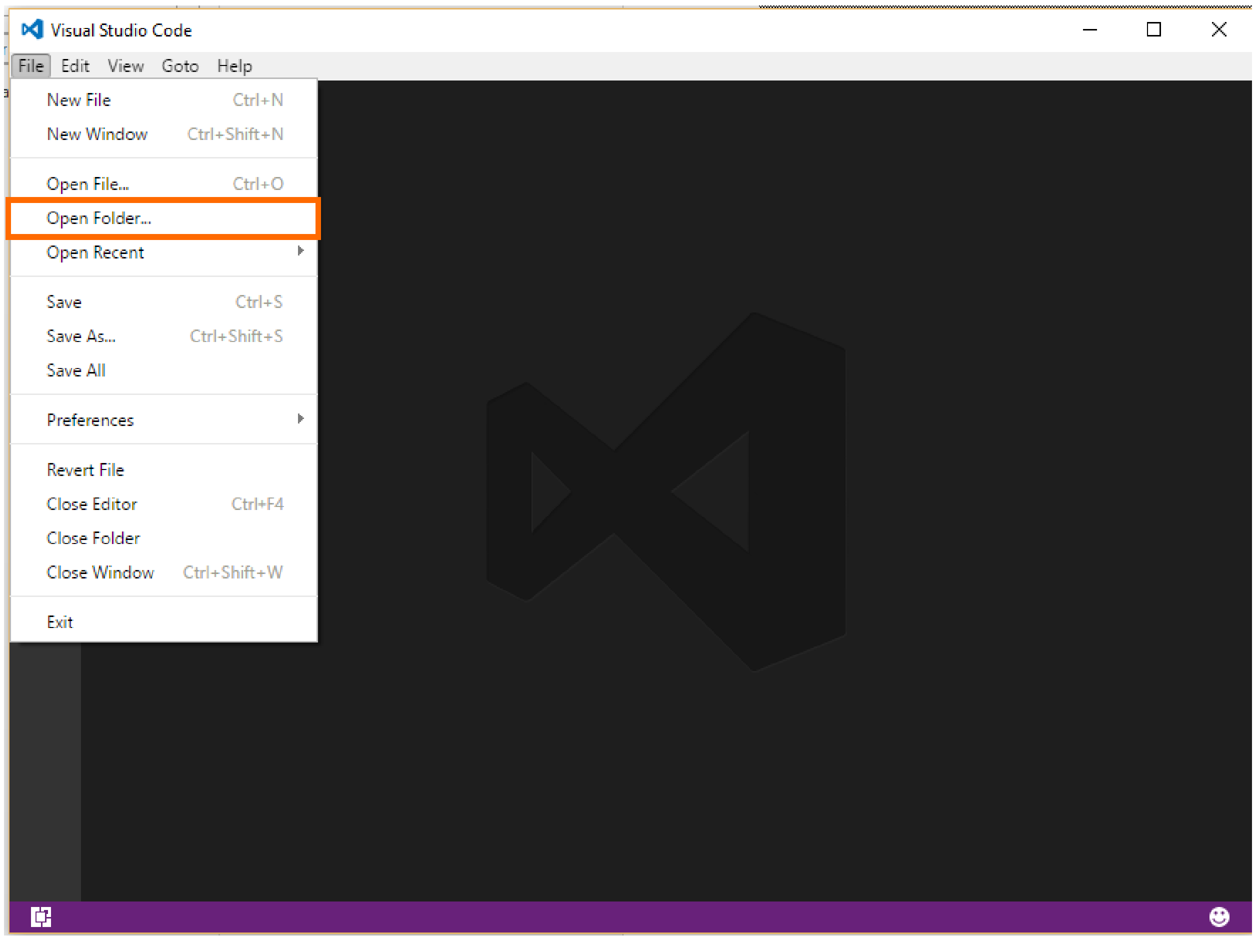


Figure 48: The Open Folder option

The directory can be located using the **Select Folder** dialog box and clicking **Select Folder**. This action will bring up the project structure to the VS Code main window.

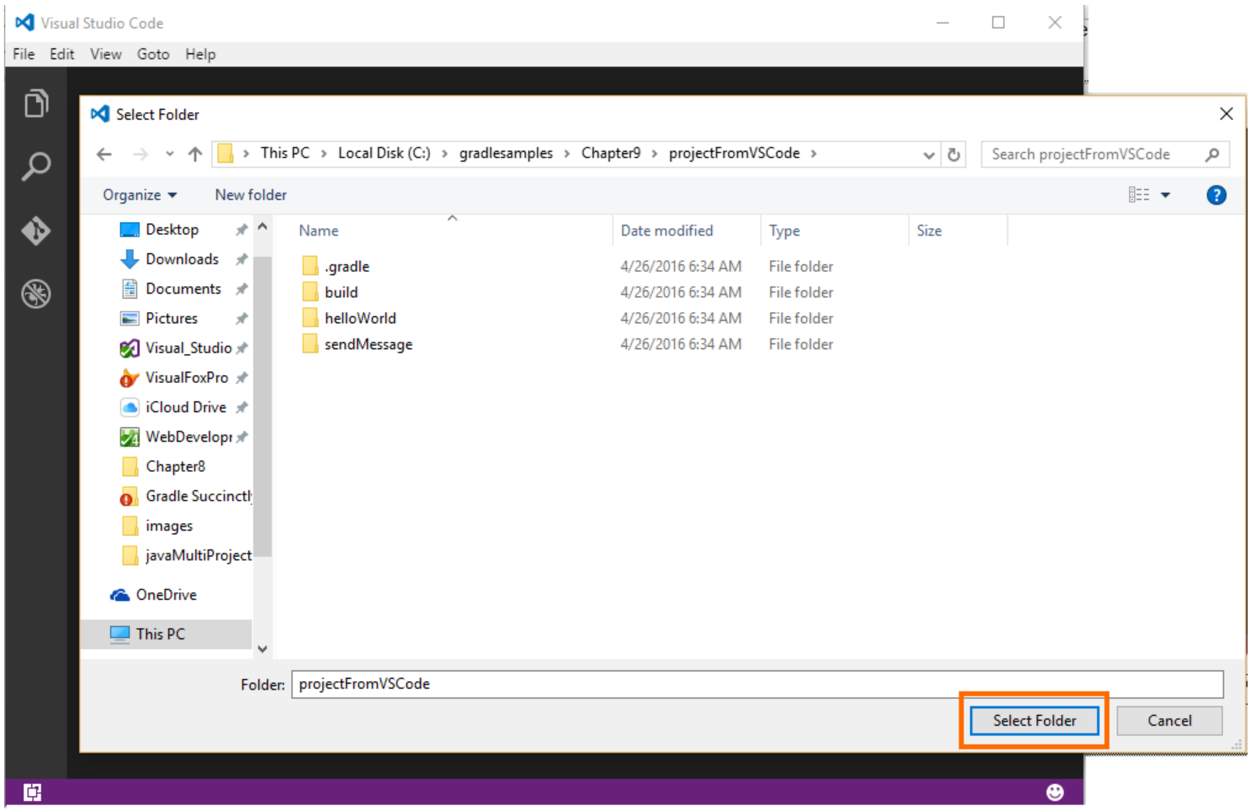


Figure 49: The Select Folder dialog box

Assuming that the build script and project's structure have been already edited, the VS Code main window will display all of the project's content at the left side. The following figure shows VS Code with the project loaded.

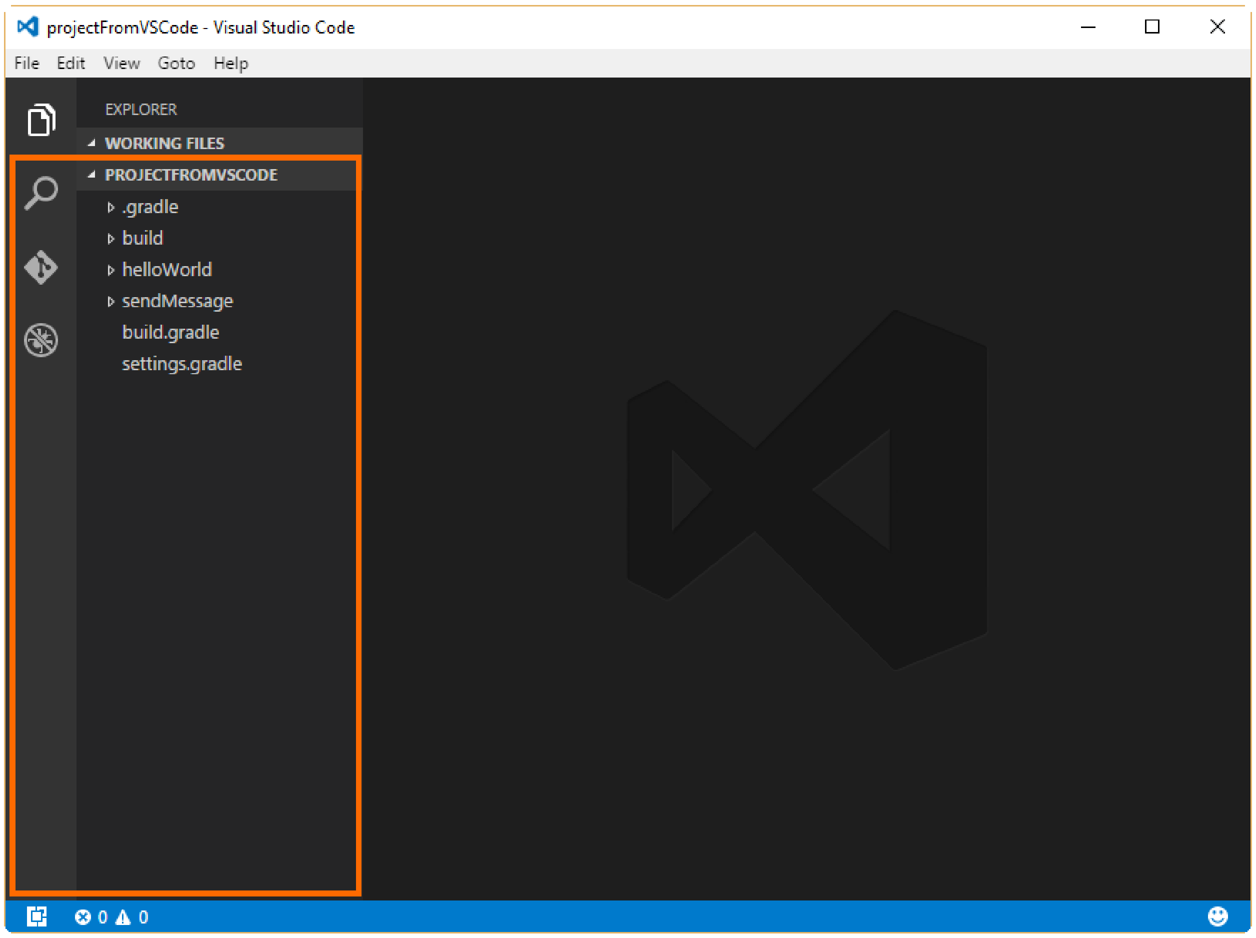


Figure 50: VS Code with the build project loaded

To edit any file contained in the project's directory, you only need to click over the name of the desired file, and its content will be displayed in the editor window, on the right side of the project's navigation bar.

Adding a task to VS Code to run Gradle

In order to run Gradle from VS Code, you need to set up a task using a **tasks.json** file. Press the **Ctrl** + **Shift** + **P** keys combination to bring up the search bar. Then, type **task** and select **Configure Task Runner** from the list.

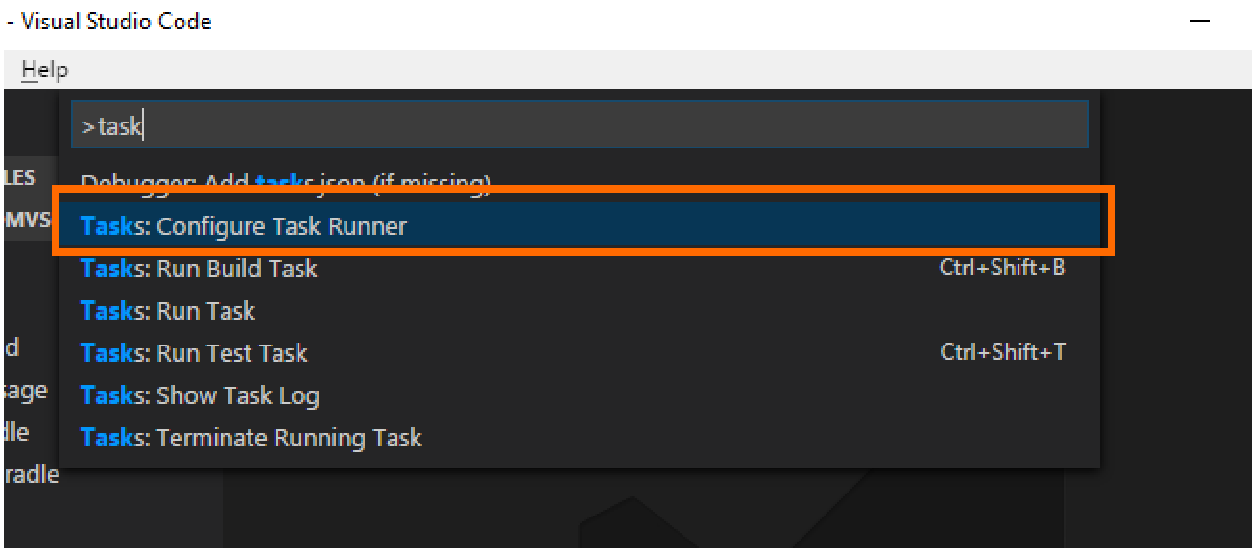

- Visual Studio Code
Help
>task
Debugger: Add tasks.json (if missing)
Tasks: Configure Task Runner
Tasks: Run Build Task Ctrl+Shift+B
Tasks: Run Task
Tasks: Run Test Task Ctrl+Shift+T
Tasks: Show Task Log
Tasks: Terminate Running Task

Figure 51: The Configure Task Runner option

After the Configure Task Runner option is selected, VS Code will ask for the type of Task Runner to be configured. Choose the **Others** option for Gradle.

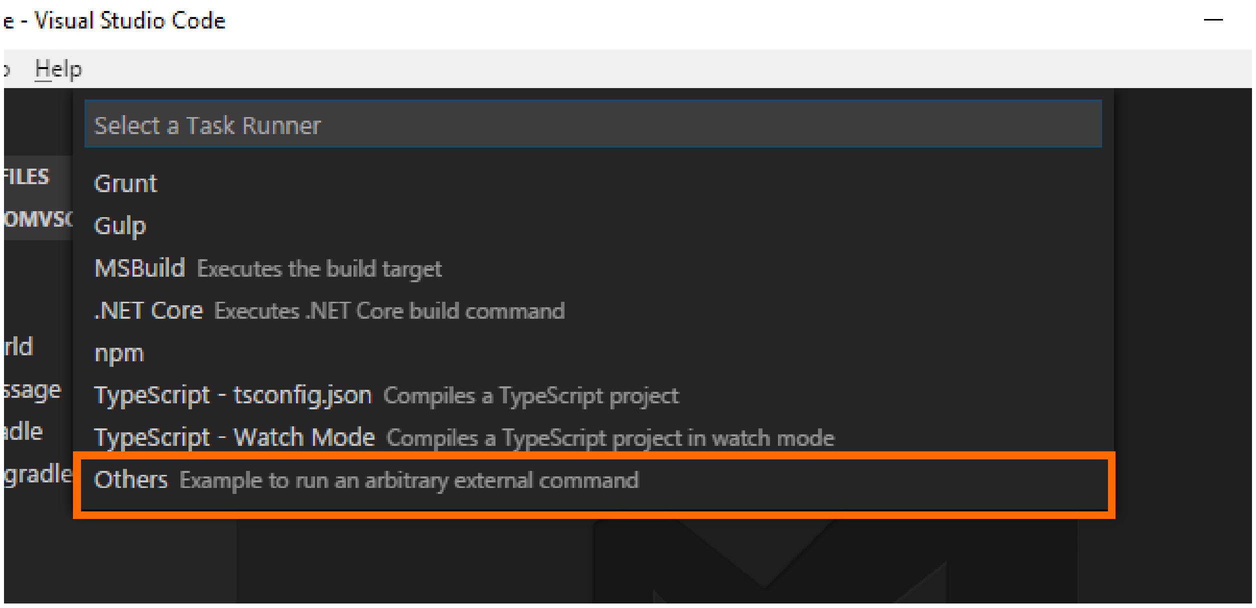
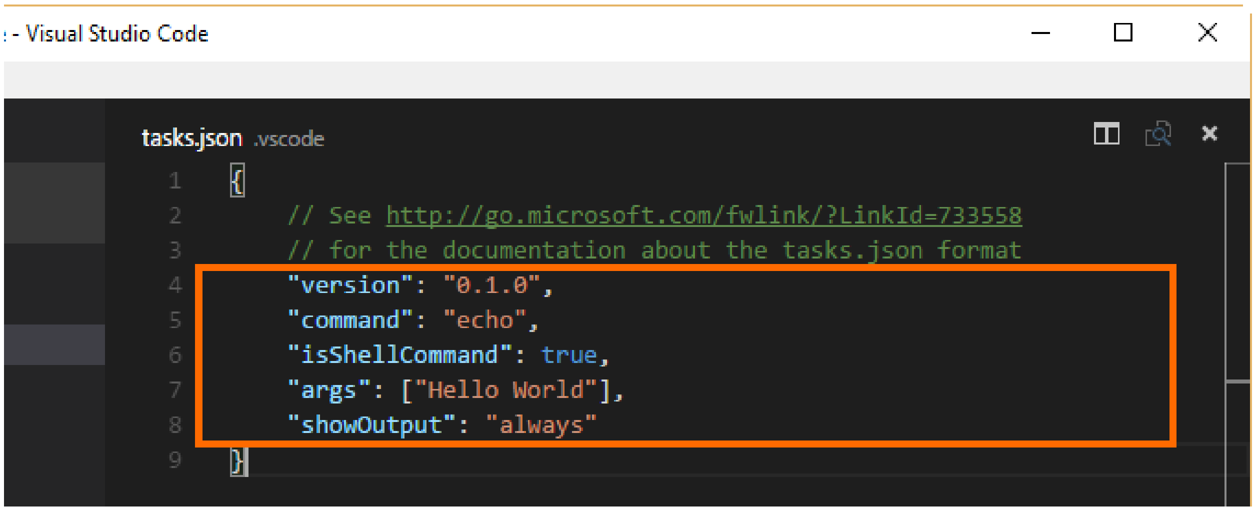

e - Visual Studio Code
Help
Select a Task Runner
Grunt
Gulp
MSBuild Executes the build target
.NET Core Executes .NET Core build command
npm
TypeScript - tsconfig.json Compiles a TypeScript project
TypeScript - Watch Mode Compiles a TypeScript project in watch mode
Others Example to run an arbitrary external command

Figure 52: The "Others" Task Runner type

A folder named **.vscode** will be created in the project's directory, and a file called **tasks.json** will be displayed in the editor window. This file contains five parameters that need to be modified to make Gradle run.

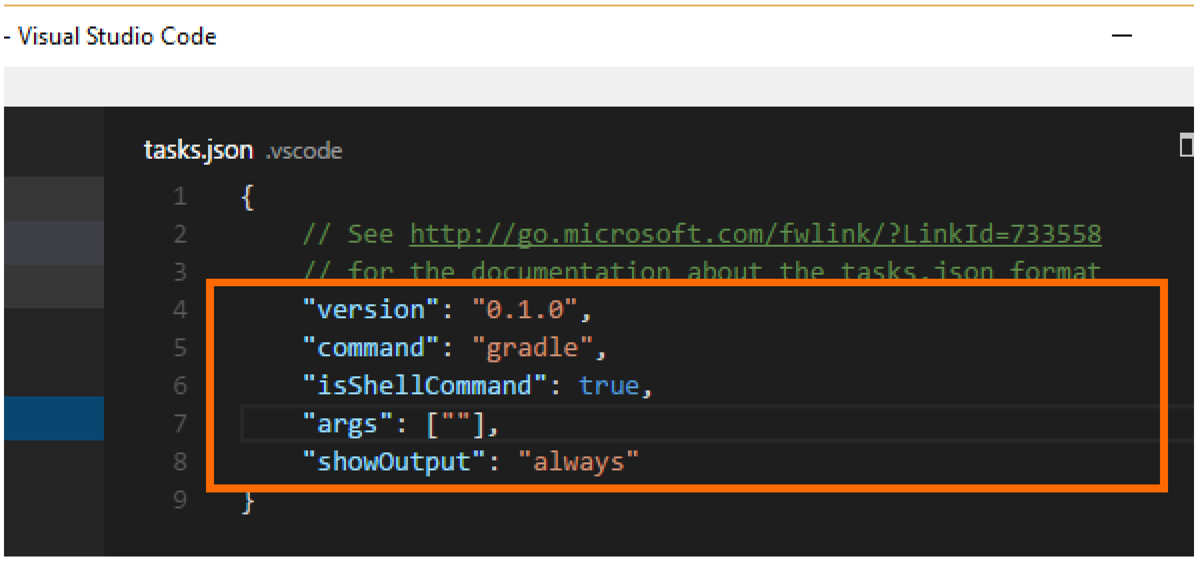


```
{
    // See http://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "0.1.0",
    "command": "echo",
    "isShellCommand": true,
    "args": ["Hello World"],
    "showOutput": "always"
}
```

Figure 53: The tasks.json file

To make Gradle run, the parameters **`command`** and **`args`** need to be edited. The **`command`** parameter's value needs to be changed to **`gradle`**, and the **`args`** (arguments) parameter can be an empty string if the build script has a default task defined, or can be the name of the task that will be executed by the build script.

After these changes, the file should look like the following figure.



```
{
    // See http://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "0.1.0",
    "command": "gradle",
    "isShellCommand": true,
    "args": [""],
    "showOutput": "always"
}
```

Figure 54: Tasks.json after changes made

Now, to run Gradle, close the file and then press the **Ctrl** + **P** keys combination to bring up the actions bar. Type **task gradle** into the actions bar and press the **Enter** key.

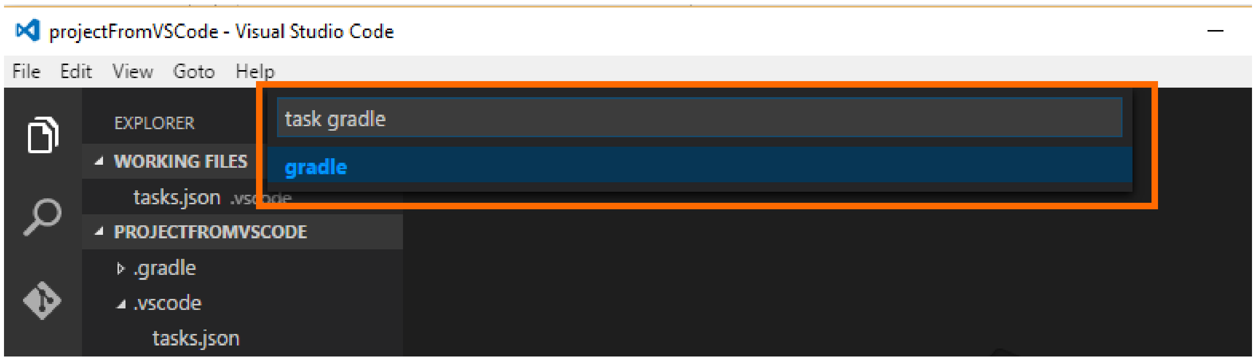
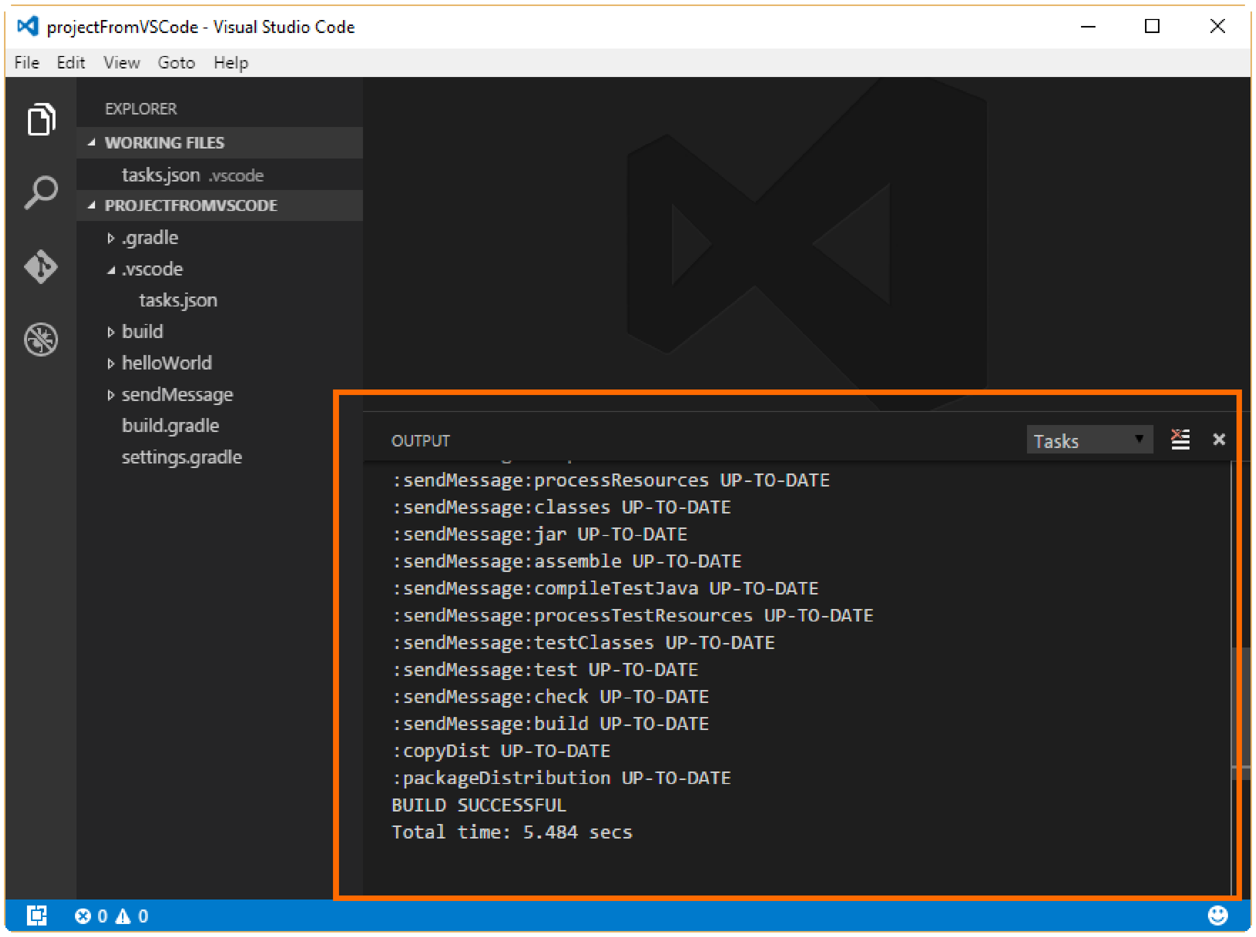


Figure 55: VS Code actions bar

After a few seconds, the output window will show the results of the build script's execution.



```
:sendMessage:processResources UP-TO-DATE
:sendMessage:classes UP-TO-DATE
:sendMessage:jar UP-TO-DATE
:sendMessage:assemble UP-TO-DATE
:sendMessage:compileTestJava UP-TO-DATE
:sendMessage:processTestResources UP-TO-DATE
:sendMessage:testClasses UP-TO-DATE
:sendMessage:test UP-TO-DATE
:sendMessage:check UP-TO-DATE
:sendMessage:build UP-TO-DATE
:copyDist UP-TO-DATE
:packageDistribution UP-TO-DATE
BUILD SUCCESSFUL
Total time: 5.484 secs
```

Figure 56: The results of the build script's execution

Chapter summary

This chapter explained how to execute build scripts with Gradle from Visual Studio Code, the free, open-source code editor from Microsoft Corporation, available for Windows, Linux, and Mac OS X platforms.

To run Gradle from VS Code, the project's folder must be opened with VS Code, and a **tasks.json** file needs to be created in that directory. This file is created with the **Configure Task Runner** option, which can be located in the search bar by typing **task**. To show this bar, press the **Ctrl** + **Shift** + **P** keys combination.

The **tasks.json** file has two parameters that need to be modified: **`command`** and **`args`**. The **`command`** parameter must be set to **`gradle`**, and the **`args`** parameter could be set to an empty string (if the build script has a default task defined), or to the name of the task that will be executed by Gradle.

To run the build script, you need to save and close the **tasks.json** file. Then, pressing the **Ctrl** + **P** keys combination will bring up the actions bar. After you type **task gradle** and press the **Enter** key, Gradle will run. The output window will show the results of the build script's execution.

General Conclusions

Gradle is a next-generation tool for build automation. Conceived upon a Groovy-based domain-specific-language (DSL), it uses convention-over-configuration principles to provide good defaults. A good example for this can be found in Java applications building, which can be accomplished with a single line of code. Its powerful DSL describes a build logic that empowers developers to create reusable and maintainable build scripts. It also supports incremental builds by intelligently determining which parts of the project are up-to-date, so that any task dependent upon those parts will not be re-executed.

Gradle has a set of rich and powerful APIs that are exposed to the developer, making available anything related with the build structure, running tasks, and their dependencies. Also, it’s possible to customize all aspects of the build behavior.

The use of the Groovy programming language within build scripts allows them to cover most of the developer’s needs, giving as a result a great tool for continuous delivery, continuous integration, and continuous testing.

Qualified by its developers as a quantum leap for building technology in the Java world, Gradle supports multiple application types and test frameworks, and a build-oriented domain model, which allows you to create concise and readable build scripts that express their intent in a simple way.

Above all, the guys of Gradle Inc., the startup which develops, distributes, and supports the Gradle open-source project, have embraced the mission “to transform how software is built and shipped.” They’re doing this by providing services related to Gradle such as build migration, performance tuning, and plugins development. They also offer training, consulting, and support for Gradle, in order to accelerate its adoption.