Get Going with Gradle

A quick-start guide for Java developers

2024 Edition

By Tom Gregory

Contents

1. Overview . . . 1
2. Introduction to Gradle . . . 3
3. Installing Gradle . . . 7
4. Gradle project structure . . . 10
5. Creating your first Gradle project . . . 12
6. How Gradle works with Java . . . 18
7. Building a Java project with Gradle . . . 25
8. Kotlin essentials for Gradle . . . 32
9. The task graph . . . 35
10. What now? . . . 44

Chapter 1. Overview

Hello and welcome to *Get Going with Gradle*. The fastest way to a working knowledge of Gradle.

Before we get started, here's a quick overview of what to expect from this book and how to make the most of it.

1.1. Who is this book for?

This book is specifically for anyone that finds themselves in one of these scenarios.

- You have no Gradle knowledge and you want to get started fast. This book will help you get there.
- You might have used Gradle for a while, but aren't confident making changes to a build.
- You know some Gradle, but want to better understand the fundamentals before learning about more advanced topics.

If any of these sound like you, then you're in the right place.

1.2. What will you learn?

By the time you reach the end of this book, you will:

- understand when to use Gradle
- have created your first Gradle project
- be able to create a simple Java project in Gradle
- understand the key aspects of a Gradle build script
- know how to write simple code in Kotlin, the Gradle scripting language

Ultimately, by the end you'll be more confident and effective working with Gradle, with a solid foundation on which you can build on in the future.

1.3. The big picture

This is a standalone book, which means as long as you put in the effort you'll achieve the above learning outcomes without any additional resources.

That said, if you already know you want to go beyond simple projects and achieve full mastery of building Java applications with Gradle, then *Gradle Build Bible* is likely the book for you. If you fall into this category, then I'll give more details at the end of this book.

1.4. Getting the most out of this book

Finally, here's some information about how you can get the most out of this book.

- Some chapters are theoretical where you just need to understand the concepts.
- Other chapters are practical, where you'll get the most benefit by actually doing the steps yourself.

You'll find full step-by-step instructions for each practical.

That's everything you need to know before beginning this book. Now it's time to Get going with Gradle.

I look forward to seeing you in the next chapter, which is an introduction to Gradle.

Chapter 2. Introduction to Gradle

In this chapter you'll get a high level overview of Gradle, including:

- What it is.
- Why you would use it.
- Why it's so powerful.
- What makes it better than other build tools.

Let's get right into it!

2.1. What is Gradle?

So first up, let's answer the question, what is Gradle?

Primarily it's a *build automation tool*. A build automation tool takes all the code in your project and packages up into a deployable unit that can be run in the target environment.

Gradle can be used to build small projects quickly, or build very complex projects that take a long time.

Gradle builds are written in Kotlin, which is a scripting language built on top of the Java Virtual Machine (JVM). You can also write Gradle build scripts using another Java JVM language called Groovy. In this book, we'll focus on the recommended Kotlin version.

In later chapters, we'll look into a few important characteristics of Kotlin that enable you to properly understand Gradle build scripts.

Finally, Gradle is highly configurable. That means that even if you have some obscure requirement in your build, you'll almost certainly find a way to do it in Gradle.

2.2. Why would you use Gradle?

Here are 4 compelling reasons why you would use Gradle:

1. Gradle makes building & running applications very easy because it's specifically designed for this purpose. It's very likely that if you try to create a build mechanism yourself for your application from scratch, you'll waste a lot of time and end up with something not as performant and streamlined as you would with Gradle.
2. There's no need for people using your projects to install Gradle. That's because the Gradle wrapper script (which we'll talk about later) comes bundled with Gradle projects. Just download a Git repository and run `./gradlew build` to build the project. It's that simple!

3. It’s very concise and less verbose than tools like Maven, because the build is defined in code and not XML.
4. It’s very performant and supports incremental builds so you don’t do the same things multiple times unnecessarily. For example, it won’t rerun the same tests if the code hasn’t changed.

2.3. What are the key Gradle concepts?

We’ll cover each of these in more detail later, but here’s a quick overview of the most important components of Gradle.

Concept 1: build script

build.gradle.kts is your Gradle build script file, where you define how your project is built

It’s written in Kotlin, and uses a Gradle domain specific language (DSL) to make defining your build as concise as possible. All this means is that it uses some special Kotlin language features we’ll cover later on to keep things neat and tidy.

If you’ve used Maven before the *build.gradle.kts* is equivalent to Maven’s *pom.xml* file

Finally, before we get into an example, the *build.gradle.kts* lives at the top level of your project

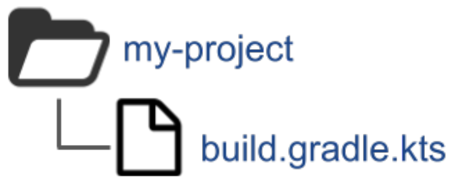


Figure 1. Build script file location

It’s time to jump into a simple Gradle build script example. Don’t worry about understanding every detail for now, because we’ll be covering this again later on.

build.gradle.kts

```
plugins { ①
    java
}

group = "org.example" ②
version = "1.0-SNAPSHOT"

repositories { ③
    mavenCentral()
}

dependencies { ④
    testImplementation("junit:junit:4.13.2")
}
```

① **Plugins** are the main way you add more functionality to your build. Here we have the *java* plugin, which adds capabilities like compiling Java code and running tests. You'll see exactly how this works later on.

② **Metadata** such as your project's *group* and *version* are important to specify. They may get used in different places in your build, like in the naming of artifacts such as Java jar files. If you're wondering where the project name is specified, that's in a different file *settings.gradle.kts*. We'll explore it later on.

③ **Repositories** is where your build's dependencies are downloaded from. A very common repository for Java projects is *Maven Central*. But Gradle also supports *Google Maven*, and you can even specify your own private repository.

④ **Dependencies** are artifacts that are required in order to build your project. During your build, Gradle downloads the dependencies from the specified repositories. Common examples include *JUnit* for running tests and *Spring Boot* for building a web application.

Concept 2: tasks

The next key concept to understand is the Gradle *task*.

A task defines a unit of work to be executed in your build. This could be anything from compiling your code to publishing it to a remote repository.

You can invoke one or many tasks from the command line. Here's an example of running the build task, which compiles and tests a project.

```
./gradlew build
```

If you're wondering what the `./gradlew` means, that's just how you invoke Gradle. You'll learn more in a moment.

You can see a list of available tasks in your project by running `./gradlew tasks`.

Of course, like most things in Gradle you can create your own custom tasks if the functionality you need doesn't already exist.

Tasks can have dependencies on other tasks. This means that the task which is depended on gets run first.

All the dependencies between tasks in a project create what is called the *Gradle task graph*.

Here's how some of the task graph looks for the *build* task.

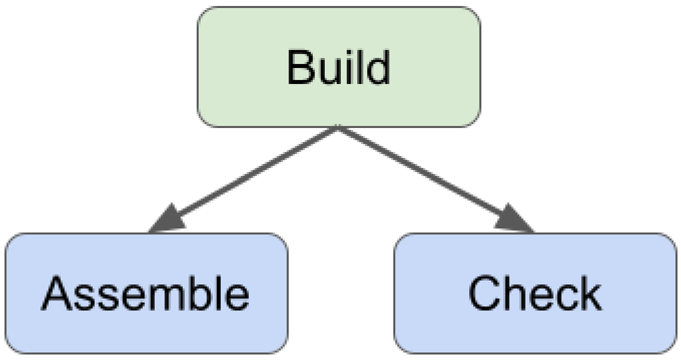


Figure 2. Task graph for the build task

The *build* task depends on the *assemble* task which packages the application and the *check* task which runs tests.

Concept 3: wrapper

The *Gradle wrapper* is a script which you use to invoke Gradle and run tasks.

In the previous section we talked about the *build* task. But how do you run the *build* task?

- In Mac or Linux environments with `./gradlew build`.
- In Windows environments with `gradlew.bat build`.

Importantly, the wrapper script is always committed into your project's version control system. This means when someone wants to build your project, they don't need to have Gradle installed on their machine. They just check out your project and build it.

The Gradle wrapper also contains a specific version of Gradle. This avoids any incompatibility problems since you know that your project's build works with the specified Gradle version.

For these reasons you should always use the Gradle wrapper script, by running `./gradlew` or `gradlew.bat`.

The only time you wouldn't use the Gradle wrapper is when you need to initialise a new Gradle project, which you can do using a local Gradle installation. We'll cover that in a practical later on.

2.4. Chapter summary

Before we complete this chapter, here are the key points to remember.

- Gradle is a modern, customisable build automation tool.
- Gradle build scripts are written in the Kotlin programming language.
- The build script is contained in a *build.gradle.kts* file.
- The build script has plugins, metadata, repositories, and dependencies.
- Tasks in Gradle represent a unit of work and are linked together in the Gradle task graph.
- You run tasks using the Gradle wrapper script, checked into version control.

Chapter 3. Installing Gradle

In this chapter, you'll install Gradle on your machine so it's ready to use in the rest of the book.

3.1. Windows Gradle Installation

Start by validating your Java installation.

- Open the Windows command prompt (open the *Start* menu, type *cmd*, then hit *Enter*).
- Validate that Java is installed.

  ```
  java -version
  ```

  It should print out details of your current Java installation. Gradle works with versions 8 and above.

Now it's time to download Gradle.

- Go to https://gradle.org/releases.
- Select the most recent Gradle release. At the time of writing this was 8.3.
- Choose the *binary-only* option. Click the link to download the Gradle zip file to your computer.

You'll now create a Gradle directory on your hard drive, and extract the zip file into that directory.

- Open *File Explorer* (open the *Start* menu, type *file*, then hit *Enter*).
- Navigate to your hard drive root directory then right click, go to *New* > *Folder*, then enter the name *Gradle*.
- Navigate to where you downloaded the Gradle zip file, copy the zip file, then paste it in the new Gradle directory.
- Right click the zip file and select *Extract all* to extract the zip file using Windows.
- You should now have an additional directory. Go into that directory, go into the *bin* directory, and copy the path from the address bar at the top of the File explorer and keep it safe for later on.

Now configure your `PATH` variable so that you can run Gradle commands from wherever you are in the command prompt

- Go to the *Start* menu, type *environment*, then hit *Enter* when *Edit the system environment variables* appears.
- On the dialog that appears, click *Environment Variables*, then under *System variables* double click the *Path* variable.
- Click *New*, then paste in the location of the Gradle *bin* directory which you copied earlier.
- Hit *Enter*, select *OK*, then *OK* again, then *OK* again.
- Close the Windows command prompt, and open a new one (*Start* menu, type *cmd*, hit *Enter*).

Now let's validate your Gradle installation:

- Run `gradle --version`

  You should see some output showing that a specific version of Gradle is installed.

If you're a Windows user, well done, that's all you need to do before moving onto the next chapter.

3.2. Linux Gradle Installation

- Validate that Java is installed:

  ```
  java -version
  ```

  It should print out details of your current Java installation. Gradle works with versions 8 and above.

- Download the latest version of Gradle using the curl command:

  ```
  curl https://downloads.gradle.org/distributions/gradle-8.3-bin.zip --output ~/gradle.zip
  ```

- Unzip the file using the unzip command:

  ```
  sudo unzip -d /opt/gradle ~/gradle.zip
  ```

  If prompted for your password, enter it because your're running this command as the root user.

- Look at the contents of the Gradle installation using the *ls* command:

  ```
  ls /opt/gradle/gradle-8.3
  ```

  You should see some files and directories.

- Setup the *PATH* environment variable:

  ```
  echo 'export PATH="$PATH:/opt/gradle/gradle-8.3/bin"' >> ~/.bashrc
  ```

- Close the terminal and open a new one.
- Show that Gradle has been successfully installed.

  ```
  gradle --version
  ```

- You should see some output showing that a specific version of Gradle is installed.

If you're a Linux user, well done, that's all you need to do before moving onto the next chapter.

3.3. Other installation options

Mac

```
brew install gradle
```

SDKMAN!

```
sdk install gradle 8.3
```

Chapter 4. Gradle project structure

In this short chapter you'll learn the basic structure of a Gradle project, in preparation for building your first Gradle project in a later chapter.

4.1. Types of Gradle project structure

Before we get into the structure of a simple project, note that Gradle does support both *single-project* and *multi-project* build structures. If you've used Maven before, you might be familiar with these concepts:

1. **A single-project structure** is normally what you use when you have a small project to build, which produces a single build output.
2. **A multi-project structure** is used for a complex project which is split into several distinct components, each of which may produce their own build outputs.

In this introductory Gradle book we'll explore the single-project structure, which is more than sufficient for many real-world use-cases.

Below is the file structure of a simple single-project Gradle build.

The structure of a simple Gradle project

```
|-- build.gradle.kts ①
|-- gradle ②
|   `-- wrapper
|       |-- gradle-wrapper.jar
|       `-- gradle-wrapper.properties
|-- gradlew ③
|-- gradlew.bat ③
|-- settings.gradle.kts ④
```

In fact, this project is so simple that the only files here are related to Gradle itself. This allows us to really understand which files are Gradle-specific, before we start adding code to our project in later chapters.

① *build.gradle.kts* is the main file where we define how our Gradle build should work, including adding plugins, metadata, repositories, and dependencies. We'll cover it again in more detail in later chapters.

② The *gradle* directory contains code and configuration for the Gradle wrapper.

③ There are the wrapper scripts themselves, including *gradlew* for Linux / Mac environments and *gradlew.bat* for Windows. Since both of these files are committed into version control, your project can automatically be built in any environment without installing Gradle separately.

④ The *settings.gradle.kts* file contains additional configuration for your project outside of the build.gradle.kts. Importantly, here you specify the project name, which gets used in various places such as for naming generated build artifacts.

4.2. Some key points

Importantly, all of the files and directories in the above project structure should be committed into version control.

There is one additional hidden directory, the *.gradle* directory, which should not be committed into version control. This is a project specific cache used internally by Gradle. We don’t need to know any more details other than that it should be added to your *.gitignore* file and not committed to version control.

Fortunately, when you use Gradle to generate your project it automatically generates the correct *.gitignore* file, but it’s important to know what is and isn’t stored in version control.

NOTE A *.gitignore* file contains a list of files and directories that are ignored by Git and never committed to version control.

That’s everything you need to know about the Gradle project structure. The next chapter is a practical where you’ll use the Gradle installation from earlier to generate your first Gradle project.

Chapter 5. Creating your first Gradle project

In this chapter, you're finally going to create a Gradle project and see with your own eyes why it's such an amazingly useful tool.

Let's get right into it!

5.1. What we'll be doing

This is another practical, where you should complete the provided steps yourself.

So what will we be doing?

1. We'll use the Gradle installation you setup earlier to run the `gradle init` command. This bootstraps a brand new Gradle project. The time to use `gradle init` is when you're starting a new project or working with an existing project which doesn't use Gradle yet, and you want to automatically generate all the necessary Gradle build files.
2. The `gradle init` command comes with a setup wizard, which we'll run through to understand the different options.
3. Once the project has been generated, we'll step through each of the new files and directories.
4. Just like with a real project, we'll commit the project into version control using Git.

Whether you're using a Linux/Mac or Windows environment, you'll find the step-by-step instructions below.

5.2. Setting up your Gradle project

Open up a terminal and navigate to your home directory.

Create a directory for the project:

```
mkdir get-going-with-gradle
```

Navigate into the directory:

```
cd get-going-with-gradle
```

Create a Gradle project using the setup wizard:

```
gradle init
```

Select type of project to generate as *basic*:

```
1
```

Select build script DSL as *Kotlin*:

```
1
```

Choose the default project name:

`<enter>`

For *Generate build using new APIs and behavior*, choose the default *no*:

`<enter>`

Here's how the console output looks when you follow the previous steps.

```
$ gradle init
Starting a Gradle Daemon (subsequent builds will be faster)

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 1

Select build script DSL:
  1: Kotlin
  2: Groovy
Enter selection (default: Kotlin) [1..2] 1

Project name (default: get-going-with-gradle):

Generate build using new APIs and behavior (some features may change in the next minor
release)? (default: no) [yes, no]


> Task :init
To learn more about Gradle by exploring our Samples at
https://docs.gradle.org/8.3/samples

BUILD SUCCESSFUL in 19s
2 actionable tasks: 2 executed
```

5.3. Interact with the new project

Try interacting with the project by running the *help* task:

`./gradlew help` (Linux/Mac)

`gradlew.bat help` (Windows)

```
$ ./gradlew help

> Task :help

Welcome to Gradle 8.3.
```

```
To run a build, run gradlew <task> ...

To see a list of available tasks, run gradlew tasks

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

To see a list of command-line options, run gradlew --help

For more detail on using Gradle, see
https://docs.gradle.org/8.3/userguide/command_line_interface.html

For troubleshooting, visit https://help.gradle.org

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

List all available tasks with the *tasks* task:

`./gradlew tasks` (Linux/Mac)

`gradlew.bat tasks` (Windows)

```
$ ./gradlew tasks

> Task :tasks

------------------------------------------------------------
Tasks runnable from root project 'get-going-with-gradle'
------------------------------------------------------------

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

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root project
'get-going-with-gradle'.
dependencies - Displays all dependencies declared in root project 'get-going-with-
gradle'.
dependencyInsight - Displays the insight into a specific dependency in root project
'get-going-with-gradle'.
help - Displays a help message.
javaToolchains - Displays the detected java toolchains.
kotlinDslAccessorsReport - Prints the Kotlin code for accessing the currently
available project extensions and conventions.
outgoingVariants - Displays the outgoing variants of root project 'get-going-with-
gradle'.
projects - Displays the sub-projects of root project 'get-going-with-gradle'.
```

```
properties - Displays the properties of root project 'get-going-with-gradle'.
resolvableConfigurations - Displays the configurations that can be resolved in root
project 'get-going-with-gradle'.
tasks - Displays the tasks runnable from root project 'get-going-with-gradle'.

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

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

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

Look at the different files in the project, excluding hidden files:

`ls -l` (Linux/Mac)

`dir` (Windows)

```
$ ls -l
total 18
-rw-r--r-- 1 Tom 197121  207 Sep 27 09:37 build.gradle.kts
drwxr-xr-x 1 Tom 197121    0 Sep 27 09:37 gradle
-rwxr-xr-x 1 Tom 197121 8639 Sep 27 09:37 gradlew
-rw-r--r-- 1 Tom 197121 2868 Sep 27 09:37 gradlew.bat
-rw-r--r-- 1 Tom 197121  377 Sep 27 09:37 settings.gradle.kts
```

5.4. Explore the project files

Look inside *build.gradle.kts*:

`cat build.gradle.kts` (Linux/Mac)

`type build.gradle.kts` (Windows)

```
$ cat build.gradle.kts
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This is a general purpose Gradle build.
 * To learn more about Gradle by exploring our Samples at
https://docs.gradle.org/8.3/samples
 */
```

Look inside *settings.gradle.kts*:

`cat settings.gradle.kts` (Linux/Mac)

`type settings.gradle.kts` (Windows)

```
$ cat settings.gradle.kts
/*
 * This file was generated by the Gradle 'init' task.
 *
 * The settings file is used to specify which projects to include in your build.
 * For more detailed information on multi-project builds, please refer to
https://docs.gradle.org/8.3/userguide/building_swift_projects.html in the Gradle
documentation.
 */

rootProject.name = "get-going-with-gradle"
```

Look at any hidden files or directories:

`ls -la` (Linux/Mac)

`dir/a` (Windows)

```
$ ls -la
total 124
drwxr-xr-x 1 Tom 197121    0 Sep 27 09:37 .
drwxr-xr-x 1 Tom 197121    0 Sep 27 09:35 ..
-rw-r--r-- 1 Tom 197121  223 Sep 27 09:37 .gitattributes
-rw-r--r-- 1 Tom 197121  108 Sep 27 09:37 .gitignore
drwxr-xr-x 1 Tom 197121    0 Sep 27 09:43 .gradle
-rw-r--r-- 1 Tom 197121  207 Sep 27 09:37 build.gradle.kts
drwxr-xr-x 1 Tom 197121    0 Sep 27 09:37 gradle
-rwxr-xr-x 1 Tom 197121 8639 Sep 27 09:37 gradlew
-rw-r--r-- 1 Tom 197121 2868 Sep 27 09:37 gradlew.bat
-rw-r--r-- 1 Tom 197121  377 Sep 27 09:37 settings.gradle.kts
```

Look inside the *.gitignore* file:

`cat .gitignore` (Linux/Mac)

`type .gitignore` (Windows)

```
$ cat .gitignore
# Ignore Gradle project-specific cache directory
.gradle

# Ignore Gradle build output directory
build
```

5.5. Commit the project into version control

Initialise this directory as a Git repository:

```
$ git init
Initialized empty Git repository in C:/workspace/get-going-with-gradle/.git/
```

Get all the files ready to be committed:

```
$ git add .
```

See what files are staged for commit:

```
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitattributes
        new file:   .gitignore
        new file:   build.gradle.kts
        new file:   gradle/wrapper/gradle-wrapper.jar
        new file:   gradle/wrapper/gradle-wrapper.properties
        new file:   gradlew
        new file:   gradlew.bat
        new file:   settings.gradle.kts
```

Commit everything:

```
$ git commit -m "Initialise project"
[main (root-commit) ffee363] Initialise project
 8 files changed, 376 insertions(+)
 create mode 100644 .gitattributes
 create mode 100644 .gitignore
 create mode 100644 build.gradle.kts
 create mode 100644 gradle/wrapper/gradle-wrapper.jar
 create mode 100644 gradle/wrapper/gradle-wrapper.properties
 create mode 100644 gradlew
 create mode 100644 gradlew.bat
 create mode 100644 settings.gradle.kts
```

Good work! You’ve just created your first Gradle project and committed it into version control.

Chapter 6. How Gradle works with Java

In this chapter you'll learn one of the most common use cases for Gradle, which is to create a Java project.

6.1. Java project requirements

Let's first put Gradle aside, and consider the most important requirements for building a Java project. You'll then see how Gradle handles each of these requirements.

1. **Compile classes:** in a Java project, we have Java classes - files with the *.java* extension, - that need to be compiled into executable *.class* files. These *.class* files are then executed in our deployment environment to run our application. So our first requirement is to compile Java classes into *.class* files
2. **Manage resources:** there may be additional resources other than Java classes that need to go into our target environment. This could be text files, images, or anything else which needs to live alongside the code. All of this needs to be properly managed.
3. **Package:** we need a mechanism to easily package the compiled classes and resources into a Java *jar* file. A *jar* file is just a *zip* file in a specific format that Java recognises, and it's a lot easier to manage than handling directories and files.
4. **Run tests:** we'd like a mechanism to easily run tests against our code. Importantly though, the tests shouldn't be packaged into the final build artifact as they're not required to execute the application.
5. **Define dependencies:** it would be great to have a way to manage dependencies in our Java project. Rather than writing code from scratch in every project, we often pull in other libraries to help do the heavy lifting. Some popular Java libraries include *Apache Commons* and *Spring Boot*. Having a way to define all these dependencies with their versions would be super-helpful.

Of course there may be other requirements with more complex projects, but this list provides a good starting point.

So how can Gradle help with this?

6.2. Gradle Java plugin

It turns out that it's extremely simple to setup a Java project with Gradle. All you have to do is apply the Java plugin, which adds all the functionality you need to meet the requirements we just covered.

Remember in the *Introduction to Gradle* chapter when we were talking about plugins in the *build.gradle.kts* file? Well, to apply the Java plugin all you have to do is add an entry into the plugins section of *build.gradle.kts* like this.

```
plugins {
    java
}
```

With this plugin applied, your Gradle project gets configured in a certain way and some additional tasks become available. That’s what plugins do, they add new functionality to your project.

The Java plugin is one of the core Gradle plugins. That means it’s important enough that it’s bundled up with the Gradle distribution, and maintained by the Gradle team.

In the rest of this chapter we’ll look at what functionality the Java plugin brings to a project, and what specific structure it expects a project to have.

6.3. Java plugin: compiling classes

Our first requirement for a Java project was to compile classes.

The Java plugin helps us to do this through a task called *compileJava*. You can run this like you would any Gradle task with `./gradlew compileJava`.

```
$ ./gradlew compileJava

BUILD SUCCESSFUL in 4s
1 actionable task: 1 executed
```

This task uses whatever Java installation is setup in your environment to compile your project’s *.java* files into *.class* files. These *.class* files get output into the Gradle *build* directory, which is a directory where any generated build outputs from Gradle get created.

Here’s an example with a Java class *MyFirstClass* in a *com.gradlehero* package.

```
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- gradlehero
    |   |           `-- MyFirstClass.java
```

When we run the *compileJava* task, the *.class* file gets output into the *build/classes* directory with a similar directory structure.

```
|-- build
|   |-- classes
|   |   `-- java
|   |       `-- main
|   |           `-- com
|   |               `-- gradlehero
|   |                   `-- MyFirstClass.class
```

Don't worry about the details of the directories for now, because I'll explain the structure in full later in this chapter. What's important to remember is just that the compiled classes go into the *build* directory.

6.4. Java plugin: managing resources

The next requirement for a Java project was to manage resources.

You guessed it, the Java plugin lets us do this through another task. The task is called *processResources*.

You run this task using `./gradlew processResources`.

```
$ ./gradlew processResources

BUILD SUCCESSFUL in 1s
```

The task looks in specific directories in your project which have been marked as resources directories, and copies the contents into the *build* directory.

The reason it's called *processResources* is that it can do some additional processing along the way, such as finding and replacing strings.

Here's an example with a text file called *super-important.txt* in a *resources* directory.

```
`-- src
    |-- main
    |   `-- resources
    |       `-- super-important.txt
```

When we run the *processResources* task, the text file gets copied into the *build* directory with a similar directory structure.

```
|-- build
|   |-- resources
|   |   `-- main
|   |       `-- super-important.txt
```

I'll go over the directory structure in more detail later, but just know for now that *processResources* copies resources into the *build* directory

6.5. Java plugin: package into jar file

The next requirement for a Java project was to package our compiled classes and resources into a *jar* file.

Unsurprisingly there's another Gradle task for this. Can you guess what it's called?

That's right, the Java plugin adds a *jar* task which you run using `./gradlew jar`.

```
$ ./gradlew jar

BUILD SUCCESSFUL in 1s
3 actionable tasks: 1 executed, 2 up-to-date
```

The *jar* task takes all the compiled classes and resources from the *build* directory, and adds them to a *jar* file.

The name of *jar* file is `<project name>-<version>.jar`. It uses the build metadata we saw earlier, defined in the *build.gradle.kts* and *settings.gradle.kts* files.

Here's an example of a project with classes and resources.

```
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- gradlehero
    |   |           `-- MyFirstClass.java
    |   `-- resources
    |       `-- super-important.txt
```

The *jar* task packages them into a jar file in the *build/libs* directory.

```
|-- build
|   |-- libs
|   |   `-- get-going-with-gradle-1.0-SNAPSHOT.jar
```

You can try building a *jar* file yourself in the next chapter, which is a practical all about creating your first Java project with Gradle.

6.6. Java plugin: easily run tests

The next requirement for a Java project was to easily run tests.

That's achieved with a task called test, which you run using `./gradlew test`.

```
$ ./gradlew test

BUILD SUCCESSFUL in 3s
4 actionable tasks: 2 executed, 2 up-to-date
```

What this does is compile your test code, process any test resources, then run the tests.

The other nice thing it does is produce a pretty test report in the *build* directory.

```
Test Summary
1
tests

0
failures

0
ignored

0.022s
duration

100%
successful
```

The test report tells you what's passed and failed. In this case, it's a 100% success rate.

6.7. Java plugin: define dependencies

The final requirement we had for a Java project was to easily define dependencies.

The way dependencies are managed in Gradle is in the *dependencies* section of *build.gradle.kts*, which we discussed briefly earlier.

As a reminder, a dependency definition looks like this.

```
dependencies {
    implementation("org.apache.commons:commons-lang3:3.13.0")
    testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
}
```

Here we have two dependencies defined. One for *Apache Commons Lang 3*, and one of *JUnit 5*.

When you define dependencies, define the *group*, *name*, and *version*, separated by a colon. This gives Gradle enough information to be able to find the dependency in whatever repositories you've configured.

When you declare the dependency, it goes into a specific dependency configuration:

- **implementation** is for dependencies required during compilation and execution of your code.
- **testImplementation** is similar, but is for dependencies required during compilation and execution of your tests.

It’s important you pick the correct dependency configuration, because they’re used to generate the Java classpath.

The classpath is used by Java so it knows about all the classes required during code compilation or execution. When the *compileJava* Gradle task uses Java to compile your classes, the classpath must include all the relevant dependencies.

6.8. Gradle Java project layout

In this chapter, we’ve talked about how the Java plugin takes classes and resources, processes them in some way, and outputs generated files into the *build* directory.

The last topic to cover is the default project layout that the Java plugin expects. Using this standard layout means Gradle knows where to find things.

- **src/main/java** for classes
- **src/main/resources** for resources
- **src/test/java** for test classes
- **src/test/resources** for test resources

As is normally the case with Gradle, these locations are configurable. It’s always best to go with the default layout because it’s the standard people expect. It’s also the same layout that the popular Maven build tool uses.

Let’s see where the classes or resources defined above end up in the build directory. Remember the *build* directory contains all the Gradle build outputs, and importantly doesn’t get committed into version control.

- **src/main/java** classes go into **build/classes/java/main**
- **src/main/resources** resources go into **build/resources/main**
- **src/test/java** test classes go into **build/classes/java/test**
- **src/test/resources** test resources go into **build/resources/test**

You don’t need to remember all these destination details, but it should help you navigate your way around the build directory when you need to.

6.9. Chapter summary

In summary:

- Applying the Java plugin initialises a project as a Java project.
- The Java plugin adds tasks for *compileJava*, *processResources*, *jar*, and *test*.
- Dependencies can be defined as *implementation* or *testImplementation*, and they end up on the generated Java classpaths.
- Always use the standard project layout so the Java plugin knows where to find files.

This has been very theoretical up to this point, which is why in the next chapter you'll jump into a practical to create your first Java project with Gradle.

Chapter 7. Building a Java project with Gradle

At this point in the book, you have practical experience of creating a basic Gradle project and running tasks. You're also familiar with the Gradle Java plugin, and the ways in which it configures a project.

Now it's time to solidify all that knowledge by putting it into practice.

7.1. What you'll be doing

You'll create a Gradle build for a simple Java application. If you don't have any knowledge of Java coding that's not a problem, because you'll have access to all the code you need. You just need to understand at a high level what the application does and how it's built.

This application is very simple. It will take as an input a language code, such as *EN* for English or *ES* for Spanish.

It will output the word *Hello* in that language, assuming it holds a translation for it.

Here are some example inputs and outputs:

- input *EN* for English ⇒ the application prints *Hello*
- input *ES* for Spanish ⇒ the application prints *Hola*

I know, it's amazing stuff. This could seriously be a competitor for *Google Translate* don't you think?

You'll build on top of the project you created in the previous practical *Creating your first Gradle project*. If you haven't created that project, then please go and complete that practical so you're in the right position to begin.

As usual, you'll find step-by-step instructions and full code snippets below.

7.2. Setting up your Gradle project

Open the project you built in the *Creating your first Gradle project* chapter. Use a command-line, text editor, or IDE as you prefer.

7.3. Adding the Java code

Add a directory *src/main/java* for the main application code.

Within the new directory, create directories for the Java package *com/gradlehero/languageapp*.

Within the new package, create a file *SayHello.java* and open it for editing.

Paste in the following Java code:

```
package com.gradlehero.languageapp;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;

public class SayHello {
    public static void main(String[] args) throws IOException {
        String language = args[0];

        InputStream resourceStream = SayHello.class.getClassLoader
().getResourceAsStream(language + ".txt");
        assert resourceStream != null;
        BufferedReader bufferedInputStream = new BufferedReader(new InputStreamReader
(resourceStream, StandardCharsets.UTF_8));

        System.out.println(bufferedInputStream.readLine());
    }
}
```

7.4. Adding the resources

Create a new directory under *src/main* called *resources*.

Add a file *en.txt* with contents *Hello!*.

Add a file *es.txt* with contents *Hola!*.

7.5. Building the application

Open *build.gradle.kts* and delete the comment.

Insert this plugin configuration:

```
plugins {
    java
}
```

See what Gradle tasks we have available:

`./gradlew tasks` (Linux/Mac)

`gradlew.bat tasks` (Windows)

All the tasks listed under *Build tasks* have been added to the project by the Java plugin.

```
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 classes of the 'main' feature.
testClasses - Assembles test classes.
```

To see all Gradle tasks, including *compileJava* and *processResources* run:

`./gradlew tasks --all` (Linux/Mac)

`gradlew.bat tasks --all` (Windows)

These tasks appear under *Other tasks*.

```
Other tasks
-----------
compileJava - Compiles main Java source.
compileTestJava - Compiles test Java source.
components - Displays the components produced by root project 'get-going-with-gradle-
practical'. [deprecated]
dependentComponents - Displays the dependent components of components in root project
'get-going-with-gradle-practical'. [deprecated]
model - Displays the configuration model of root project 'get-going-with-gradle-
practical'. [deprecated]
prepareKotlinBuildScriptModel
processResources - Processes main resources.
processTestResources - Processes test resources.
```

Compile the Java classes:

`./gradlew compileJava` (Linux/Mac)

`gradlew.bat compileJava` (Windows)

Look for output in *build/classes/java/main*.

```
$ ls build/classes/java/main/com/gradlehero/languageapp/
SayHello.class
```

Process resources:

`./gradlew processResources` (Linux/Mac) `gradlew.bat processResources` (Windows)

Look for output in *build/resources/main*.

```
$ ls build/resources/main/
en.txt  es.tst
```

Generate a jar file:

`./gradlew jar` (Linux/Mac)

`gradlew.bat jar` (Windows)

Look for the jar file in *build/libs*.

```
$ ls build/libs/
get-going-with-gradle-practical.jar
```

7.6. Running the application

Run the following Java command:

`java -jar build/libs/get-going-with-gradle.jar en` (Linux/Mac/Windows)

You'll see an error which says *no main manifest attribute*.

Add this jar configuration to the end of *build.gradle.kts*:

```
tasks.named<Jar>("jar") {
  manifest {
    attributes["Main-Class"] = "com.gradlehero.languageapp.SayHello"
  }
}
```

This configuration sets up a *Main-Class* attribute in the jar manifest, which tells Java which class to run when the jar is executed.

Build the jar file again:

`./gradlew jar` (Linux/Mac)

`gradlew.bat jar` (Windows)

Rerun the Java command:

`java -jar build/libs/get-going-with-gradle.jar en` (Linux/Mac/Windows)

You should see the output text *Hello!*.

Run the same Java command but replace *en* with *es* for Spanish

`java -jar build/libs/get-going-with-gradle.jar es` (Linux/Mac/Windows)

You should see the output text *Hola!*.

Awesome! You've just built your first Java application with Gradle.

7.7. Testing the application

Tests live in *src/test/java*, so under *src* create a new directory *test/java*.

Within this directory create the same package structure by creating directories *com/gradlehero/languageapp*.

Add a new file *SayHelloTest.java* and open it for editing.

Paste in the following Java code:

```java
package com.gradlehero.languageapp;

import org.junit.jupiter.api.Test;

import com.gradlehero.languageapp.SayHello;

import java.io.IOException;

public class SayHelloTest {
    @Test
    public void testSayHello() throws IOException {
        SayHello.main(new String[]{"en"});
    }
}
```

Try running the tests:

`./gradlew test` (Linux/Mac)

`gradlew.bat test` (Windows)

You'll get an error here saying that it can't find the *org.junit.jupiter.api* package.

Add this dependency configuration block to *build.gradle.kts*, after the plugins:

```kotlin
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
}
```

Add this repositories configuration block, putting it just before the dependencies:

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

Rerun the test command:

`./gradlew test` (Linux/Mac)

`gradlew.bat test` (Windows)

The build should be successful.

Open the test report in a browser, located in *build/reports/tests/test/index.html*.

You'll see 0 tests have been run.

```
Test Summary
0
tests

0
failures

0
ignored

-
duration

-
successful
```

Add this test configuration block to *build.gradle.kts*, below the dependencies block:

```
tasks.named<Test>("test") {
    useJUnitPlatform()
}
```

This configures tests to use the Junit 5 platform.

Run the tests again:

`./gradlew test` (Linux/Mac)

`gradlew.bat test` (Windows)

Open the test report again in a browser, located in *build/reports/tests/test/index.html*.

You should now see that 1 test was run, with a 100% success rate.

```
Test Summary
1
tests

0
failures

0
ignored

0.019s
duration

100%
successful
```

7.8. Commit your changes

Get all the files ready to be committed:

`git add .`

See what files are staged for commit:

`git status`

Commit everything:

`git commit -m "Create Java project for language app."`

Good work! You’ve just built, run, and tested your first Java project with Gradle!

Chapter 8. Kotlin essentials for Gradle

In this chapter, you'll learn three key Kotlin language features to help you to better understand Gradle build scripts.

8.1. Feature 1: Kotlin is a JVM language

Kotlin is a scripting language built to run in the JVM (Java Virtual Machine).

Kotlin has some nice features that make writing code quicker than Java. Gradle uses these features to make defining your build very concise.

We'll cover some more advanced features in the rest of this chapter, but here are 3 fundamental differences between Kotlin and Java.

1. Semicolons are optional in Kotlin.

```
val name = "Bob"
println(name)
//prints Bob
```

2. Strings support interpolation by default.

```
var name = "Bob"
println("His name is ${name}")
// prints His name is Bob
```

3. `val` is for read-only variables and `var` is for mutable variables.

```
val name = "Bob"
println("${name} and always ${name}")
// prints Bob and always Bob

var mutableName = "Bob v2"
mutableName = "Bob v3"
println("New and improved ${mutableName}")
//prints New and improved Bob v3
```

8.2. Feature 2: lambda expressions

In Kotlin, lambda expressions are a way to pass a function around, then execute it at a later point.

```
val lambdaExpression = {
    for (i in 3 downTo 1) {
        println(i)
        }
        println(" Lift off!")
    }
lambdaExpression()
//prints 321 Lift off!
```

Lambda expressions are used heavily in the Gradle build script. Consider this *repositories* definition, for example.

```
repositories { ①
    mavenCentral() ②
}
```

① *repositories* is just a function call with a lambda expression as an argument.

② The contents of the lambda expression is `mavenCentral()`, which gets executed at a later point.

8.3. Feature 3: parentheses are optional in some scenarios

In Kotlin, parentheses are normally required when calling a function.

Take this Gradle dependency definition, for example.

```
dependencies {
    implementation("org.apache.commons:commons-lang3:3.13.0")
}
```

When we call the *implementation* function, we use parentheses to pass the dependency string argument.

However, when a lambda expression is the final argument to a function, the parentheses can be left out.

To illustrate this, let's look again at the *repositories* example from *Concept 2*.

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

The lambda expression is the final argument in the call to *repositories*, so the parentheses are omitted.

8.4. Why do I need to know this?

Why do you need to know these Kotlin language features? Isn't it enough just to know how to write a Gradle build script?

Whilst that may be enough for the basics, this knowledge will help fast-track your Gradle learning for these reasons:

1. If you use IntelliJ IDEA, you can browse the Gradle source code directly and understand what functions are available to use in your build script. This can be quicker than referring to documentation.

   For example, here's a function definition from one of the main Gradle interfaces called *Project*, containing the *repositories* function we just looked at, and many more.

   ```
   void    repositories(Closure configureClosure)
   Configures the repositories for this project.
   ```

   All these functions can be called directly from *build.gradle.kts*.

2. With this knowledge, you can make use of more advanced Gradle features like writing custom plugins. We don't cover these topics in this introductory book, but these fundamentals allow you to continue learning if you want to.

Chapter 9. The task graph

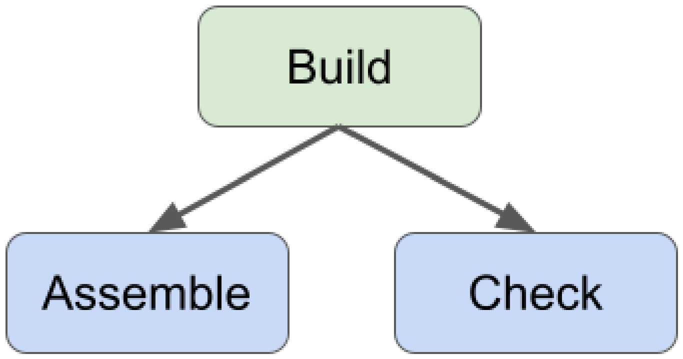
In this chapter, you'll learn in more detail what the Gradle task graph is, what it looks like in a Java project, and how you can make the best use of it in your day-to-day work.

9.1. Task graph essentials

We already discussed in *Introduction to Gradle* how tasks can have dependencies on other tasks. A dependency on another task means that the other task gets run first.

NOTE Task dependencies are a completely separate concept to the dependencies you setup for compiling your code, such as JUnit.

Let's use as an example some of the tasks added by the Java plugin. The *build* task depends on the *assemble* and *check* tasks.



This means that when you run `./gradlew build`, *assemble* and *check* get run first. We haven't used any of these tasks yet in this book, but don't worry because in a moment you'll see them in action.

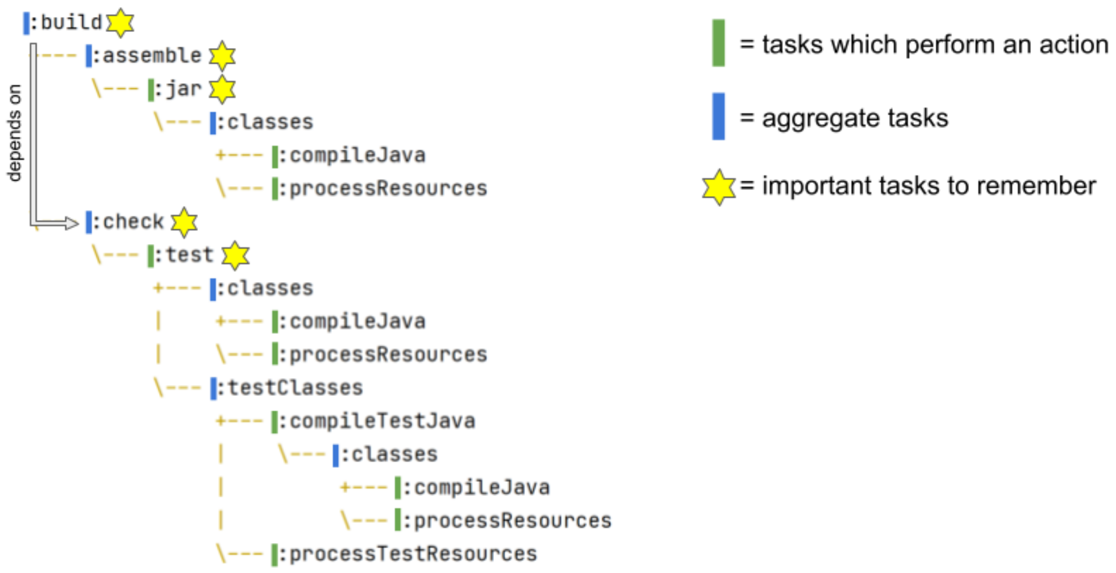
The interesting thing is that the *build* task itself doesn't do anything when it's run It's only there to *aggregate* both the *assemble* and *check* tasks together.

- **assemble** handles building your project including creating artifacts such as jar files.
- **check** handles testing your project.

The *build* task, using task dependencies, makes sure that both *assemble* and *check* get run. That's really nice, as it means if you ever need to build and test a whole project, you can just run a single task, *build*, to do everything.

9.2. Java plugin task graph

That was just 3 of the tasks added by the Java plugin. There are lots more, so let's look at the complete task graph.


depends on
= tasks which perform an action
= aggregate tasks
= important tasks to remember

```
:build
+--- :assemble
     \--- :jar
          \--- :classes
               +--- :compileJava
               \--- :processResources
:check
\--- :test
     +--- :classes
     |    +--- :compileJava
     |    \--- :processResources
     \--- :testClasses
          +--- :compileTestJava
          |    \--- :classes
          |         +--- :compileJava
          |         \--- :processResources
          \--- :processTestResources
```

Yes, there's quite a lot to look at here. Don't worry as we're going to step through this bit-by-bit

The lines that connect a task on the left to a task to the right imply a dependency on the task to right. so for example, *build* depends on *check*.

You already have practical experience of some of these tasks, including *jar*, *compileJava*, *processResources*, and *test*. These tasks are highlighted in green, showing tasks which actually perform an action when they're run.

For example:

- **jar** creates a jar file.
- **compileJava** compiles classes.
- **processResources** well…processes resources.
- **test** runs tests.

Let's take a look at the *classes* task, highlighted in blue. This task doesn't perform any action itself when it's run, it merely aggregates *compileJava* and *processResources* using task dependencies. Blue indicates aggregate tasks

Moving up the tree, *assemble* is another aggregate task, which when run causes the jar task to run, which in turn runs classes, which runs *compileJava* and *processResources*. You get the idea.

Test tasks

Let's focus now on the tasks under the *test* task. You'll notice we have the same *compileJava*, *processResources*, and *classes* tasks. This ensures that when we run the test task, our main application code has been properly compiled and resources processed

These 3 tasks are duplicated again under *compileTestJava*. *compileTestJava* itself is the task which compiles all your test code and similarly *processTestResources* processes all your test resources.

Above all these we have:

- *testClasses*, another aggregate task which depends on *compileTestJava* and *processTestResources*.
- *test*, which ensures that both the main code and test code are built, before running the tests themselves.
- *check*, another aggregate task which just depends on *test*.

Finally, at the very top level is *build*, an aggregate task which builds and tests everything in your project.

Thankfully you don’t need remember all this information in its entirety.

Important tasks

Highlighted with a star are five key Gradle tasks to remember which will help when using Gradle in your day-to-day work.

- **build** is used when you just need to build and test the whole project.
- **assemble** and **jar** you can think of as equivalent, and are used when you only need to assemble the project without testing it.
- **check** and **test** you can think of as equivalent, and are used to test your code without assembling it.

But why you wouldn’t just run *build* all the time? Well, medium to large projects can take a while to build, so picking the specific Gradle task for what you need to accomplish can save time.

For example, if you just need to test your code change then you can run *check* rather than *build*. This could be quicker, since *check* doesn’t depend on the *jar* task, so no *jar* is created.

Conversely, if you just need to *build* your project into a *jar* file, without testing your changes, you could run the *assemble* task.

Let’s now jump back into the project as we left it in the *Building a Java project with Gradle* practical chapter, to demonstrate a few of these important Gradle tasks.

9.3. Practical demonstration

Before we get started, we need to run a task which we haven’t talked about yet called *clean*.

This simple task deletes the *build* directory. Normally you don’t run this very often, but we’ll run it now to ensure we start in a fresh state and can see how each of the other tasks changes the directory structure.

So let’s run `./gradlew clean`.

```
$ ./gradlew clean

BUILD SUCCESSFUL in 1s
```

```
1 actionable task: 1 executed
```

You can see the *build* directory was deleted

```
$ ls build
ls: cannot access 'build': No such file or directory
```

assemble task

The first important task to run is *assemble*, which compiles your code, processes resources, and builds a jar file.

So let's run `./gradlew assemble`.

```
$ ./gradlew assemble

BUILD SUCCESSFUL in 2s
3 actionable tasks: 3 executed
```

Let's take a look in the newly generated *build* directory.

```
build
|-- classes
|   `-- java
|       `-- main ①
|           `-- com
|               `-- gradlehero
|                   `-- languageapp
|                       `-- SayHello.class
|-- generated
|   `-- sources
|       |-- annotationProcessor
|       |   `-- java
|       |       `-- main
|       `-- headers
|           `-- java
|               `-- main
|-- libs
|   `-- get-going-with-gradle-practical.jar ③
|-- resources
|   `-- main ②
|       |-- en.txt
|       `-- es.txt
`-- tmp
    |-- compileJava
    |   `-- previous-compilation-data.bin
    `-- jar
```

```
        `-- MANIFEST.MF
```

① Compiled classes are in the *classes/java/main* directory.

② Resources are in the *resources/main* directory.

③ A *jar* file was generated in the libs directory.

So *assemble* is great for generating all your build artifacts in one go. Notice that we don't have any test reports generated as *assemble* doesn't run the tests.

Let's delete the *build* directory again by running `./gradlew clean`.

```
$ ./gradlew clean

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

check task

The next important task to run is *check*. This task compiles the main code, processes the main resources, compiles test code, processes test resources, and actually runs the tests too

Let's run `./gradlew check`.

```
$ ./gradlew check

BUILD SUCCESSFUL in 2s
4 actionable tasks: 4 executed
```

Let's take a look in the *build* directory.

```
build
|-- classes
|   `-- java
|       |-- main ①
|       |   `-- com
|       |       `-- gradlehero
|       |           `-- languageapp
|       |               `-- SayHello.class
|       `-- test ①
|           `-- com
|               `-- gradlehero
|                   `-- languageapp
|                       `-- SayHelloTest.class
|-- generated
|   `-- sources
|       |-- annotationProcessor
|       |   `-- java
|       |       |-- main
|       |       `-- test
|       `-- headers
|           `-- java
|               |-- main
|               `-- test
|-- reports
|   `-- tests
|       `-- test ③
|           |-- classes
|           |   `-- com.gradlehero.languageapp.SayHelloTest.html
|           |-- css
|           |   |-- base-style.css
|           |   `-- style.css
|           |-- index.html
|           |-- js
|           |   `-- report.js
|           `-- packages
|               `-- com.gradlehero.languageapp.html
|-- resources
|   `-- main ②
|       |-- en.txt
|       `-- es.txt
|-- test-results
|   `-- test
|       |-- TEST-com.gradlehero.languageapp.SayHelloTest.xml
|       `-- binary
|           |-- output.bin
|           |-- output.bin.idx
|           `-- results.bin
`-- tmp
    |-- compileJava
    |   `-- previous-compilation-data.bin
    |-- compileTestJava
    |   `-- previous-compilation-data.bin
    `-- test
```

① We have both *classes/java/main* and *classes/java/test* because Gradle has compiled main and test classes.

② In the *resources* directory are only main resources, but if we had test resources they'd appear here too.

③ There's a reports directory containing a test report, since our tests were also executed.

Note that we don't have a *libs* directory or jar file, as the *check* task doesn't depend on *jar*.

So *check* is the task to run when you just need to run the tests without assembling everything in your project.

Let's delete the build directory once more with `./gradlew clean`.

```
$ ./gradlew clean

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

build task

The final important task to try is *build*, which depends on the two tasks we've just run, *assemble* and *check*.

Let's run `./gradlew build`.

```
$ ./gradlew build

BUILD SUCCESSFUL in 2s
5 actionable tasks: 5 executed
```

Let's have a look in the *build* directory.

```
build
|-- classes
|   `-- java
|       |-- main ①
|       |   `-- com
|       |       `-- gradlehero
|       |           `-- languageapp
|       |               `-- SayHello.class
|       `-- test ①
|           `-- com
|               `-- gradlehero
|                   `-- languageapp
|                       `-- SayHelloTest.class
|-- generated
|   `-- sources
|       |-- annotationProcessor
|       |   `-- java
|       |       |-- main
|       |       `-- test
|       `-- headers
|           `-- java
|               |-- main
|               `-- test
|-- libs
|   `-- get-going-with-gradle-practical.jar ③
|-- reports
|   `-- tests
|       `-- test ④
|           |-- classes
|           |   `-- com.gradlehero.languageapp.SayHelloTest.html
|           |-- css
|           |   |-- base-style.css
|           |   `-- style.css
|           |-- index.html
|           |-- js
|           |   `-- report.js
|           `-- packages
|               `-- com.gradlehero.languageapp.html
|-- resources
|   `-- main ②
|       |-- en.txt
|       `-- es.txt
|-- test-results
|   `-- test
|       |-- TEST-com.gradlehero.languageapp.SayHelloTest.xml
|       `-- binary
|           |-- output.bin
|           |-- output.bin.idx
|           `-- results.bin
`-- tmp
    |-- compileJava
    |   `-- previous-compilation-data.bin
    |-- compileTestJava
    |   `-- previous-compilation-data.bin
    |-- jar
    |   `-- MANIFEST.MF
    `-- test
```

① We have compiled main and test code.

② We have the processed resources.

③ We have a jar file in the libs directory.

④ We have test reports showing that the tests ran.

So *build* is the task to run when you just need to assemble your whole project and run all the tests.

NOTE The *build* directory also contains *generated*, *test-results*, and *tmp* directories, used internally by Gradle. You can ignore these directories.

9.4. Chapter summary

There was a lot of information in the last few pages, so here's a summary of the key takeaways.

1. Some tasks actually perform an action and some are just aggregate tasks.
2. Aggregate tasks combine multiple tasks together for convenience.
3. *assemble* and *jar* both compile code and build a jar file.
4. *check* and *test* both test your code.

5. *build* does both an *assemble* and a *check*.

You now have a clear understanding of the different tasks used by Gradle in Java projects and how they relate to each other.

Chapter 10. What now?

Congratulations on reaching this point of the book. You now have knowledge of the Gradle fundamentals as well as practical experience in creating Gradle projects from scratch.

So what now?

10.1. One final thing

You're ready to start using Gradle in your day-to-day work, whether that be your job, side-project, or just for fun. You have everything you need to work effectively with simple Java projects in Gradle.

To improve this book for you and other students, I'd really appreciate your opinion. This could help other students who might be considering whether to read this book. Please follow this link to share your thoughts.

10.2. Continue your Gradle learning

This was an introduction to Gradle, giving you the knowledge to work with simple Java projects in your day-to-day work.

To go beyond basic projects and make sure you implement Gradle effectively, I suggest you continue your learning to cover these additional topics:

- Using other dependency configurations to build Java projects more efficiently e.g *compileOnly* and *runtimeOnly*.
- Locating and configuring tasks to get your project built and tested exactly how you want.
- Running Java applications from Gradle with the *application* plugin.
- Building Spring Boot projects in Gradle most effectively.
- The differences between building applications and libraries, and what you need to do differently for each.
- Using the Java toolchain to ensure your code always builds consistently with the same Java version.

The good news is there are plenty of resources available online to cover these topics, like those over at docs.gradle.org.

But if you'd like to continue learning with me using a step-by-step approach, then the *Gradle Build Bible* book gets you to mastery of Java projects in Gradle as fast as possible.

If you found this book helpful, Gradle Build Bible multiplies that by a factor of 10, giving you everything you need to master every aspect of the Gradle build script.

- Never feel intimidated by a Gradle build script again!
- Take back control of your Gradle build and finally get your team working more productively.

- Ensure your Gradle build scales with the growth of your application, by implementing multi-project builds, and custom tasks and plugins.

In fact, this book was requested by engineers like you who wanted something to take them to the next level, having read this introductory book.

Just CLICK HERE to continue your journey today.