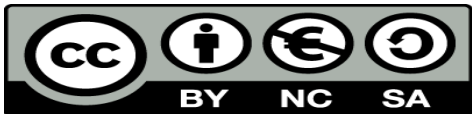
2. Basic JavaFX and Gradle

Part III: Gradle

Service and Process Programming

Arturo Bernal
Nacho Iborra

IES San Vicente



This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit
http://creativecommons.org/licenses/by-nc-sa/4.0/

Table of Contents

2. Basic JavaFX and Gradle

1 Gradle Introduction

Gradle is a flexible JVM based build tool. This tool uses a language called Groovy that runs on the Java Virtual Machine (Other languages that use the JVM to run are Scala, Clojure and Kotlin for example). Gradle is much more powerful and easy to understand than other XML based tools like Ant or Maven, and not only works with Java, but with C/C++, Python, and many more. Gradle is also the oficial build system for the Android pltaform.

Important features of Gradle are that you don't need to install it in your system thanks to the Gradle Wrapper (we'll talk more about it later), and you can work with the same Gradle project in console or in any of the most used IDEs (Netbeans, IntelliJ and Eclipse) thanks to their integration with the tool (no more incompatibilities between IDEs).

Gradle can also track dependencies on other external libraries and download them automatically from a Maven repository (along with other libraries that the ones to be downloaded depend on). It can distinguish between dependencies for testing, compiling or runtime environment.

Platforms like LinkedIn or Netflix among many others use Gradle as their build platform, and it's the fastest growing build platform for Java (and other languages) right now.

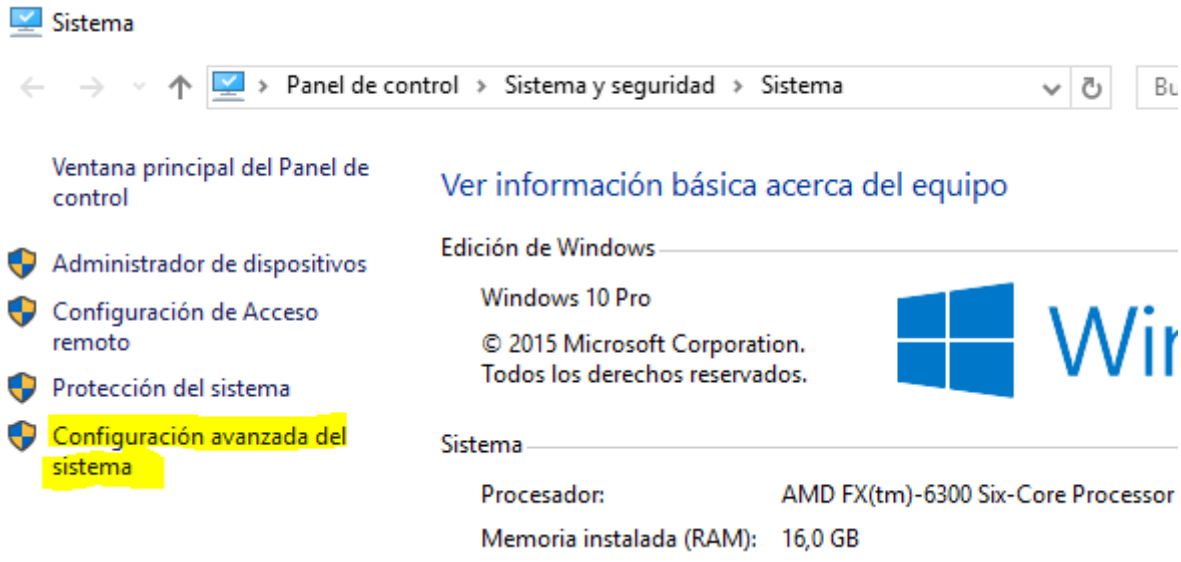
1.1 Gradle installation and Gradle Wrapper

Although the Gradle integration with different IDEs makes unnecessary to install Gradle in the system, it's usually not a bad idea to install it just in case (for example to create a new project from the command line or migrate an existing project).

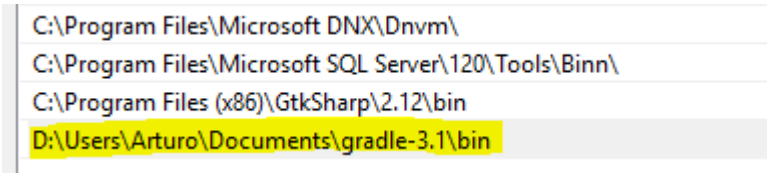
1.1.1 Installing on Windows

Installing Gradle on Windows is not as easy as installing on Linux or Mac, but is not difficult either. First, you'll have to download Grade from: https://gradle.org/gradle-download/ and extract it in a directory where you know where it is and you don't delete it accidentally.

Now, you'll have to create and update some environment variables. Go to your computer → properties and open advanced system settings.



Then click on **Environment Variables** (Variables de entorno). Edit the Path variable and add the route to the bin directory inside Gradle's main directory:



You should be ready to go.

```
C:\Users\Arturo>gradle -v

------------------------------------------------------------
Gradle 3.1
------------------------------------------------------------

Build time:   2016-09-19 10:53:53 UTC
Revision:     13f38ba699afd86d7cdc4ed8fd7dd3960c0b1f97

Groovy:       2.4.7
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_101 (Oracle Corporation 25.101-b13)
OS:           Windows 10 10.0 amd64
```

1.1.2 Installing on Linux

Installing Gradle on Linux (specially on a Ubuntu or derivative distribution) is really easy. If you have a recent version and Gradle is in the repositories, just install it with this command:

```
sudo apt-get install gradle
```

However, if the version included in your distribution's repository is not recent and you want to install the latest (or almost) version of Gradle, just add this PPA and install from it:

```
sudo add-apt-repository ppa:cwchien/gradle
sudo apt-get update
sudo apt-get install gradle
```

Now, we can check if the tool is installed by checking the version:

```
arturo@arturo-Lenovo:~$ gradle -v

------------------------------------------------------------
Gradle 2.14.1
------------------------------------------------------------

Build time:   2016-07-18 06:38:37 UTC
Revision:     d9e2113d9fb05a5caabba61798bdb8dfdca83719

Groovy:       2.4.4
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_101 (Oracle Corporation 25.101-b13)
OS:           Linux 4.4.0-34-generic amd64
```

1.1.3 Installing on Mac

To install Gradle in Mac, the simplest way is to install the Homebrew packet manager first (write everything on the same line):

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Then execute this command to install the tool:

```
brew install gradle
```

```
MacBook-Pro-de-[redacted]:~ [redacted]$ gradle -v

------------------------------------------------------------
Gradle 3.0
------------------------------------------------------------

Build time:   2016-08-15 13:15:01 UTC
Revision:     ad76ba00f59ecb287bd3c037bd25fc3df13ca558

Groovy:       2.4.7
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_73 (Oracle Corporation 25.73-b02)
OS:           Mac OS X 10.11.6 x86_64
```

Another tool we can use (In Linux or any based Unix system like Mac) is SDKMAN, which is very similar to Homebrew:

```
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk install gradle
```

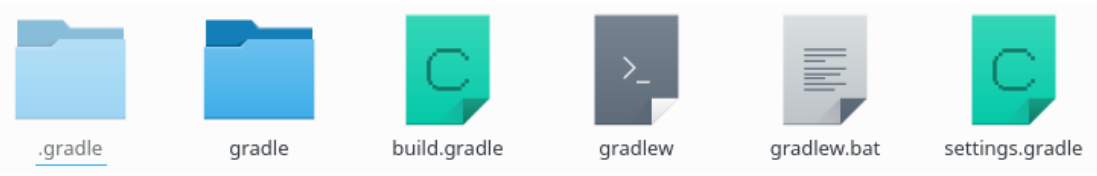
1.1.4 Gradle Wrapper

The Gradle Wrapper is a script, available for Bash (**gradlew**) and Windows (**gradlew.bat**). This script downloads a specific version of Gradle to use in the project and works the same way as the **gradle** command. This gives the advantage when you are in a developer team that everyone works with the same Gradle version. Also, nobody working in the project needs to install Gradle onto their system if the project works with this wrapper.

We will see how to include Gradle Wrapper in our project manually, and how creating a Gradle project from an IDE already includes the wrapper.

1.2 Creating a Gradle project manually

To create a Java project with Gradle manually, first you must have installed Gradle into your system. Then create a main directory for the project and execute the command: **gradle init**. This command will create an standard structure and install the Gradle Wrapper. Now, inside the main directory you'll see something like this:



As you can see, along with the Gradle Wrapper, the main configuration files have been created. In the file **settings.gradle**, you should see the name of the project:

```
rootProject.name = 'HelloGradle'
```

In the text file called **gradle.build**, a standard configuration will be established (you'll need to **uncomment it**). This, for example, is the way to tell Gradle to make standard Java tasks such as **build** or **test** available in the project:

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

We'll see what the other parts of this file mean in future sections. Now, we are going to look at the tasks available. Now we can run Gradle commands using the wrapper (recommended) using **gradlew** (Linux, Mac/UNIX) or **gradlew.bat** (Windows). Remember that that this file needs to have execution permissions. The first time we execute a command with the wrapper, it will automatically download the corresponding version of Gradle and place it into .**gradle/wrapper/dists** inside our home directory.

Lets take a brief look at the Build tasks that Gradle offers by default for Java projects:

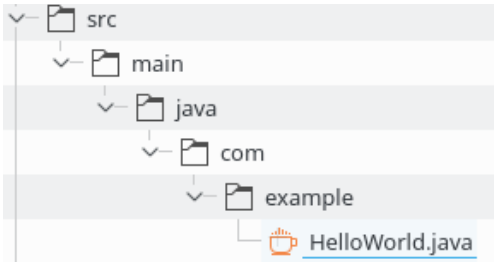
```
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean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

In order to try and build our project, we are going to create a simple Java file that outputs a "Hello World" message. By default this is where Gradle will try to find our Java code and test files:

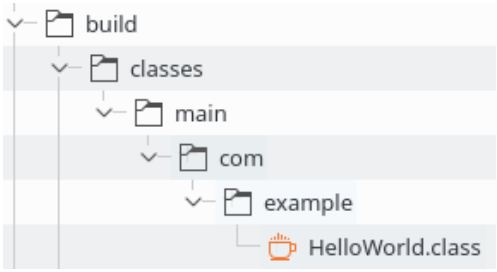
- **src/main/java** → Java files that will be compiled
- **src/main/resources** → Other files that will be copied such as image and text files.
- **src/test/java** → JavaUnit or other testing framework's Java files.
- **src/test/resources** → Other files used for testing purposes.

We create the necessary directories and the Java file inside the main directory:

```
package com.example;

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



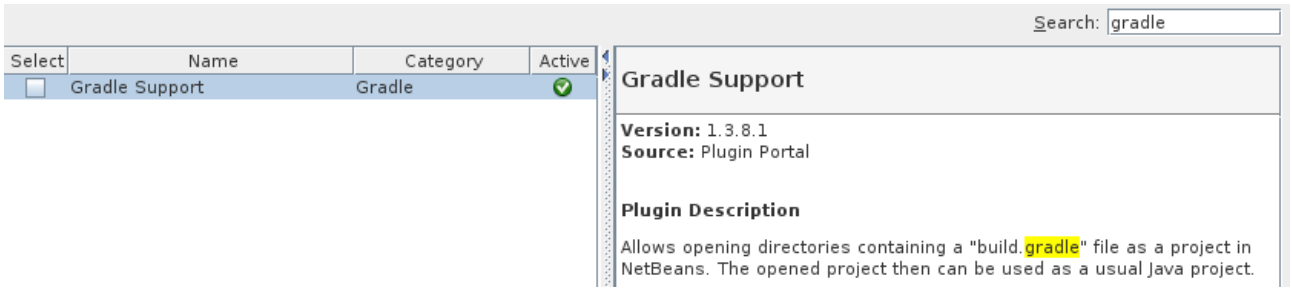
Now, if we execute **./gradlew build**, it will compile our class and create a build structure inside our directory.



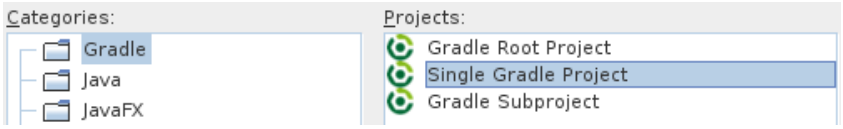

```
$: java -cp build/classes/main/ com.example.HelloWorld
Hello World!
```

1.3 Creating a Gradle project in Netbeans

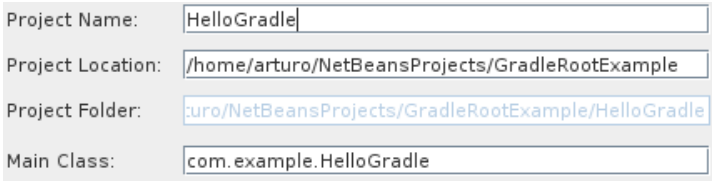
In order to create (or open existing) Gradle projects in Netbeans, first we have to make sure that the Gradle plugin is installed, or install it from **tools** → **plugins**:

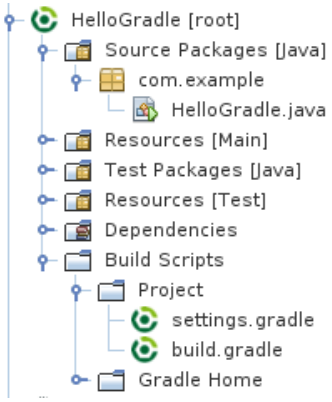


Now, when we create a new project, we'll have the option to create a Gradle project:

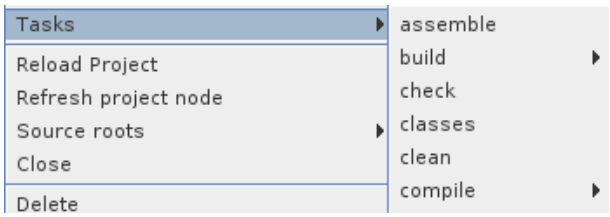


Gradle can manage multiple projects inside the same directory (Gradle root project with Gradle subprojects). But usually, we'll be interested in creating a Single Gradle Project.





It will create the default directory structure that we described in section 1.2. If we click with the right mouse button on the Project's name, we'll have access to predefined Gradle tasks (the same tasks that appeared executing ./gradlew task).



2 Gradle tasks

We can create custom tasks inside the **build.gradle** file. Remember that Gradle uses Groovy, a language that runs on the JVM and it's similar to Java but not identical. A task is some code that Gradle executes. It has a name, a lifecycle (initialization phase, configuration phase and execution phase), properties, actions and dependencies. This are three ways on which we can create a new task (with the name "task1"):

task("task1") ↔ task "task1" ↔ task task1

We can add a description to a task and some code to execute with the property **doLast**. We can do it in different ways:

```
task task1
task1.description = "This is task number 1"
task1.doLast { println "Hello, this is task 1" }
task1 << { println "Using << is the same as .doLast (and appends the code)"}

task task2 {
    description "This is task number 2"
    doLast {
        println "Hello, this is task 2"
        println "I can also execute multiple instructions from here"
    }
}
```

If we run **./gradlew tasks**, we'll see now that our custom tasks appear on the list:

```
Other tasks
-----------
task1 - This is task number 1
task2 - This is task number 2
```

Now we can run a custom task using its name and see how its code executes:

```
$: ./gradlew task1
:task1
Hello, this is task 1
Using << is the same as .doLast (and appends the code)

BUILD SUCCESSFUL

Total time: 4.289 secs
```

We can also silence the default Gradle output and see only what our task outputs to the console by using the **-q** (quiet) option:

```
$: ./gradlew -q task2
Hello, this is task 2
I can also execute multiple instructions from here
```

2.1 Task dependencies

We can tell a task that depends on another task by using the **dependsOn** property. That way we are telling Gradle to execute first the task (or tasks separated by comma) specified on that property before executing the current task:

```
task2.dependsOn task1
```

```
$: ./gradlew task2
:task1
Hello, this is task 1
Using << is the same as .doLast (and appends the code)
:task2
Hello, this is task 2
I can also execute multiple instructions from here
```

However, we should be careful in order to avoid circular dependencies (in this example task1 must execute before task2 but also task2 mus execute before task1, which makes no sense and it's impossible):

```
task2.dependsOn task1
task1.dependsOn task2
```

```
$: ./gradlew task2

FAILURE: Build failed with an exception.

* What went wrong:
Circular dependency between the following tasks:
:task1
\--- :task2
     \--- :task1 (*)
```

Another simple example with more tasks and dependencies would be this one:

```
task task1 << { println "Task 1"}
task task2 << { println "Task 2"}
task task3 << { println "Task 3"}
task task4 << { println "Task 4"}
task task5 << { println "Task 5"}

task1.dependsOn task2,task3,task4
task2.dependsOn task5
```

Now tasks 2, 3 and 4 must execute before task 1, and also task 5 must also execute before task 2, giving this result when task 1 is executed:

```
$: ./gradlew -q task1
Task 5
Task 2
Task 3
Task 4
Task 1
```

Also when executing more than one task we can use the properties mustRunAfter to tell that a task must run after another task (can have circular dependencies problem), or shouldRunAfter, which makes Gradle try to execute those task in the established order but if there's a risk of circular dependency, it won't respect the exact order.

```
task1.dependsOn task2,task3,task4
task2.dependsOn task5
```

```
task2.shouldRunAfter task3,task4
task4.shouldRunAfter task2 // Impossible!!
```

```
$: ./gradlew -q task1
Task 3
Task 4
Task 5
Task 2
Task 1
```

To execute a task **after** another instead of before we use the property **finalizedBy**:

```
task1.finalizedBy task3,task5
```

```
$: ./gradlew -q task1
Task 1
Task 3
Task 5
```

You can learn more here: https://docs.gradle.org/current/userguide/more_about_tasks.html

2.2 Task properties

We can define global properties for our project in the build.gradle file, using the **def** keyword and declaring it as a variable (name = value). Then, we can use those properties inside our tasks by putting their name with a $ (dollar) before.

```
def author = "Arturo"

task task2 {
    description "This is task number 2"
    doLast {
        println "Hello $author, this is task 2"
    }
}
```

```
$: ./gradlew -q task2
Hello Arturo, this is task 2
```

2.3 Typed tasks

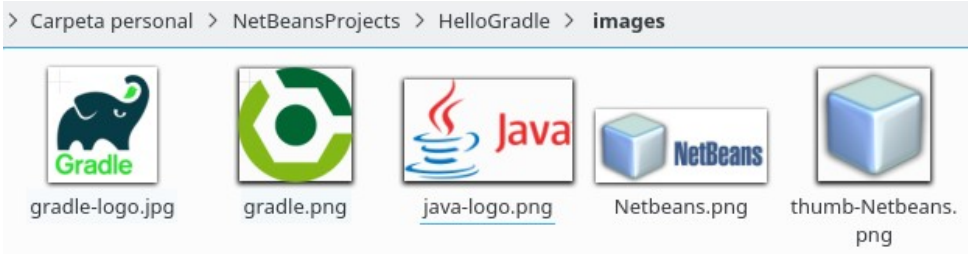
There are some predefined task in Gradle that make much more simple doing some actions like copying or compressing files, uploading files, testing, etc. We can see what types of task we can define (apart from generic tasks) here (scroll to the section called **Task types**): https://docs.gradle.org/current/dsl/

2.3.1 Copy task

For example, if we want to make a task that copies a set of files into other directory, we can (instead of making all the code necessary in Groovy) create a task if type Copy. This task will have some defined properties for the job:

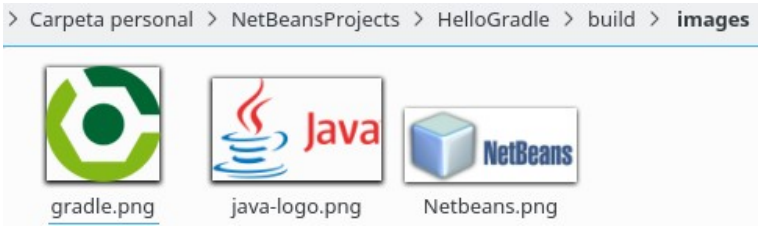
```
task copyImages (type: Copy) {
    include '**/*.png','**/*.PNG' // Files that will be copied
    exclude '**/thumb*' // Files that start with thumb won't be copied
    from 'images' // Origin directory (in project's root)
    into 'build/images' // Destination directory
}
```

Now, imagine that we have these images inside our images/ directory:



If we execute the **copyImages** task and go to **build/images/**, all PNG images (except the one that starts with thumb) will have been copied there.

```
$: gradle copyImages
:copyImages

BUILD SUCCESSFUL
```



2.3.2 Zip task

In this example, we'll compress all files inside **images/** directory into an **images.zip** file:

```
task zipImages(type:Zip) {
    from ('images/') // Directory where the files are
    include '*' // Include all files
    archiveName 'images.zip' // Name of the resulting zip archive
    destinationDir file('.') // directory where the zip archive will be placed
}
```

2.3.3 Wrapper task

The Wrapper task allows us to specify the version of Gradle that the Gradle Wrapper will use. Check that you specify a correct existing version.

```
task wrapper(type: Wrapper) {
    gradleVersion = '3.0'
}
```

Now, if we execute the **gradle wrapper** task, anytime we use the Gradle Wrapper in the project for anything, it will check if the version specified is downloaded, and if not, it will download it for us. This a very comfortable way to update the Gradle version included in our project.

```
$: ./gradlew -v
Downloading https://services.gradle.org/distributions/gradle-3.0-bin.zip
```

```
------------------------------------------------------------
Gradle 3.0
------------------------------------------------------------

Build time:   2016-08-15 13:15:01 UTC
Revision:     ad76ba00f59ecb287bd3c037bd25fc3df13ca558

Groovy:       2.4.7
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_101 (Oracle Corporation 25.101-b13)
OS:           Linux 4.4.0-34-generic amd64
```

Exercise 7

Create a Gradle project named **GradleWithDoc** with a class inside it (with the main method). The name of the class doesn't matter. Create at least two public methods inside that class with some javaDoc documentation format comments describing them.

Create a Gradle task of type JavaDoc named **generateDoc** that will generate the JavaDoc documentation for that class inside a directory called **doc**.

Hint: If you follow the example on Gradle's web, you'll only need to specify **source** and **destinationDir** properties, nothing else. After you run the task check the directory, **build/reports/docs**. That's where the JavaDoc documentation should be.

3 Library dependencies

It's very common that our project depends on other projects JAR files or on external libraries like JSON, Junit, Hibernate, etc. The most common way to include external dependencies is through local JAR files or Maven repositories (local or remote).

Also, dependencies can be classified into testing, compile or runtime dependencies (compile dependencies are also included as runtime dependencies). If a library that is a dependency of our project has other dependencies (transitive dependencies), they will also be downloaded with it.

To see the dependencies defined on a project, run this command:

```
$: ./gradlew -q dependencies

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

archives - Configuration for archive artifacts.
No dependencies

compile - Dependencies for source set 'main'.
\--- org.slf4j:slf4j-api:1.7.21
```

You can also run gradle -q dependencies --configuration {compile|runtime|testCompile|…} to see only those specific dependencies:

```
$: ./gradlew -q dependencies --configuration testCompile

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

testCompile - Dependencies for source set 'test'.
+--- org.slf4j:slf4j-api:1.7.21
\--- junit:junit:4.12
     \--- org.hamcrest:hamcrest-core:1.3
```

3.1 Adding dependencies

To add a dependency, you must include a repository that has it (local or remote so it can be downloaded). We can include Maven's central repositories by calling inside the **repositories block** mavenCentral() → http, or jcenter() → https, recommended:

```
repositories {
    // Use 'jcenter' for resolving your dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}
```

We can include other external repositories by specifying its url like this:

```
url "http://jcenter.bintray.com/"
```

If we look at the default **dependencies block** we'll see something like this:

```
dependencies {
    compile 'org.slf4j:slf4j-api:1.7.21'
    testCompile 'junit:junit:4.12'
}
```

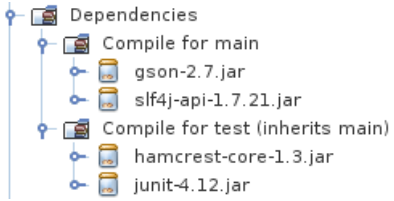
To define dependencies, we write first what kind of dependencies we want to specify (compile, runtime, testCompile, testRuntime, …), and then, a string containing the group of the package, the package name and its version (separated by a colon ':' character).

To know that information, go to https://mvnrepository.com/ and search for the package (library) you want to add. For example, lets include gson (Google's library for parsing JSON). We search for it and select the version we want to include, and then go to the **Gradle** tab and see what we have to include:

Instead of using this format, we can do the same by writing:

```
compile 'com.google.code.gson:gson:2.7'
```

We can check that everything is correct by right-clicking on dependencies in our Netbeans project and selecting 'Download sources' option (or running the project). If we now reload the project in our IDE we should see that included dependencies have been updated.



To add a local JAR file as a dependency, do it like this:

```
runtime files('libs/library1.jar', 'libs/library2.jar')
runtime fileTree(dir: 'libs', include: '*.jar') // Every JAR inside libs/
```

To add another local Gradle project as a dependency (lets call it Dependency), put it inside your main project's directory and include it like this:

```
include ':Dependency' // This goes outside
...
dependencies {
    ...
    compile project(':Dependency')
}
```

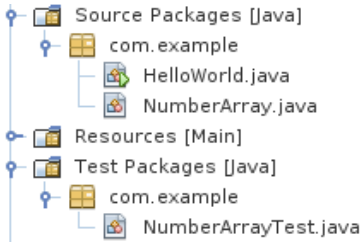
3.2 Testing with JUnit

Gradle integrates very well with testing tools like JUnit. It has predefined testing tasks that look for unit tests inside directory **src/test/java**, compiles those tests into **build/classes/tests** and generates test reports inside **build/reports/test** (this is by default. It can be changed).

Lets assume that we have this class inside our **com.example** package:

```
public class NumberArray {
    private List<Integer> list;

    public NumberArray() {
        this.list = new ArrayList<>();
    }

    public void addNumber(Integer newNumber) {
        list.add(newNumber);
    }

    public void deleteNumber(int index) {
        list.remove(index);
    }

    public Integer getNumber(int index) {
        return list.get(index);
    }

    public Integer sumNumbers() {
        return list.stream().mapToInt(n -> n).sum();
    }
}
```

As you can see this is a very simple class representing an array of integers, but enough to show how to make some tests based on it. We'll create a class called **NumberArrayTest** that will contain some unit tests for this class inside the test folder.

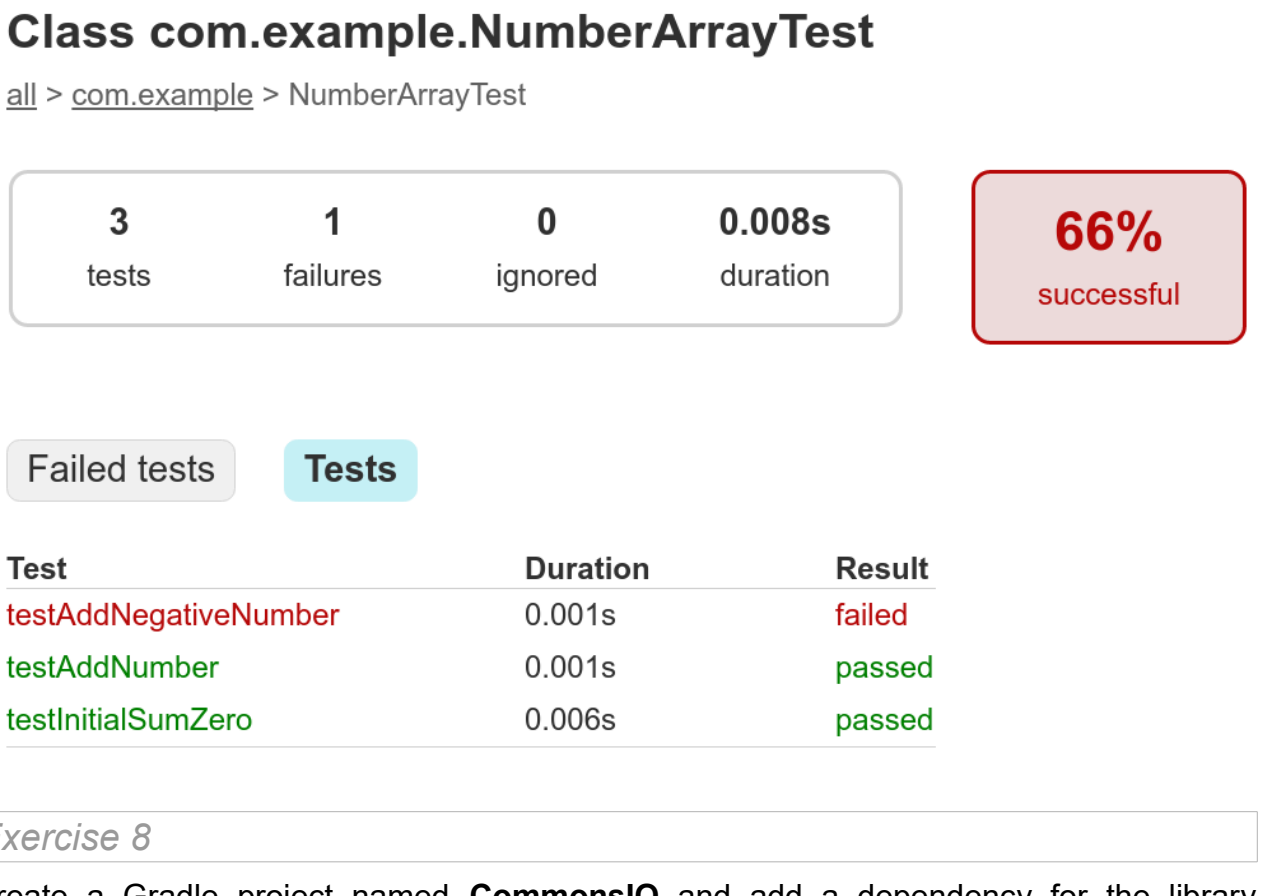


This class will contain three test methods. When we run the built-in **gradle test** task on the console or from the IDE, it should run the test and inform us if anything went wrong.

```
public class NumberArrayTest {
    private NumberArray nArray;

    @Before // This will be called before each test. Avoids code duplication.
    public void initialize() {
        nArray = new NumberArray();
    }

    @Test
    public void testInitialSumZero() {
        // Error message (optional), expected value, real value
        Assert.assertEquals("The total sum at the beginning is not 0!", 0,
                            nArray.sumNumbers().intValue());
    }

    @Test
    public void testAddNumber() {
        nArray.addNumber(10);
        Assert.assertEquals("Added only 10, the total sum should be 10!", 10,
                            nArray.sumNumbers().intValue());
    }

    @Test(expected = IllegalArgumentException.class) // Exception expected
    public void testAddNegativeNumber() {
        nArray.addNumber(-5); // Numbers shouldn't be negative
    }
}
```

If we look at these tests, the first two should work, but the third should fail, because we are expecting that negative values can't be inserted into the array (IllegalArgumentException thrown) but in the class method we don't check anything and don't throw any exception.

If we open the **build/reports/tests/index.html** file, we'll see more detailed information about the tests runt (Go to: classes → com.example.NumberArrayTest → Tests).



Class com.example.NumberArrayTest

all > com.example > NumberArrayTest

3	1	0	0.008s
tests	failures	ignored	duration

66% successful

Failed tests | Tests

Test	Duration	Result
testAddNegativeNumber	0.001s	failed
testAddNumber	0.001s	passed
testInitialSumZero	0.006s	passed

Exercise 8

Create a Gradle project named **CommonsIO** and add a dependency for the library https://mvnrepository.com/artifact/commons-io/commons-io/2.5. From that library use the **FileUtils** class and its method **readLines**, to read the lines of any file and print them on the console. Example.