



# Building Java Applications

Ken Kousen

---

## Table of Contents

What you'll build

What you'll need

Check the user manual

Run the init task

Review the generated project files

Execute the build

Run the application

Summary

Help improve this guide

---

This guide demonstrates how to create a Java project in the standard form using Gradle's Build Init plugin.

## What you'll build

You'll generate a Java application with the standard layout.

## What you'll need

- About 8 minutes
- A text editor
- A command prompt
- The Java Development Kit (JDK), version 1.7 or higher
- Any recent Gradle distribution

## Check the user manual

Gradle comes with a built-in plugin called the Build Init plugin. It is documented in the Gradle User Manual at [https://docs.gradle.org/current/userguide/build\\_init\\_plugin.html](https://docs.gradle.org/current/userguide/build_init_plugin.html).

The plugin has one task, called `init`, that generates the project. The `init` task calls the (also built-in) `wrapper` task to create a Gradle wrapper script, `gradlew`.

To run the `init` task, you run the following from a command prompt:

```
$ gradle init --type <name>
```

where `name` is one of the following:

- `java-application`
- `java-library`
- `scala-library`
- `groovy-library`
- `basic`

This guide uses the `java-application` type.

The first step is to create a folder for the new project and change directory into it.

```
$ mkdir java-demo  
$ cd java-demo
```

## Run the init task

From inside the new project directory, run the `init` task with the `java-application` argument.

```
$ gradle init --type java-application  
Starting a Gradle Daemon (subsequent builds will be faster)  
:wrapper  
:init
```

```
BUILD SUCCESSFUL
```

The `init` task runs the `wrapper` task first, which generates the `gradlew` and `gradlew.bat` wrapper scripts. Then it creates the new project with the following structure:

```
|— build.gradle
|— gradle      1
|   |— wrapper
|   |   |— gradle-wrapper.jar
|   |   |— gradle-wrapper.properties
|— gradlew
|— gradlew.bat
|— settings.gradle
|— src
|   |— main
|   |   |— java      2
|   |   |   |— App.java
|   |— test          3
|   |   |— java
|   |   |   |— AppTest.java
```

- 1 Generated folder for wrapper files
- 2 Default Java source folder
- 3 Default Java test folder

## Review the generated project files

The `settings.gradle` file is heavily commented, but has only one active line:

*settings.gradle*

```
rootProject.name= 'java-demo'
```

GROOVY

This assigns the name of the root project to `java-demo`, which is the default.

The generated `build.gradle` file also has many comments. The active portion is reproduced here (note version numbers for the dependencies may be updated in later versions of Gradle):

### *build.gradle*

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

GROOVY

```
repositories {
    jcenter() 1
}
```

```
dependencies {
    compile 'com.google.guava:guava:21.0' 2
    testCompile 'junit:junit:4.12' 3
}
```

```
mainClassName = 'App' 4
```

- 1 public Bintray Artifactory repository
- 2 Google Guava library
- 3 JUnit testing library
- 4 Class with "main" method (used by Application plugin)

The build file adds the `java` and `application` plugins. The former support Java projects. The latter lets you designate one class as having a `main` method, which can be executed by the build from the command line. In the demo, the name of the `main` class is `App`.

The file `src/main/java/App.java` is shown here:

### *src/main/java/App.java*

```
public class App {  
    public String getGreeting() {  
        return "Hello world.";  
    }  
  
    public static void main(String[] args) { 1  
        System.out.println(new App().getGreeting());  
    }  
}
```

1 Called by Application plugin "run" task

The test class, `src/test/java/AppTest.java` is shown next:

*src/test/java/AppTest.java*

```
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class AppTest {  
    @Test public void testAppHasAGreeting() {  
        App classUnderTest = new App();  
        assertNotNull("app should have a greeting",  
            classUnderTest.getGreeting());  
    }  
}
```

The generated test class has a single test annotated with JUnit's `@Test` annotation. The test instantiates the `App` class, invokes the `getGreeting` method, and checks that the returned value is not null.

## Execute the build

To build the project, run the `build` command. You can use the regular `gradle` command, but when a project includes a wrapper script, it is considered good form to use it instead.

```
$ ./gradlew build
:compileJava
:processResources NO-SOURCE
:classes
:jar
:startScripts
:distTar
:distZip
:assemble
:compileTestJava
:processTestResources NO-SOURCE
:testClasses
:test
:check
:build
```

BUILD SUCCESSFUL



The first time you run the wrapper script, `gradlew`, there may be a delay while that version of `gradle` is downloaded and stored locally in your `~/ .gradle/wrapper/dists` folder.

The first time you run the build, Gradle will check whether or not you already have the Guava and JUnit libraries in your cache under your `~/ .gradle` directory. If not, the libraries will be downloaded and stored there. The next time you run the build, the cached versions will be used. The `build` task compiles the classes, runs the tests, and generates a test report.

You can view the test report by opening the HTML output file, located at `build/reports/tests/test/index.html`.

A sample report is shown here:

## Test Summary

**1** tests  
**0** failures  
**0** ignored  
**0.001s** duration

**100%**  
successful

Packages

Classes

Package	Tests	Failures	Ignored	Duration	Success rate
<a href="#">default-package</a>	1	0	0	0.001s	100%

## Run the application

Because the Gradle build used the Application plugin, you can run the application from the command line. First, use the `tasks` task to see what task has been added by the plugin.



```
$ ./gradlew tasks
:tasks

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

Application tasks
-----
run - Runs this project as a JVM application

// ... many other tasks ...
```

The `run` task tells Gradle to execute the `main` method in the class assigned to the `mainClassName` property.

```
$ ./gradlew run
:compileJava UP-TO-DATE
:processResources NO-SOURCE
:classes UP-TO-DATE
:run
Hello world.
```

BUILD SUCCESSFUL

## Summary

You now have a new Java project that you generated using Gradle's build init plugin. In the process, you saw:

- How to generate a Java application
- How the generated build file and sample Java files are structured

- How to run the build and view the test report
- How to execute a Java application using the `run` task from the Application plugin

## Help improve this guide

Have feedback or a question? Found a typo? Like all Gradle guides, help is just a GitHub issue away. Please add an issue or pull request to [gradle-guides/creating-java-applications](https://github.com/gradle-guides/creating-java-applications) (<https://github.com/gradle-guides/creating-java-applications/>) and we'll get back to you.

Last updated 2017-08-09 20:16:38 UTC