Docs

[User Manual](http://docs.google.com/userguide/userguide.html)

[Guides and Tutorials](https://guides.gradle.org)

[DSL Reference](http://docs.google.com/dsl/)

[Javadoc](http://docs.google.com/javadoc/)

[Release Notes](http://docs.google.com/release-notes.html)

[Forums](https://discuss.gradle.org/)

[Training](https://gradle.org/training/)

[Try Gradle Enterprise](https://gradle.com/enterprise)

[PDF](http://docs.google.com/userguide/userguide.pdf)

* [User Manual Home](http://docs.google.com/userguide/userguide.html)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Installing Gradle](http://docs.google.com/userguide/installation.html)
* [Tutorials](https://guides.gradle.org/)

### Reference

* [Groovy DSL Reference](http://docs.google.com/dsl/)
* [Gradle API Javadoc](http://docs.google.com/javadoc/)
* [Core Plugins](http://docs.google.com/userguide/plugin_reference.html)
* [Gradle & Third-party Tools](http://docs.google.com/userguide/third_party_integration.html)

### Getting Started

* [Creating New Gradle Builds](https://guides.gradle.org/creating-new-gradle-builds/)
* [Creating Build Scans](https://guides.gradle.org/creating-build-scans/)
* [Migrating From Maven](https://guides.gradle.org/migrating-from-maven/)

### Running Gradle Builds

* [Command-Line Interface](http://docs.google.com/userguide/command_line_interface.html)
* [Customizing Execution](#gjdgxs)
  + [Configuring the Build Environment](http://docs.google.com/userguide/build_environment.html)
  + [Configuring the Gradle Daemon](http://docs.google.com/userguide/gradle_daemon.html)
  + [Initialization Scripts](http://docs.google.com/userguide/init_scripts.html)
* [Directory Layout](http://docs.google.com/userguide/directory_layout.html)
* [Executing Multi-Project Builds](http://docs.google.com/userguide/intro_multi_project_builds.html)
* [Gradle Wrapper](http://docs.google.com/userguide/gradle_wrapper.html)
* [Troubleshooting](http://docs.google.com/userguide/troubleshooting.html)
* [Using Build Scans](https://docs.gradle.com/build-scan-plugin)
* [Enabling and Configuring the Build Cache](http://docs.google.com/userguide/build_cache.html)
* [Integrating Separate Gradle Builds (Composite Builds)](http://docs.google.com/userguide/composite_builds.html)

### Authoring Gradle Builds

* [Fundamentals](#30j0zll)
  + [Introducing the Basics of Build Scripts](http://docs.google.com/userguide/tutorial_using_tasks.html)
  + [Working with Tasks](http://docs.google.com/userguide/more_about_tasks.html)
  + [Learning More About Build Scripts](http://docs.google.com/userguide/writing_build_scripts.html)
  + [Working with Files](http://docs.google.com/userguide/working_with_files.html)
  + [Creating Custom Task Types](http://docs.google.com/userguide/custom_tasks.html)
  + [Using Gradle Plugins](http://docs.google.com/userguide/plugins.html)
  + [The Standard Gradle Plugins](http://docs.google.com/userguide/standard_plugins.html)
  + [Understanding the Build Lifecycle](http://docs.google.com/userguide/build_lifecycle.html)
  + [Working with Logging](http://docs.google.com/userguide/logging.html)
  + [Configuring Multi-Project Builds](http://docs.google.com/userguide/multi_project_builds.html)
* [Best Practices](#1fob9te)
  + [Authoring Maintainable Build Scripts](http://docs.google.com/userguide/authoring_maintainable_build_scripts.html)
  + [Organizing Gradle Projects](http://docs.google.com/userguide/organizing_gradle_projects.html)
  + [Optimizing Build Performance](https://guides.gradle.org/performance/)
  + [Using the Build Cache](https://guides.gradle.org/using-build-cache/)
* [Dependency Management](#3znysh7)
  + [Introduction to Dependency Management](http://docs.google.com/userguide/introduction_dependency_management.html)
  + [Dependency Management Terminology](http://docs.google.com/userguide/dependency_management_terminology.html)
  + [Dependency Types](http://docs.google.com/userguide/dependency_types.html)
  + [Repository Types](http://docs.google.com/userguide/repository_types.html)
  + [Declaring Dependencies](http://docs.google.com/userguide/declaring_dependencies.html)
  + [Declaring Repositories](http://docs.google.com/userguide/declaring_repositories.html)
  + [Inspecting Dependencies](http://docs.google.com/userguide/inspecting_dependencies.html)
  + [Managing Dependency Configurations](http://docs.google.com/userguide/managing_dependency_configurations.html)
  + [Managing Transitive Dependencies](http://docs.google.com/userguide/managing_transitive_dependencies.html)
  + [Dependency Locking](http://docs.google.com/userguide/dependency_locking.html)
  + [Troubleshooting Dependency Resolution](http://docs.google.com/userguide/troubleshooting_dependency_resolution.html)
  + [Customizing Dependency Resolution Behavior](http://docs.google.com/userguide/customizing_dependency_resolution_behavior.html)
  + [Dependency Cache Internals](http://docs.google.com/userguide/dependency_cache.html)
  + [Working with Dependencies](http://docs.google.com/userguide/working_with_dependencies.html)
* [Publishing Artifacts](http://docs.google.com/userguide/artifact_management.html)
* [C++ Projects](#2et92p0)
  + [Building Native Software](http://docs.google.com/userguide/native_software.html)
  + [Software Model Concepts](http://docs.google.com/userguide/software_model_concepts.html)
  + [Rule-based Model Configuration](http://docs.google.com/userguide/software_model.html)
  + [Implementing Model Rules in a Plugin](http://docs.google.com/userguide/rule_source.html)
  + [Extending the Software Model](http://docs.google.com/userguide/software_model_extend.html)
* [Java Projects](#tyjcwt)
  + [Building Java & JVM projects](http://docs.google.com/userguide/building_java_projects.html)
  + [Testing Java & JVM projects](http://docs.google.com/userguide/java_testing.html)
* [Advanced Techniques](#3dy6vkm)
  + [Configuring Tasks Lazily](http://docs.google.com/userguide/lazy_configuration.html)
  + [Developing Parallel Tasks](https://guides.gradle.org/using-the-worker-api/)
  + [Testing Your Build with TestKit](http://docs.google.com/userguide/test_kit.html)
  + [Using Ant from Gradle](http://docs.google.com/userguide/ant.html)
* [Sample Gradle builds](#1t3h5sf)
  + [Groovy DSL Samples](https://github.com/gradle/gradle/tree/master/subprojects/docs/src/samples)
  + [Kotlin DSL Samples](https://github.com/gradle/kotlin-dsl/tree/master/samples)

### Extending Gradle

* [Writing Custom Plugins](http://docs.google.com/userguide/custom_plugins.html)
* [Plugin Development Guides](https://gradle.org/guides/?q=Plugin+Development)

[Edit this page](https://github.com/gradle/gradle/edit/master/subprojects/docs/src/docs/userguide/)

# Building Java & JVM projects

Contents

[Introduction](#4d34og8)

[Declaring your source files via source sets](#2s8eyo1)

[Managing your dependencies](#17dp8vu)

[Compiling your code](#3rdcrjn)

[Managing resources](#26in1rg)

[Running tests](#lnxbz9)

[Packaging and publishing](#35nkun2)

[Generating API documentation](#1ksv4uv)

[Cleaning the build](#44sinio)

[Building Java libraries](#2jxsxqh)

[Building Java applications](#z337ya)

[Building Java web applications](#3j2qqm3)

[Building Java EE applications](#1y810tw)

Gradle uses a convention-over-configuration approach to building JVM-based projects that borrows several conventions from Apache Maven. In particular, it uses the same default directory structure for source files and resources, and it works with Maven-compatible repositories.

We will look at Java projects in detail in this chapter, but most of the topics apply to other supported JVM languages as well, such as [Kotlin](https://guides.gradle.org/building-kotlin-jvm-libraries/), [Groovy](http://docs.google.com/groovy_plugin.html#groovy_plugin) and [Scala](http://docs.google.com/scala_plugin.html#scala_plugin). If you don’t have much experience with building JVM-based projects with Gradle, take a look at the [Java Quickstart](http://docs.google.com/tutorial_java_projects.html#tutorial_java_projects) first as it will give you a good overview of the basics.

[Introduction](#4d34og8)

The simplest build script for a Java project applies the [Java Plugin](http://docs.google.com/java_plugin.html#java_plugin) and optionally sets the project version and Java compatibility versions:

[Example: Applying the Java Plugin](#4i7ojhp)

**build.gradle**

plugins {  
 id 'java'  
}  
  
sourceCompatibility = '1.8'  
targetCompatibility = '1.8'  
version = '1.2.1'

By applying the Java Plugin, you get a whole host of features:

* A compileJava task that compiles all the Java source files under *src/main/java*
* A compileTestJava task for source files under *src/test/java*
* A test task that runs the tests from *src/test/java*
* A jar task that packages the main compiled classes and resources from *src/main/resources* into a single JAR named *<project>-<version>.jar*
* A javadoc task that generates Javadoc for the main classes

This isn’t sufficient to build any non-trivial Java project — at the very least, you’ll probably have some file dependencies. But it means that your build script only needs the information that is specific to *your* project.

| **✨** | Although the properties in the example are optional, we recommend that you specify them in your projects. The compatibility options mitigate against problems with the project being built with different Java compiler versions, and the version string is important for tracking the progression of the project. The project version is also used in archive names by default. |
| --- | --- |

The Java Plugin also integrates the above tasks into the standard [Base Plugin lifecycle tasks](http://docs.google.com/base_plugin.html#sec:base_tasks):

* jar is attached to assemble [[1](#2xcytpi)]
* test is attached to check

The rest of the chapter explains the different avenues for customizing the build to your requirements. You will also see later how to adjust the build for libraries, applications, web apps and enterprise apps.

[Declaring your source files via source sets](#2s8eyo1)

Gradle’s Java support was the first to introduce a new concept for building source-based projects: *source sets*. The main idea is that source files and resources are often logically grouped by type, such as application code, unit tests and integration tests. Each logical group typically has its own sets of file dependencies, classpaths, and more. Significantly, the files that form a source set *don’t have to be located in the same directory*!

Source sets are a powerful concept that tie together several aspects of compilation:

* the source files and where they’re located
* the compilation classpath, including any required dependencies (via Gradle [configurations](http://docs.google.com/dependency_management_terminology.html#sub:terminology_configuration))
* where the compiled class files are placed

You can see how these relate to one another in this diagram:



*Figure 1. Source sets and Java compilation*

The shaded boxes represent properties of the source set itself. On top of that, the Java Plugin automatically creates a compilation task for every source set you or a plugin defines — named compile*SourceSet*Java — and several [dependency configurations](http://docs.google.com/java_plugin.html#java_source_set_configurations).

| **✨** | The main source set  Most language plugins, Java included, automatically create a source set called main, which is used for the project’s production code. This source set is special in that its name is not included in the names of the configurations and tasks, hence why you have just a compileJava task and compileOnly and implementation configurations rather than compileMainJava, mainCompileOnly and mainImplementation respectively. |
| --- | --- |

Java projects typically include resources other than source files, such as properties files, that may need processing — for example by replacing tokens within the files — and packaging within the final JAR. The Java Plugin handles this by automatically creating a dedicated task for each defined source set called process*SourceSet*Resources (or processResources for the main source set). The following diagram shows how the source set fits in with this task:



*Figure 2. Processing non-source files for a source set*

As before, the shaded boxes represent properties of the source set, which in this case comprises the locations of the resource files and where they are copied to.

In addition to the main source set, the Java Plugin defines a test source set that represents the project’s tests. This source set is used by the test task, which runs the tests. You can learn more about this task and related topics in the [Java testing](http://docs.google.com/java_testing.html#java_testing) chapter.

Projects typically use this source set for unit tests, but you can also use it for integration, acceptance and other types of test if you wish. The alternative approach is to [define a new source set](#1ci93xb) for each of your other test types, which is typically done for one or both of the following reasons:

* You want to keep the tests separate from one another for aesthetics and manageability
* The different test types require different compilation or runtime classpaths or some other difference in setup

You can see an example of this approach in the Java testing chapter, which shows you [how to set up integration tests](http://docs.google.com/java_testing.html#sec:configuring_java_integration_tests) in a project.

You’ll learn more about source sets and the features they provide in:

* [Customizing file and directory locations](#3whwml4)
* [Configuring Java integration tests](http://docs.google.com/java_testing.html#sec:configuring_java_integration_tests)

[Managing your dependencies](#17dp8vu)

The vast majority of Java projects rely on libraries, so managing a project’s dependencies is an important part of building a Java project. Dependency management is a big topic, so we will focus on the basics for Java projects here. If you’d like to dive into the detail, check out the [introduction to dependency management](http://docs.google.com/introduction_dependency_management.html#introduction_dependency_management).

Specifying the dependencies for your Java project requires just three pieces of information:

* Which dependency you need, such as a name and version
* What it’s needed for, e.g. compilation or running
* Where to look for it

The first two are specified in a dependencies {} block and the third in a repositories {} block. For example, to tell Gradle that your project requires version 3.6.7 of [Hibernate](http://hibernate.org/) Core to compile and run your production code, and that you want to download the library from the Maven Central repository, you can use the following fragment:

[Example: Declaring dependencies](#2bn6wsx)

**build.gradle**

repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'org.hibernate:hibernate-core:3.6.7.Final'  
}

The Gradle terminology for the three elements is as follows:

* *Repository* (ex: mavenCentral()) — where to look for the modules you declare as dependencies
* *Configuration* (ex: implementation) - a named collection of dependencies, grouped together for a specific goal such as compiling or running a module — a more flexible form of Maven scopes
* *Module coordinate* (ex: org.hibernate:hibernate-core-3.6.7.Final) — the ID of the dependency, usually in the form '*<group>*:*<module>*:*<version>*' (or '*<groupId>*:*<artifactId>*:*<version>*' in Maven terminology)

You can find a more comprehensive glossary of dependency management terms [here](http://docs.google.com/dependency_management_terminology.html#dependency_management_terminology).

As far as configurations go, the main ones of interest are:

* compileOnly — for dependencies that are necessary to compile your production code but shouldn’t be part of the runtime classpath
* implementation (supersedes compile) — used for compilation and runtime
* runtimeOnly (supersedes runtime) — only used at runtime, not for compilation
* testCompileOnly — same as compileOnly except it’s for the tests
* testImplementation — test equivalent of implementation
* testRuntimeOnly — test equivalent of runtimeOnly

You can learn more about these and how they relate to one another in the [plugin reference chapter](http://docs.google.com/java_plugin.html#sec:java_plugin_and_dependency_management).

Be aware that the [Java Library Plugin](http://docs.google.com/java_library_plugin.html#java_library_plugin) creates an additional configuration — api — for dependencies that are required for compiling both the module and any modules that depend on it.

| **✨** | Why no compile configuration?  The Java Plugin has historically used the compile configuration for dependencies that are required to both compile and run a project’s production code. It is now deprecated — although it won’t be going away any time soon — because it doesn’t distinguish between dependencies that impact the public API of a Java library project and those that don’t. You can learn more about the importance of this distinction in [Building Java libraries](#2jxsxqh). |
| --- | --- |

We have only scratched the surface here, so we recommend that you read the dedicated dependency management chapters once you’re comfortable with the basics of building Java projects with Gradle. Some common scenarios that require further reading include:

* Defining a custom [Maven-](http://docs.google.com/repository_types.html#sub:maven_repo) or [Ivy-compatible](http://docs.google.com/repository_types.html#sec:ivy_repositories) repository
* Using dependencies from a [local filesystem directory](http://docs.google.com/repository_types.html#sec:flat_dir_resolver)
* Declaring dependencies with [changing](http://docs.google.com/declaring_dependencies.html#sub:declaring_dependency_with_changing_version) (e.g. SNAPSHOT) and [dynamic](http://docs.google.com/declaring_dependencies.html#sub:declaring_dependency_with_dynamic_version) (range) versions
* Declaring a sibling [project as a dependency](http://docs.google.com/declaring_dependencies.html#sec:declaring_project_dependency)
* [Controlling transitive dependencies and their versions](http://docs.google.com/managing_transitive_dependencies.html#managing_transitive_dependencies)
* Testing your fixes to a 3rd-party dependency via [composite builds](http://docs.google.com/composite_builds.html#composite_builds) (a better alternative to publishing to and consuming from [Maven Local](http://docs.google.com/repository_types.html#sub:maven_local))

You’ll discover that Gradle has a rich API for working with dependencies — one that takes time to master, but is straightforward to use for common scenarios.

[Compiling your code](#3rdcrjn)

Compiling both your production and test code can be trivially easy if you follow the conventions:

1. Put your production source code under the *src/main/java* directory
2. Put your test source code under *src/test/java*
3. Declare your production compile dependencies in the compileOnly or implementation configurations (see previous section)
4. Declare your test compile dependencies in the testCompileOnly or testImplementation configurations
5. Run the compileJava task for the production code and compileTestJava for the tests

Other JVM language plugins, such as the one for [Groovy](http://docs.google.com/groovy_plugin.html#groovy_plugin), follow the same pattern of conventions. We recommend that you follow these conventions wherever possible, but you don’t have to. There are several options for customization, as you’ll see next.

[Customizing file and directory locations](#3whwml4)

Imagine you have a legacy project that uses an *src* directory for the production code and *test* for the test code. The conventional directory structure won’t work, so you need to tell Gradle where to find the source files. You do that via source set configuration.

Each source set defines where its source code resides, along with the resources and the output directory for the class files. You can override the convention values by using the following syntax:

[Example: Declaring custom source directories](#qsh70q)

**build.gradle**

sourceSets {  
 main {  
 java {  
 srcDirs = ['src']  
 }  
 }  
  
 test {  
 java {  
 srcDirs = ['test']  
 }  
 }  
}

Now Gradle will only search directly in *src* and *test* for the respective source code. What if you don’t want to override the convention, but simply want to *add* an extra source directory, perhaps one that contains some third-party source code you want to keep separate? The syntax is similar:

[Example: Declaring custom source directories additively](#3as4poj)

**build.gradle**

sourceSets {  
 main {  
 java {  
 srcDir 'thirdParty/src/main/java'  
 }  
 }  
}

Crucially, we’re using the *method* srcDir() here to append a directory path, whereas setting the srcDirs property replaces any existing values. This is a common convention in Gradle: setting a property replaces values, while the corresponding method appends values.

You can see all the properties and methods available on source sets in the DSL reference for [SourceSet](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSet.html) and [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html). Note that srcDirs and srcDir() are both on SourceDirectorySet.

[Changing compiler options](#1pxezwc)

Most of the compiler options are accessible through the corresponding task, such as compileJava and compileTestJava. These tasks are of type [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html), so read the task reference for an up-to-date and comprehensive list of the options.

For example, if you want to use a separate JVM process for the compiler and prevent compilation failures from failing the build, you can use this configuration:

[Example: Setting Java compiler options](#49x2ik5)

**build.gradle**

compileJava {  
 options.incremental = true  
 options.fork = true  
 options.failOnError = false  
}

That’s also how you can change the verbosity of the compiler, disable debug output in the byte code and configure where the compiler can find annotation processors.

Two common options for the Java compiler are defined at the project level:

sourceCompatibility

Defines which language version of Java your source files should be treated as.

targetCompatibility

Defines the minimum JVM version your code should run on, i.e. it determines the version of byte code the compiler generates.

If you need or want more than one compilation task for any reason, you can either [create a new source set](#1ci93xb) or simply define a new task of type [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html). We look at setting up a new source set next.

[Compiling and testing Java 6/7](#2p2csry)

Gradle can only run on Java version 7 or higher. However, support for running Gradle on Java 7 has been deprecated and is scheduled to be removed in Gradle 5.0. There are two reasons for deprecating support for Java 7:

* Java 7 reached [end of life](http://www.oracle.com/technetwork/java/javase/eol-135779.html). Therefore, Oracle ceased public availability of security fixes and upgrades for Java 7 as of April 2015.
* Once support for Java 7 has ceased (likely with Gradle 5.0), Gradle’s implementation can start to use Java 8 APIs optimized for performance and usability.

Gradle still supports compiling, testing, generating Javadoc and executing applications for Java 6 and Java 7. Java 5 is not supported.

To use Java 6 or Java 7, the following tasks need to be configured:

* JavaCompile task to fork and use the correct Java home
* Javadoc task to use the correct javadoc executable
* Test and the JavaExec task to use the correct java executable.

The following sample shows how the build.gradle needs to be adjusted. In order to be able to make the build machine-independent, the location of the old Java home and target version should be configured in GRADLE\_USER\_HOME/gradle.properties [[2](#147n2zr)] in the user’s home directory on each developer machine, as shown in the example.

[Example: Configure Java 6 build](#3o7alnk)

**gradle.properties**

# in $HOME/.gradle/gradle.properties  
javaHome=/Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/Home  
targetJavaVersion=1.7

**build.gradle**

assert hasProperty('javaHome'): "Set the property 'javaHome' in your your gradle.properties pointing to a Java 6 or 7 installation"  
assert hasProperty('targetJavaVersion'): "Set the property 'targetJavaVersion' in your your gradle.properties to '1.6' or '1.7'"  
  
sourceCompatibility = targetJavaVersion  
  
def javaExecutablesPath = new File(javaHome, 'bin')  
def javaExecutables = [:].withDefault { execName ->  
 def executable = new File(javaExecutablesPath, execName)  
 assert executable.exists(): "There is no ${execName} executable in ${javaExecutablesPath}"  
 executable  
}  
tasks.withType(AbstractCompile) {  
 options.with {  
 fork = true  
 forkOptions.javaHome = file(javaHome)  
 }  
}  
tasks.withType(Javadoc) {  
 executable = javaExecutables.javadoc  
}  
tasks.withType(Test) {  
 executable = javaExecutables.java  
}  
tasks.withType(JavaExec) {  
 executable = javaExecutables.java  
}

[Compiling independent sources separately](#1ci93xb)

Most projects have at least two independent sets of sources: the production code and the test code. Gradle already makes this scenario part of its Java convention, but what if you have other sets of sources? One of the most common scenarios is when you have separate integration tests of some form or other. In that case, a custom source set may be just what you need.

You can see a complete example for setting up integration tests in the [Java testing chapter](http://docs.google.com/java_testing.html#sec:configuring_java_integration_tests). You can set up other source sets that fulfil different roles in the same way. The question then becomes: when should you define a custom source set?

To answer that question, consider whether the sources:

1. Need to be compiled with a unique classpath
2. Generate classes that are handled differently from the main and test ones
3. Form a natural part of the project

If your answer to both 3 and either one of the others is yes, then a custom source set is probably the right approach. For example, integration tests are typically part of the project because they test the code in main. In addition, they often have either their own dependencies independent of the test source set or they need to be run with a custom Test task.

Other common scenarios are less clear cut and may have better solutions. For example:

* Separate API and implementation JARs — it may make sense to have these as separate projects, particularly if you already have a multi-project build
* Generated sources — if the resulting sources should be compiled with the production code, add their path(s) to the main source set and make sure that the compileJava task depends on the task that generates the sources

If you’re unsure whether to create a custom source set or not, then go ahead and do so. It should be straightforward and if it’s not, then it’s probably not the right tool for the job.

[Managing resources](#26in1rg)

Many Java projects make use of resources beyond source files, such as images, configuration files and localization data. Sometimes these files simply need to be packaged unchanged and sometimes they need to be processed as template files or in some other way. Either way, the Java Plugin adds a specific [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) task for each source set that handles the processing of its associated resources.

The task’s name follows the convention of process*SourceSet*Resources — or processResources for the main source set — and it will automatically copy any files in *src/[sourceSet]/resources* to a directory that will be included in the production JAR. This target directory will also be included in the runtime classpath of the tests.

Since processResources is an instance of the Copy task, you can perform any of the processing described in the [Working With Files](http://docs.google.com/working_with_files.html#sec:copying_files) chapter.

[Java properties files and reproducible builds](#23ckvvd)

You can easily create Java properties files via the [WriteProperties](http://docs.google.com/dsl/org.gradle.api.tasks.WriteProperties.html) task, which fixes a well-known problem with Properties.store() that can reduce the usefulness of [incremental builds](http://docs.google.com/more_about_tasks.html#sec:up_to_date_checks).

The standard Java API for writing properties files produces a unique file every time, even when the same properties and values are used, because it includes a timestamp in the comments. Gradle’s WriteProperties task generates exactly the same output byte-for-byte if none of the properties have changed. This is achieved by a few tweaks to how a properties file is generated:

* no timestamp comment is added to the output
* the line separator is system independent, but can be configured explicitly (it defaults to '\n')
* the properties are sorted alphabetically

Sometimes it can be desirable to recreate archives in a byte for byte way on different machines. You want to be sure that building an artifact from source code produces the same result, byte for byte, no matter when and where it is built. This is necessary for projects like reproducible-builds.org.

These tweaks not only lead to better incremental build integration, but they also help with [reproducible builds](https://reproducible-builds.org). In essence, reproducible builds guarantee that you will see the same results from a build execution — including test results and production binaries — no matter when or on what system you run it.

[Running tests](#lnxbz9)

Alongside providing automatic compilation of unit tests in *src/test/java*, the Java Plugin has native support for running tests that use JUnit 3, 4 & 5 (JUnit 5 support [came in Gradle 4.6](https://docs.gradle.org/4.6/release-notes.html#junit-5-support)) and TestNG. You get:

* An automatic test task of type [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html), using the test source set
* An HTML test report that includes the results from *all* Test tasks that run
* Easy filtering of which tests to run
* Fine-grained control over how the tests are run
* The opportunity to create your own test execution and test reporting tasks

You do *not* get a Test task for every source set you declare, since not every source set represents tests! That’s why you typically need to [create your own Test tasks](#1ci93xb) for things like integration and acceptance tests if they can’t be included with the test source set.

As there is a lot to cover when it comes to testing, the topic has its [own chapter](http://docs.google.com/java_testing.html#java_testing) in which we look at:

* How tests are run
* How to run a subset of tests via filtering
* How Gradle discovers tests
* How to configure test reporting and add your own reporting tasks
* How to make use of specific JUnit and TestNG features

You can also learn more about configuring tests in the DSL reference for [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html).

[Packaging and publishing](#35nkun2)

How you package and potentially publish your Java project depends on what type of project it is. Libraries, applications, web applications and enterprise applications all have differing requirements. In this section, we will focus on the bare bones provided by the Java Plugin.

The one and only packaging feature provided by the Java Plugin directly is a jar task that packages all the compiled production classes and resources into a single JAR. This JAR is then added as an artifact — as opposed to a dependency — in the archives configuration, hence why it is automatically built by the assemble task.

If you want any other JAR or alternative archive built, you either have to apply an appropriate plugin or create the task manually. For example, if you want a task that generates a 'sources' JAR, define your own Jar task like so:

[Example: Defining a custom task to create a 'sources' JAR](#ihv636)

**build.gradle**

task sourcesJar(type: Jar) {  
 classifier = 'sources'  
 from sourceSets.main.allJava  
}

See [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html) for more details on the configuration options available to you. And note that you need to use classifier rather than appendix here for correct publication of the JAR.

If you instead want to create an 'uber' (AKA 'fat') JAR, then you can use a task definition like this:

[Example: Creating a Java uber or fat JAR](#32hioqz)

**build.gradle**

plugins {  
 id 'java'  
}  
  
version = '1.0.0'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'commons-io:commons-io:2.6'  
}  
  
task uberJar(type: Jar) {  
 appendix = 'uber'  
  
 from sourceSets.main.output  
 from configurations.runtimeClasspath.  
 findAll { it.name.endsWith('jar') }.  
 collect { zipTree(it) }  
}

There are several options for publishing a JAR once it has been created:

* the [Maven Publish Plugin](http://docs.google.com/publishing_maven.html#publishing_maven)
* the [Ivy Publish Plugin](http://docs.google.com/publishing_ivy.html#publishing_ivy)
* the uploadArchives task — the [original publishing mechanism](http://docs.google.com/artifact_management.html#artifact_management) — which works with both Ivy and (if you apply the [Maven Plugin](http://docs.google.com/maven_plugin.html#maven_plugin)) Maven

The former two "Publish" plugins are the preferred options.

[Modifying the JAR manifest](#1hmsyys)

Each instance of the Jar, War and Ear tasks has a manifest property that allows you to customize the *MANIFEST.MF* file that goes into the corresponding archive. The following example demonstrates how to set attributes in the JAR’s manifest:

[Example: Customization of MANIFEST.MF](#41mghml)

**build.gradle**

jar {  
 manifest {  
 attributes("Implementation-Title": "Gradle",  
 "Implementation-Version": version)  
 }  
}

See [Manifest](http://docs.google.com/javadoc/org/gradle/api/java/archives/Manifest.html) for the configuration options it provides.

You can also create standalone instances of Manifest. One reason for doing so is to share manifest information between JARs. The following example demonstrates how to share common attributes between JARs:

[Example: Creating a manifest object.](#2grqrue)

**build.gradle**

ext.sharedManifest = manifest {  
 attributes("Implementation-Title": "Gradle",  
 "Implementation-Version": version)  
}  
task fooJar(type: Jar) {  
 manifest = project.manifest {  
 from sharedManifest  
 }  
}

Another option available to you is to merge manifests into a single Manifest object. Those source manifests can take the form of a text for or another Manifest object. In the following example, the source manifests are all text files except for sharedManifest, which is the Manifest object from the previous example:

[Example: Separate MANIFEST.MF for a particular archive](#vx1227)

**build.gradle**

task barJar(type: Jar) {  
 manifest {  
 attributes key1: 'value1'  
 from sharedManifest, 'src/config/basemanifest.txt'  
 from('src/config/javabasemanifest.txt',  
 'src/config/libbasemanifest.txt') {  
 eachEntry { details ->  
 if (details.baseValue != details.mergeValue) {  
 details.value = baseValue  
 }  
 if (details.key == 'foo') {  
 details.exclude()  
 }  
 }  
 }  
 }  
}

Manifests are merged in the order they are declared in the from statement. If the base manifest and the merged manifest both define values for the same key, the merged manifest wins by default. You can fully customize the merge behavior by adding eachEntry actions in which you have access to a [ManifestMergeDetails](http://docs.google.com/javadoc/org/gradle/api/java/archives/ManifestMergeDetails.html) instance for each entry of the resulting manifest. Note that the merge is done lazily, either when generating the JAR or when Manifest.writeTo() or Manifest.getEffectiveManifest() are called.

Speaking of writeTo(), you can use that to easily write a manifest to disk at any time, like so:

[Example: Saving a MANIFEST.MF to disk](#3fwokq0)

**build.gradle**

jar.manifest.writeTo("$buildDir/mymanifest.mf")

[Generating API documentation](#1ksv4uv)

The Java Plugin provides a javadoc task of type [Javadoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Javadoc.html), that will generate standard Javadocs for all your production code, i.e. whatever source is in the main source set. The task supports the core Javadoc and standard doclet options described in the [Javadoc reference documentation](http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#options). See [CoreJavadocOptions](http://docs.google.com/javadoc/org/gradle/external/javadoc/CoreJavadocOptions.html) and [StandardJavadocDocletOptions](http://docs.google.com/javadoc/org/gradle/external/javadoc/StandardJavadocDocletOptions.html) for a complete list of those options.

As an example of what you can do, imagine you want to use Asciidoc syntax in your Javadoc comments. To do this, you need to add Asciidoclet to Javadoc’s doclet path. Here’s an example that does just that:

[Example: Using a custom doclet with Javadoc](#1v1yuxt)

**build.gradle**

configurations {  
 asciidoclet  
}  
  
dependencies {  
 asciidoclet 'org.asciidoctor:asciidoclet:1.+'  
}  
  
javadoc {  
 options.docletpath = configurations.asciidoclet.files.toList()  
 options.doclet = 'org.asciidoctor.Asciidoclet'  
}

You don’t have to create a configuration for this, but it’s an elegant way to handle dependencies that are required for a unique purpose.

You might also want to create your own Javadoc tasks, for example to generate API docs for the tests:

[Example: Defining a custom Javadoc task](#4f1mdlm)

**build.gradle**

task testJavadoc(type: Javadoc) {  
 source = sourceSets.test.allJava  
}

These are just two non-trivial but common customizations that you might come across.

[Cleaning the build](#44sinio)

The Java Plugin adds a clean task to your project by virtue of applying the [Base Plugin](http://docs.google.com/base_plugin.html#base_plugin). This task simply deletes everything in the $buildDir directory, hence why you should always put files generated by the build in there. The task is an instance of [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html) and you can change what directory it deletes by setting its dir property.

[Building Java libraries](#2jxsxqh)

The unique aspect of library projects is that they are used (or "consumed") by other Java projects. That means the dependency metadata published with the JAR file — usually in the form of a Maven POM —  is crucial. In particular, consumers of your library should be able to distinguish between two different types of dependencies: those that are only required to compile your library and those that are also required to compile the consumer.

Gradle manages this distinction via the [Java Library Plugin](http://docs.google.com/java_library_plugin.html#java_library_plugin), which introduces an *api* configuration in addition to the *implementation* one covered in this chapter. If the types from a dependency appear in public fields or methods of your library’s public classes, then that dependency is exposed via your library’s public API and should therefore be added to the *api* configuration. Otherwise, the dependency is an internal implementation detail and should be added to *implementation*.

| **✨** | The Java Library Plugin automatically applies the standard Java Plugin as well. |
| --- | --- |

If you’re unsure of the difference between an API and implementation dependency, the [Java Library Plugin chapter](http://docs.google.com/java_library_plugin.html#sec:java_library_recognizing_dependencies) has a detailed explanation. In addition, you can see a basic, practical example of building a Java library in the corresponding [*guide*](https://guides.gradle.org/building-java-libraries/).

[Building Java applications](#z337ya)

Java applications packaged as a JAR aren’t set up for easy launching from the command line or a desktop environment. The [Application Plugin](http://docs.google.com/application_plugin.html#application_plugin) solves the command line aspect by creating a distribution that includes the production JAR, its dependencies and launch scripts Unix-like and Windows systems.

See the plugin’s chapter for more details, but here’s a quick summary of what you get:

* assemble creates ZIP and TAR distributions of the application containing everything needed to run it
* A run task that starts the application from the build (for easy testing)
* Shell and Windows Batch scripts to start the application

Note that you will need to explicitly apply the Java Plugin in your build script.

You can see a basic example of building a Java application in the corresponding [*guide*](https://guides.gradle.org/building-java-applications/).

[Building Java web applications](#3j2qqm3)

Java web applications can be packaged and deployed in a number of ways depending on the technology you use. For example, you might use [Spring Boot](https://projects.spring.io/spring-boot/) with a fat JAR or a [Reactive](https://www.reactivemanifesto.org/)-based system running on [Netty](https://netty.io/). Whatever technology you use, Gradle and its large community of plugins will satisfy your needs. Core Gradle, though, only directly supports traditional Servlet-based web applications deployed as WAR files.

That support comes via the [War Plugin](http://docs.google.com/war_plugin.html#war_plugin), which automatically applies the Java Plugin and adds an extra packaging step that does the following:

* Copies static resources from *src/main/webapp* into the root of the WAR
* Copies the compiled production classes into a *WEB-INF/classes* subdirectory of the WAR
* Copies the library dependencies into a *WEB-INF/lib* subdirectory of the WAR

This is done by the war task, which effectively replaces the jar task — although that task remains — and is attached to the assemble lifecycle task. See the plugin’s chapter for more details and configuration options.

There is no core support for running your web application directly from the build, but we do recommend that you try the [Gretty](https://plugins.gradle.org/plugin/org.gretty) community plugin, which provides an embedded Servlet container.

[Building Java EE applications](#1y810tw)

Java enterprise systems have changed a lot over the years, but if you’re still deploying to JEE application servers, you can make use of the [Ear Plugin](http://docs.google.com/ear_plugin.html#ear_plugin). This adds conventions and a task for building EAR files. The plugin’s chapter has more details.

[1](#2u6wntf). In fact, any artifact added to the archives configuration will be built by assemble

[2](#19c6y18). For more details on gradle.properties see [Gradle configuration properties](http://docs.google.com/build_environment.html#sec:gradle_configuration_properties)

Docs

* [User Manual](http://docs.google.com/userguide/userguide.html)
* [DSL Reference](http://docs.google.com/dsl/)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Javadoc](http://docs.google.com/javadoc/)

News

* [Blog](https://blog.gradle.org/)
* [Newsletter](https://newsletter.gradle.com/)
* [Twitter](https://twitter.com/gradle)

Products

* [Build Scans](https://gradle.com/build-scans)
* [Build Cache](https://gradle.com/build-cache)
* [Enterprise Docs](https://gradle.com/enterprise/resources)

Get Help

* [Forums](https://discuss.gradle.org/c/help-discuss)
* [GitHub](https://github.com/gradle/)
* [Training](https://gradle.org/training/)
* [Services](https://gradle.org/services/)

Subscribe for important Gradle updates and news

Subscribe

By entering your email, you agree to our [Terms](https://gradle.org/terms/) and [Privacy Policy](https://gradle.org/privacy/), including receipt of emails. You can unsubscribe at any time.

© [Gradle Inc.](https://gradle.com) 2018 All rights reserved.

[Careers](https://gradle.com/careers) | [Privacy](https://gradle.org/privacy) | [Terms of Service](https://gradle.org/terms) | [Contact](https://gradle.org/contact/)