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/)

# Working With Files

Contents

[Copying a single file](#4d34og8)

[Copying multiple files](#2s8eyo1)

[Copying directory hierarchies](#17dp8vu)

[Creating archives (zip, tar, etc.)](#3rdcrjn)

[Unpacking archives](#26in1rg)

[Creating "uber" or "fat" JARs](#lnxbz9)

[Creating directories](#35nkun2)

[Moving files and directories](#1ksv4uv)

[Renaming files on copy](#44sinio)

[Deleting files and directories](#2jxsxqh)

[File paths in depth](#z337ya)

[File copying in depth](#3j2qqm3)

[Archive creation in depth](#1y810tw)

Almost every Gradle build interacts with files in some way: think source files, file dependencies, reports and so on. That’s why Gradle comes with a comprehensive API that makes it simple to perform the file operations you need.

The API has two parts to it:

* Specifying which files and directories to process
* Specifying what to do with them

The [File paths in depth](#z337ya) section covers the first of these in detail, while subsequent sections, like [File copying in depth](#3j2qqm3), cover the second. To begin with, we’ll show you examples of the most common scenarios that users encounter.

[Copying a single file](#4d34og8)

You copy a file by creating an instance of Gradle’s builtin [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) task and configuring it with the location of the file and where you want to put it. This example mimics copying a generated report into a directory that will be packed into an archive, such as a ZIP or TAR:

[Example: How to copy a single file](#4i7ojhp)

**build.gradle**

task copyReport(type: Copy) {  
 from file("${buildDir}/reports/my-report.pdf")  
 into file("${buildDir}/toArchive")  
}

The [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method is used to create a file or directory path relative to the current project and is a common way to make build scripts work regardless of the project path. The file and directory paths are then used to specify what file to copy using [Copy.from(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html#org.gradle.api.tasks.Copy:from(java.lang.Object%5B%5D)) and which directory to copy it to using [Copy.into(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html#org.gradle.api.tasks.Copy:into(java.lang.Object)).

You can even use the path directly without the file() method, as explained early in the section [File copying in depth](#3j2qqm3):

[Example: Using implicit string paths](#2xcytpi)

**build.gradle**

task copyReport2(type: Copy) {  
 from "${buildDir}/reports/my-report.pdf"  
 into "${buildDir}/toArchive"  
}

Although hard-coded paths make for simple examples, they also make the build brittle. It’s better to use a reliable, single source of truth, such as a task or shared project property. In the following modified example, we use a report task defined elsewhere that has the report’s location stored in its outputFile property:

[Example: Prefer task/project properties over hard-coded paths](#1ci93xb)

**build.gradle**

task copyReport3(type: Copy) {  
 from myReportTask.outputFile  
 into archiveReportsTask.dirToArchive  
}

We have also assumed that the reports will be archived by archiveReportsTask, which provides us with the directory that will be archived and hence where we want to put the copies of the reports.

[Copying multiple files](#2s8eyo1)

You can extend the previous examples to multiple files very easily by providing multiple arguments to from():

[Example: Using multiple arguments with from()](#3whwml4)

**build.gradle**

task copyReportsForArchiving(type: Copy) {  
 from "${buildDir}/reports/my-report.pdf", "src/docs/manual.pdf"  
 into "${buildDir}/toArchive"  
}

Two files are now copied into the archive directory. You can also use multiple from() statements to do the same thing, as shown in the first example of the section [File copying in depth](#3j2qqm3).

Now consider another example: what if you want to copy all the PDFs in a directory without having to specify each one? To do this, attach inclusion and/or exclusion patterns to the copy specification. Here we use a string pattern to include PDFs only:

[Example: Using a flat filter](#2bn6wsx)

**build.gradle**

task copyPdfReportsForArchiving(type: Copy) {  
 from "${buildDir}/reports"  
 include "\*.pdf"  
 into "${buildDir}/toArchive"  
}

One thing to note, as demonstrated in the following diagram, is that only the PDFs that reside directly in the reports directory are copied:



*Figure 1. The effect of a flat filter on copying*

You can include files in subdirectories by using an Ant-style glob pattern (\*\*/\*), as done in this updated example:

[Example: Using a deep filter](#qsh70q)

**build.gradle**

task copyAllPdfReportsForArchiving(type: Copy) {  
 from "${buildDir}/reports"  
 include "\*\*/\*.pdf"  
 into "${buildDir}/toArchive"  
}

This task has the following effect:



*Figure 2. The effect of a deep filter on copying*

One thing to bear in mind is that a deep filter like this has the side effect of copying the directory structure below reports as well as the files. If you just want to copy the files without the directory structure, you need to use an explicit fileTree(*dir*) { *includes* }.files expression. We talk more about the difference between file trees and file collections in the [File trees](#3as4poj) section.

This is just one of the variations in behavior you’re likely to come across when dealing with file operations in Gradle builds. Fortunately, Gradle provides elegant solutions to almost all those use cases. Read the *in-depth* sections later in the chapter for more detail on how the file operations work in Gradle and what options you have for configuring them.

[Copying directory hierarchies](#17dp8vu)

You may have a need to copy not just files, but the directory structure they reside in as well. This is the default behavior when you specify a directory as the from() argument, as demonstrated by the following example that copies everything in the reports directory, including all its subdirectories, to the destination:

[Example: Copying an entire directory](#1pxezwc)

**build.gradle**

task copyReportsDirForArchiving(type: Copy) {  
 from "${buildDir}/reports"  
 into "${buildDir}/toArchive"  
}

The key aspect that users struggle with is controlling how much of the directory structure goes to the destination. In the above example, do you get a toArchive/reports directory or does everything in reports go straight into toArchive? The answer is the latter. If a directory is part of the from() path, then it *won’t* appear in the destination.

So how do you ensure that reports itself is copied across, but not any other directory in $buildDir? The answer is to add it as an include pattern:

[Example: Copying an entire directory, including itself](#49x2ik5)

**build.gradle**

task copyReportsDirForArchiving2(type: Copy) {  
 from("${buildDir}") {  
 include "reports/\*\*"  
 }  
 into "${buildDir}/toArchive"  
}

You’ll get the same behavior as before except with one extra level of directory in the destination, i.e. toArchive/reports.

One thing to note is how the include() directive applies only to the from(), whereas the directive in the previous section applied to the whole task. These different levels of granularity in the copy specification allow you to easily handle most requirements that you will come across. You can learn more about this in the section on [child specifications](#2p2csry).

[Creating archives (zip, tar, etc.)](#3rdcrjn)

From the perspective of Gradle, packing files into an archive is effectively a copy in which the destination is the archive file rather than a directory on the file system. This means that creating archives looks a lot like copying, with all of the same features!

The simplest case involves archiving the entire contents of a directory, which this example demonstrates by creating a ZIP of the toArchive directory:

[Example: Archiving a directory as a ZIP](#147n2zr)

**build.gradle**

task packageDistribution(type: Zip) {  
 archiveName = "my-distribution.zip"  
 destinationDir = file("${buildDir}/dist")  
  
 from "${buildDir}/toArchive"  
}

Notice how we specify the destination and name of the archive instead of an into(): both are required. You often won’t see them explicitly set, because most projects apply the [Base Plugin](http://docs.google.com/base_plugin.html#base_plugin). It provides some conventional values for those properties. The next example demonstrates this and you can learn more about the conventions in the [archive naming](#3o7alnk) section.

Each type of archive has its own task type, the most common ones being [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html), [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html) and [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html). They all share most of the configuration options of Copy, including filtering and renaming.

One of the most common scenarios involves copying files into specified subdirectories of the archive. For example, let’s say you want to package all PDFs into a docs directory in the root of the archive. This docs directory doesn’t exist in the source location, so you have to create it as part of the archive. You do this by adding an into() declaration for just the PDFs:

[Example: Using the Base Plugin for its archive name convention](#23ckvvd)

**build.gradle**

plugins {  
 id 'base'  
}  
  
version = "1.0.0"  
  
task packageDistribution(type: Zip) {  
 from("${buildDir}/toArchive") {  
 exclude "\*\*/\*.pdf"  
 }  
  
 from("${buildDir}/toArchive") {  
 include "\*\*/\*.pdf"  
 into "docs"  
 }  
}

As you can see, you can have multiple from() declarations in a copy specification, each with its own configuration. See [Using child copy specifications](#2p2csry) for more information on this feature.

[Unpacking archives](#26in1rg)

Archives are effectively self-contained file systems, so unpacking them is a case of copying the files from that file system onto the local file system — or even into another archive. Gradle enables this by providing some wrapper functions that make archives available as hierarchical collections of files ([file trees](#3as4poj)).

The two functions of interest are [Project.zipTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:zipTree(java.lang.Object)) and [Project.tarTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:tarTree(java.lang.Object)), which produce a [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) from a corresponding archive file. That file tree can then be used in a from() specification, like so:

[Example: Unpacking a ZIP file](#ihv636)

**build.gradle**

task unpackFiles(type: Copy) {  
 from zipTree("src/resources/thirdPartyResources.zip")  
 into "${buildDir}/resources"  
}

As with a normal copy, you can control which files are unpacked via filters and even rename files as they are unpacked.

If you’re a Java developer and are wondering why there is no jarTree() method, that’s because zipTree() works perfectly well for JARs, WARs and EARs.

[Creating "uber" or "fat" JARs](#lnxbz9)

In the Java space, applications and their dependencies typically used to be packaged as separate JARs within a single distribution archive. That still happens, but there is another approach that is now common: placing the classes and resources of the dependencies directly into the application JAR, creating what is known as an uber or fat JAR.

Gradle makes this approach easy to accomplish. Consider the aim: to copy the contents of other JAR files into the application JAR. All you need for this is the [Project.zipTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:zipTree(java.lang.Object)) method and the [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html) task, as demonstrated by the uberJar task in the following example:

[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) }  
}

In this case, we’re taking the runtime dependencies of the project — configurations.runtimeClasspath.files — and wrapping each of the JAR files with the zipTree() method. The result is a collection of ZIP file trees, the contents of which are copied into the uber JAR alongside the application classes.

[Creating directories](#35nkun2)

Many tasks need to create directories to store the files they generate, which is why Gradle automatically manages this aspect of tasks when they explicitly define file and directory outputs. You can learn about this feature in the [incremental build](http://docs.google.com/more_about_tasks.html#sec:up_to_date_checks) section of the user guide. All core Gradle tasks ensure that any output directories they need are created if necessary using this mechanism.

In cases where you need to create a directory manually, you can use the [Project.mkdir(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:mkdir(java.lang.Object)) method from within your build scripts or custom task implementations. Here’s a simple example that creates a single images directory in the project folder:

[Example: Manually creating a directory](#1hmsyys)

**build.gradle**

task ensureDirectory {  
 doLast {  
 mkdir "images"  
 }  
}

As described in the [Apache Ant manual](https://ant.apache.org/manual/Tasks/mkdir.html), the mkdir task will automatically create all necessary directories in the given path and will do nothing if the directory already exists.

[Moving files and directories](#1ksv4uv)

Gradle has no API for moving files and directories around, but you can use the [Apache Ant integration](http://docs.google.com/ant.html#ant) to easily do that, as shown in this example:

[Example: Moving a directory using the Ant task](#41mghml)

**build.gradle**

task moveReports {  
 doLast {  
 ant.move file: "${buildDir}/reports",  
 todir: "${buildDir}/toArchive"  
 }  
}

This is not a common requirement and should be used sparingly as you lose information and can easily break a build. It’s generally preferable to copy directories and files instead.

[Renaming files on copy](#44sinio)

The files used and generated by your builds sometimes don’t have names that suit, in which case you want to rename those files as you copy them. Gradle allows you to do this as part of a copy specification using the rename() configuration.

The following example removes the "-staging-" marker from the names of any files that have it:

[Example: Renaming files as they are copied](#2grqrue)

**build.gradle**

task copyFromStaging(type: Copy) {  
 from "src/main/webapp"  
 into "${buildDir}/explodedWar"  
  
 rename '(.+)-staging(.+)', '$1$2'  
}

You can use regular expressions for this, as in the above example, or closures that use more complex logic to determine the target filename. For example, the following task truncates filenames:

[Example: Truncating filenames as they are copied](#vx1227)

**build.gradle**

task copyWithTruncate(type: Copy) {  
 from "${buildDir}/reports"  
 rename { String filename ->  
 if (filename.size() > 10) {  
 return filename[0..7] + "~" + filename.size()  
 }  
 else return filename  
 }  
 into "${buildDir}/toArchive"  
}

As with filtering, you can also apply renaming to a subset of files by configuring it as part of a child specification on a from().

[Deleting files and directories](#2jxsxqh)

You can easily delete files and directories using either the [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html) task or the [Project.delete(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:delete(org.gradle.api.Action)) method. In both cases, you specify which files and directories to delete in a way supported by the [Project.files(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), and [ProjectLayout.configurableFiles(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-) methods.

For example, the following task deletes the entire contents of a build’s output directory:

[Example: Deleting a directory](#3fwokq0)

**build.gradle**

task myClean(type: Delete) {  
 delete buildDir  
}

If you want more control over which files are deleted, you can’t use inclusions and exclusions in the same way as for copying files. Instead, you have to use the builtin filtering mechanisms of FileCollection and FileTree. The following example does just that to clear out temporary files from a source directory:

[Example: Deleting files matching a specific pattern](#1v1yuxt)

**build.gradle**

task cleanTempFiles(type: Delete) {  
 delete fileTree("src").matching {  
 include "\*\*/\*.tmp"  
 }  
}

You’ll learn more about file collections and file trees in the next section.

[File paths in depth](#z337ya)

In order to perform some action on a file, you need to know where it is, and that’s the information provided by file paths. Gradle builds on the standard Java [File](https://docs.oracle.com/javase/7/docs/api/java/io/File.html) class, which represents the location of a single file, and provides new APIs for dealing with collections of paths. This section shows you how to use the Gradle APIs to specify file paths for use in tasks and file operations.

But first, an important note on using hard-coded file paths in your builds.

[On hard-coded file paths](#4f1mdlm)

Many examples in this chapter use hard-coded paths as string literals. This makes them easy to understand, but it’s not good practice for real builds. The problem is that paths often change and the more places you need to change them, the more likely you are to miss one and break the build.

Where possible, you should use tasks, task properties, and [project properties](http://docs.google.com/writing_build_scripts.html#sec:extra_properties) — in that order of preference — to configure file paths. For example, if you were to create a task that packages the compiled classes of a Java application, you should aim for something like this:

[Example: How to minimize the number of hard-coded paths in your build](#2u6wntf)

**build.gradle**

ext {  
 archivesDirPath = "${buildDir}/archives"  
}  
  
task packageClasses(type: Zip) {  
 appendix = "classes"  
 destinationDir = file(archivesDirPath)  
  
 from compileJava  
}

See how we’re using the compileJava task as the source of the files to package and we’ve created a project property archivesDirPath to store the location where we put archives, on the basis we’re likely to use it elsewhere in the build.

Using a task directly as an argument like this relies on it having [defined outputs](http://docs.google.com/more_about_tasks.html#sec:task_inputs_outputs), so it won’t always be possible. In addition, this example could be improved further by relying on the Java plugin’s convention for destinationDir rather than overriding it, but it does demonstrate the use of project properties.

[Single files and directories](#19c6y18)

Gradle provides the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method for specifying the location of a single file or directory. Relative paths are resolved relative to the project directory, while absolute paths remain unchanged.

| **⚠** | Never use new File(relative path), as this creates a path relative to the current working directory, which could be anywhere. |
| --- | --- |

Here are some examples of using the file() method with different types of argument:

[Example: Locating files](#3tbugp1)

**build.gradle**

// Using a relative path  
File configFile = file('src/config.xml')  
  
// Using an absolute path  
configFile = file(configFile.absolutePath)  
  
// Using a File object with a relative path  
configFile = file(new File('src/config.xml'))  
  
// Using a java.nio.file.Path object with a relative path  
configFile = file(Paths.get('src', 'config.xml'))  
  
// Using an absolute java.nio.file.Path object  
configFile = file(Paths.get(System.getProperty('user.home')).resolve('global-config.xml'))

As you can see, you can pass strings, File instances and [Path](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html) instances to the file() method, all of which result in an absolute File object. You can find other options for argument types in the reference guide, linked in the previous paragraph.

What happens in the case of multi-project builds? The file() method will always turn relative paths into paths that are relative to the current project directory, which may be a child project. If you want to use a path that’s relative to the *root project* directory, then you need to use the special [Project.getRootDir()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:rootDir) property to construct an absolute path, like so:

[Example: Creating a path relative to a parent project](#28h4qwu)

**build.gradle**

File configFile = file("${rootDir}/shared/config.xml")

Let’s say you’re working on a multi-project build in a dev/projects/AcmeHealth directory. You use the above example in the build of the library you’re fixing — at AcmeHealth/subprojects/AcmePatientRecordLib/build.gradle. The file path will resolve to the absolute version of dev/projects/AcmeHealth/shared/config.xml.

The file() method can be used to configure any task that has a property of type File. Many tasks, though, work on multiple files, so we look at how to specify sets of files next.

[File collections](#nmf14n)

A *file collection* is simply a set of file paths that’s represented by the [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) interface. *Any* file paths. It’s important to understand that the file paths don’t have to be related in any way, so they don’t have to be in the same directory or even have a shared parent directory. You will also find that many parts of the Gradle API use FileCollection, such as the copying API discussed later in this chapter and [dependency configurations](http://docs.google.com/managing_dependency_configurations.html#managing_dependency_configurations).

The recommended way to specify a collection of files is to use the link:../javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-[ProjectLayout.files(java.lang.Object...) method, which returns a FileCollection instance. This method is very flexible and allows you to pass multiple strings, File instances, collections of strings, collections of Files, and more. You can even pass in tasks as arguments if they have [defined outputs](http://docs.google.com/more_about_tasks.html#sec:task_inputs_outputs). Learn about all the supported argument types in the reference guide.

As with the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method covered in the [previous section](#19c6y18), all relative paths are evaluated relative to the current project directory. The following example demonstrates some of the variety of argument types you can use — strings, File instances, a list and a [Path](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html):

[Example: Creating a file collection](#37m2jsg)

**build.gradle**

FileCollection collection = layout.files('src/file1.txt',  
 new File('src/file2.txt'),  
 ['src/file3.csv', 'src/file4.csv'],  
 Paths.get('src', 'file5.txt'))

File collections have some important attributes in Gradle. They can be:

* created lazily
* iterated over
* filtered
* combined

*Lazy creation* of a file collection is useful when you need to evaluate the files that make up a collection at the time a build runs. In the following example, we query the file system to find out what files exist in a particular directory and then make those into a file collection:

[Example: Implementing a file collection](#1mrcu09)

**build.gradle**

task list {  
 doLast {  
 File srcDir  
  
 // Create a file collection using a closure  
 collection = layout.files { srcDir.listFiles() }  
  
 srcDir = file('src')  
 println "Contents of $srcDir.name"  
 collection.collect { relativePath(it) }.sort().each { println it }  
  
 srcDir = file('src2')  
 println "Contents of $srcDir.name"  
 collection.collect { relativePath(it) }.sort().each { println it }  
 }  
}

**Output of** gradle -q list

> gradle -q list  
Contents of src  
src/dir1  
src/file1.txt  
Contents of src2  
src2/dir1  
src2/dir2

The key to lazy creation is passing a closure to the files() method. Your closure simply needs to return a value of a type accepted by files(), such as List<File>, String, FileCollection, etc.

*Iterating over a file collection* can be done through the each() method on the collection or using the collection in a for loop. In both approaches, the file collection is treated as a set of File instances, i.e. your iteration variable will be of type File.

The following example demonstrates such iteration as well as how you can convert file collections to other types using the as operator or supported properties:

[Example: Using a file collection](#46r0co2)

**build.gradle**

// Iterate over the files in the collection  
 collection.each { File file ->  
 println file.name  
 }  
  
 // Convert the collection to various types  
 Set set = collection.files  
 Set set2 = collection as Set  
 List list = collection as List  
 String path = collection.asPath  
 File file = collection.singleFile  
 File file2 = collection as File  
  
 // Add and subtract collections  
 def union = collection + layout.files('src/file2.txt')  
 def difference = collection - layout.files('src/file2.txt')

You can also see at the end of the example *how to combine file collections* using the + and - operators to merge and subtract them. An important feature of the resulting file collections is that they are *live*. In other words, when you combine file collections in this way, the result always reflects what’s currently in the source file collections, even if they change during the build.

For example, imagine collection in the above example gains an extra file or two after union is created. As long as you use union after those files are added to collection, union will also contain those additional files. The same goes for the different file collection.

Live collections are also important when it comes to *filtering*. If you want to use a subset of a file collection, you can take advantage of the [FileCollection.filter(org.gradle.api.specs.Spec)](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html#filter-org.gradle.api.specs.Spec-) method to determine which files to "keep". In the following example, we create a new collection that consists of only the files that end with .txt in the source collection:

[Example: Filtering a file collection](#2lwamvv)

**build.gradle**

FileCollection textFiles = collection.filter { File f ->  
 f.name.endsWith(".txt")  
}

**Output of** gradle -q filterTextFiles

> gradle -q filterTextFiles  
src/file1.txt  
src/file2.txt  
src/file5.txt

If collection changes at any time, either by adding or removing files from itself, then textFiles will immediately reflect the change because it is also a live collection. Note that the closure you pass to filter() takes a File as an argument and should return a boolean.

[File trees](#3as4poj)

A *file tree* is a file collection that retains the directory structure of the files it contains and has the type [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html). This means that all the paths in a file tree must have a shared parent directory. The following diagram highlights the distinction between file trees and file collections in the common case of copying files:



*Figure 3. The differences in how file trees and file collections behave when copying files*

| **✨** | Although FileTree extends FileCollection (an is-a relationship), their behaviors do differ. In other words, you can use a file tree wherever a file collection is required, but remember: a file collection is a flat list/set of files, while a file tree is a file and directory hierarchy. To convert a file tree to a flat collection, use the [FileTree.getFiles()](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html#getFiles--) property. |
| --- | --- |

The simplest way to create a file tree is to pass a file or directory path to the [Project.fileTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:fileTree(java.lang.Object)) method. This will create a tree of all the files and directories in that base directory (but not the base directory itself). The following example demonstrates how to use the basic method and, in addition, how to filter the files and directories using Ant-style patterns:

[Example: Creating a file tree](#111kx3o)

**build.gradle**

// Create a file tree with a base directory  
FileTree tree = fileTree(dir: 'src/main')  
  
// Add include and exclude patterns to the tree  
tree.include '\*\*/\*.java'  
tree.exclude '\*\*/Abstract\*'  
  
// Create a tree using path  
tree = fileTree('src').include('\*\*/\*.java')  
  
// Create a tree using closure  
tree = fileTree('src') {  
 include '\*\*/\*.java'  
}  
  
// Create a tree using a map  
tree = fileTree(dir: 'src', include: '\*\*/\*.java')  
tree = fileTree(dir: 'src', includes: ['\*\*/\*.java', '\*\*/\*.xml'])  
tree = fileTree(dir: 'src', include: '\*\*/\*.java', exclude: '\*\*/\*test\*/\*\*')

You can see more examples of supported patterns in the API docs for [PatternFilterable](http://docs.google.com/javadoc/org/gradle/api/tasks/util/PatternFilterable.html). Also, see the API documentation for fileTree() to see what types you can pass as the base directory.

By default, fileTree() returns a FileTree instance that applies some default exclusion patterns for convenience — the same defaults as Ant in fact. For the complete default exclusion list, see [the Ant manual](http://ant.apache.org/manual/dirtasks.html#defaultexcludes).

If those default exclusions prove problematic, you can workaround the issue by using the [defaultexcludes Ant task](https://ant.apache.org/manual/Tasks/defaultexcludes.html), as demonstrated in this example:

[Example: Changing Ant default exclusions for a copy task](#3l18frh)

**build.gradle**

task forcedCopy(type: Copy) {  
 into "${buildDir}/inPlaceApp"  
 from 'src/main/webapp'  
  
 doFirst {  
 ant.defaultexcludes remove: "\*\*/.git"  
 ant.defaultexcludes remove: "\*\*/.git/\*\*"  
 ant.defaultexcludes remove: "\*\*/\*~"  
 }  
  
 doLast {  
 ant.defaultexcludes default: true  
 }  
}

In general, it’s best to ensure that the default exclusions are reset whenever you change them as modifications are visible to the entire build. The above example is performing such a reset in its doLast action.

You can do many of the same things with file trees that you can with file collections:

* iterate over them (depth first)
* filter them (using [FileTree.matching(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html#matching-org.gradle.api.Action-) and Ant-style patterns)
* merge them

You can also traverse file trees using the [FileTree.visit(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html#visit-org.gradle.api.Action-) method. All of these techniques are demonstrated in the following example:

[Example: Using a file tree](#206ipza)

**build.gradle**

// Iterate over the contents of a tree  
tree.each {File file ->  
 println file  
}  
  
// Filter a tree  
FileTree filtered = tree.matching {  
 include 'org/gradle/api/\*\*'  
}  
  
// Add trees together  
FileTree sum = tree + fileTree(dir: 'src/test')  
  
// Visit the elements of the tree  
tree.visit {element ->  
 println "$element.relativePath => $element.file"  
}

We’ve discussed how to create your own file trees and file collections, but it’s also worth bearing in mind that many Gradle plugins provide their own instances of file trees, such as [Java’s source sets](http://docs.google.com/building_java_projects.html#sec:java_source_sets). These can be used and manipulated in exactly the same way as the file trees you create yourself.

Another specific type of file tree that users commonly need is the archive, i.e. ZIP files, TAR files, etc. We look at those next.

[Using archives as file trees](#4k668n3)

An archive is a directory and file hierarchy packed into a single file. In other words, it’s a special case of a file tree, and that’s exactly how Gradle treats archives. Instead of using the fileTree() method, which only works on normal file systems, you use the [Project.zipTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:zipTree(java.lang.Object)) and [Project.tarTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:tarTree(java.lang.Object)) methods to wrap archive files of the corresponding type (note that JAR, WAR and EAR files are ZIPs). Both methods return FileTree instances that you can then use in the same way as normal file trees. For example, you can extract some or all of the files of an archive by copying its contents to some directory on the file system. Or you can merge one archive into another.

Here are some simple examples of creating archive-based file trees:

[Example: Using an archive as a file tree](#2zbgiuw)

**build.gradle**

// Create a ZIP file tree using path  
FileTree zip = zipTree('someFile.zip')  
  
// Create a TAR file tree using path  
FileTree tar = tarTree('someFile.tar')  
  
//tar tree attempts to guess the compression based on the file extension  
//however if you must specify the compression explicitly you can:  
FileTree someTar = tarTree(resources.gzip('someTar.ext'))

You can see a practical example of extracting an archive file [in among the common scenarios](#26in1rg) we cover.

[Understanding implicit conversion to file collections](#1egqt2p)

Many objects in Gradle have properties which accept a set of input files. For example, the [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html) task has a source property that defines the source files to compile. You can set the value of this property using any of the types supported by the [files()](#nmf14n) method, as mentioned in the api docs. This means you can, for example, set the property to a File, String, collection, FileCollection or even a closure.

This is a feature of specific tasks! That means implicit conversion will not happen for just any task that has a FileCollection or FileTree property. If you want to know whether implicit conversion happens in a particular situation, you will need to read the relevant documentation, such as the corresponding task’s API docs. Alternatively, you can remove all doubt by explicitly using link:../javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-[ProjectLayout.files(java.lang.Object...) in your build.

Here are some examples of the different types of arguments that the source property can take:

[Example: Specifying a set of files](#3ygebqi)

**build.gradle**

task compile(type: JavaCompile)  
  
// Use a File object to specify the source directory  
compile {  
 source = file('src/main/java')  
}  
  
// Use a String path to specify the source directory  
compile {  
 source = 'src/main/java'  
}  
  
// Use a collection to specify multiple source directories  
compile {  
 source = ['src/main/java', '../shared/java']  
}  
  
// Use a FileCollection (or FileTree in this case) to specify the source files  
compile {  
 source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/\*\*' }  
}  
  
// Using a closure to specify the source files.  
compile {  
 source = {  
 // Use the contents of each zip file in the src dir  
 file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect { zipTree(it) }  
 }  
}

One other thing to note is that properties like source have corresponding methods in core Gradle tasks. Those methods follow the convention of *appending* to collections of values rather than replacing them. Again, this method accepts any of the types supported by the [files()](#nmf14n) method, as shown here:

[Example: Appending a set of files](#2dlolyb)

**build.gradle**

compile {  
 // Add some source directories use String paths  
 source 'src/main/java', 'src/main/groovy'  
  
 // Add a source directory using a File object  
 source file('../shared/java')  
  
 // Add some source directories using a closure  
 source { file('src/test/').listFiles() }  
}

As this is a common convention, we recommend that you follow it in your own custom tasks. Specifically, if you plan to add a method to configure a collection-based property, make sure the method appends rather than replaces values.

[File copying in depth](#3j2qqm3)

The basic process of copying files in Gradle is a simple one:

* Define a task of type [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)
* Specify which files (and potentially directories) to copy
* Specify a destination for the copied files

But this apparent simplicity hides a rich API that allows fine-grained control of which files are copied, where they go, and what happens to them as they are copied — renaming of the files and token substitution of file content are both possibilities, for example.

Let’s start with the last two items on the list, which form what is known as a *copy specification*. This is formally based on the [CopySpec](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html) interface, which the Copy task implements, and offers:

* A [CopySpec.from(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#from-java.lang.Object...-) method to define what to copy
* An [CopySpec.into(java.lang.Object)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#into-java.lang.Object-) method to define the destination

CopySpec has several additional methods that allow you to control the copying process, but these two are the only required ones. into() is straightforward, requiring a directory path as its argument in any form supported by the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method. The from() configuration is far more flexible.

Not only does from() accept multiple arguments, it also allows several different types of argument. For example, some of the most common types are:

* A String — treated as a file path or, if it starts with "file://", a file URI
* A File — used as a file path
* A FileCollection or FileTree — all files in the collection are included in the copy
* A task — the files or directories that form a task’s [defined outputs](http://docs.google.com/more_about_tasks.html#sec:task_inputs_outputs) are included

In fact, from() accepts all the same arguments as [Project.files(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), and [ProjectLayout.configurableFiles(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-), so see those methods for a more detailed list of acceptable types.

Something else to consider is what type of thing a file path refers to:

* A file — the file is copied as is
* A directory — this is effectively treated as a file tree: everything in it, including subdirectories, is copied. However, the directory itself is not included in the copy.
* A non-existent file — the path is ignored

Here is an example that uses multiple from() specifications, each with a different argument type. You will probably also notice that into() is configured lazily using a closure — a technique that also works with from():

[Example: Specifying copy task source files and destination directory](#sqyw64)

**build.gradle**

task anotherCopyTask(type: Copy) {  
 // Copy everything under src/main/webapp  
 from 'src/main/webapp'  
 // Copy a single file  
 from 'src/staging/index.html'  
 // Copy the output of a task  
 from copyTask  
 // Copy the output of a task using Task outputs explicitly.  
 from copyTaskWithPatterns.outputs  
 // Copy the contents of a Zip file  
 from zipTree('src/main/assets.zip')  
 // Determine the destination directory later  
 into { getDestDir() }  
}

Note that the lazy configuration of into() is different from a [child specification](#2p2csry), even though the syntax is similar. Keep an eye on the number of arguments to distinguish between them.

[Filtering files](#3cqmetx)

You’ve already seen that you can filter file collections and file trees directly in a Copy task, but you can also apply filtering in any copy specification through the [CopySpec.include(java.lang.String…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#include-java.lang.String...-) and [CopySpec.exclude(java.lang.String…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#exclude-java.lang.String...-) methods.

Both of these methods are normally used with Ant-style include or exclude patterns, as described in [PatternFilterable](http://docs.google.com/javadoc/org/gradle/api/tasks/util/PatternFilterable.html). You can also perform more complex logic by using a closure that takes a [FileTreeElement](http://docs.google.com/javadoc/org/gradle/api/file/FileTreeElement.html) and returns true if the file should be included or false otherwise. The following example demonstrates both forms, ensuring that only .html and .jsp files are copied, except for those .html files with the word "DRAFT" in their content:

[Example: Selecting the files to copy](#1rvwp1q)

**build.gradle**

task copyTaskWithPatterns(type: Copy) {  
 from 'src/main/webapp'  
 into "${buildDir}/explodedWar"  
 include '\*\*/\*.html'  
 include '\*\*/\*.jsp'  
 exclude { FileTreeElement details ->  
 details.file.name.endsWith('.html') &&  
 details.file.text.contains('DRAFT')  
 }  
}

A question you may ask yourself at this point is what happens when inclusion and exclusion patterns overlap? Which pattern wins? Here are the basic rules:

* If there are no explicit inclusions or exclusions, everything is included
* If at least one inclusion is specified, only files and directories matching the patterns are included
* Any exclusion pattern overrides any inclusions, so if a file or directory matches at least one exclusion pattern, it won’t be included, regardless of the inclusion patterns

Bear these rules in mind when creating combined inclusion and exclusion specifications so that you end up with the exact behavior you want.

Note that the inclusions and exclusions in the above example will apply to *all* from() configurations. If you want to apply filtering to a subset of the copied files, you’ll need to use [child specifications](#2p2csry).

[Renaming files](#4bvk7pj)

The [example of how to rename files on copy](#44sinio) gives you most of the information you need to perform this operation. It demonstrates the two options for renaming:

* Using a regular expression
* Using a closure

Regular expressions are a flexible approach to renaming, particularly as Gradle supports regex groups that allow you to remove and replaces parts of the source filename. The following example shows how you can remove the string "-staging-" from any filename that contains it using a simple regular expression:

[Example: Renaming files as they are copied](#2r0uhxc)

**build.gradle**

task rename(type: Copy) {  
 from 'src/main/webapp'  
 into "${buildDir}/explodedWar"  
 // Use a closure to convert all file names to upper case  
 rename { String fileName ->  
 fileName.toUpperCase()  
 }  
 // Use a regular expression to map the file name  
 rename '(.+)-staging-(.+)', '$1$2'  
 rename(/(.+)-staging-(.+)/, '$1$2')  
}

You can use any regular expression supported by the Java [Pattern](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html) class and the substitution string (the second argument of rename() works on the same principles as the [Matcher.appendReplacement()](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#appendReplacement(java.lang.StringBuffer,%20java.lang.String)) method.

| **✨** | Regular expressions in Groovy build scripts  There are two common issues people come across when using regular expressions in this context:   1. If you use a slashy string (those delimited by '/') for the first argument, you *must* include the parentheses for rename() as shown in the above example. 2. It’s safest to use single quotes for the second argument, otherwise you need to escape the '$' in group substitutions, i.e. "\$1\$2"   The first is a minor inconvenience, but slashy strings have the advantage that you don’t have to escape backslash ('\') characters in the regular expression. The second issue stems from Groovy’s support for embedded expressions using ${ } syntax in double-quoted and slashy strings. |
| --- | --- |

The closure syntax for rename() is straightforward and can be used for any requirements that simple regular expressions can’t handle. You’re given the name of a file and you return a new name for that file, or null if you don’t want to change the name. Do be aware that the closure will be executed for every file that’s copied, so try to avoid expensive operations where possible.

[Filtering file content (token substitution, templating, etc.)](#1664s55)

Not to be confused with filtering which files are copied, *file content filtering* allows you to transform the content of files while they are being copied. This can involve basic templating that uses token substitution, removal of lines of text, or even more complex filtering using a full-blown template engine.

The following example demonstrates several forms of filtering, including token substitution using the [CopySpec.expand(java.util.Map)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#expand-java.util.Map-) method and another using [CopySpec.filter(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#filter-java.lang.Class-) with an [Ant filter](https://ant.apache.org/manual/Types/filterchain.html):

[Example: Filtering files as they are copied](#3q5sasy)

**build.gradle**

import org.apache.tools.ant.filters.FixCrLfFilter  
import org.apache.tools.ant.filters.ReplaceTokens  
  
task filter(type: Copy) {  
 from 'src/main/webapp'  
 into "${buildDir}/explodedWar"  
 // Substitute property tokens in files  
 expand(copyright: '2009', version: '2.3.1')  
 expand(project.properties)  
 // Use some of the filters provided by Ant  
 filter(FixCrLfFilter)  
 filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])  
 // Use a closure to filter each line  
 filter { String line ->  
 "[$line]"  
 }  
 // Use a closure to remove lines  
 filter { String line ->  
 line.startsWith('-') ? null : line  
 }  
 filteringCharset = 'UTF-8'  
}

The filter() method has two variants, which behave differently:

* one takes a [FilterReader](https://docs.oracle.com/javase/7/docs/api/java/io/FilterReader.html) and is designed to work with Ant filters, such as ReplaceTokens
* one takes a closure or [Transformer](http://docs.google.com/javadoc/org/gradle/api/Transformer.html) that defines the transformation for each line of the source file

Note that both variants assume the source files are text based. When you use the ReplaceTokens class with filter(), the result is a template engine that replaces tokens of the form @tokenName@ (the Ant-style token) with values that you define.

The expand() method treats the source files as [Groovy templates](http://docs.groovy-lang.org/latest/html/api/groovy/text/SimpleTemplateEngine.html), which evaluate and expand expressions of the form ${expression}. You can pass in property names and values that are then expanded in the source files. expand() allows for more than basic token substitution as the embedded expressions are full-blown Groovy expressions.

| **✨** | It’s good practice to specify the character set when reading and writing the file, otherwise the transformations won’t work properly for non-ASCII text. You configure the character set with the [CopySpec.getFilteringCharset()](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#getFilteringCharset--) property. If it’s not specified, the JVM default character set is used, which is likely to be different from the one you want. |
| --- | --- |

[Using the CopySpec class](#25b2l0r)

A copy specification (or copy spec for short) determines what gets copied to where, and what happens to files during the copy. You’ve alread seen many examples in the form of configuration for Copy and archiving tasks. But copy specs have two attributes that are worth covering in more detail:

1. They can be independent of tasks
2. They are hierarchical

The first of these attributes allows you to *share copy specs within a build*. The second provides fine-grained control within the overall copy specification.

[Sharing copy specs](#kgcv8k)

Consider a build that has several tasks that copy a project’s static website resources or add them to an archive. One task might copy the resources to a folder for a local HTTP server and another might package them into a distribution. You could manually specify the file locations and appropriate inclusions each time they are needed, but human error is more likely to creep in, resulting in inconsistencies between tasks.

One solution Gradle provides is the [Project.copySpec(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copySpec(org.gradle.api.Action)) method. This allows you to create a copy spec outside of a task, which can then be attached to an appropriate task using the [CopySpec.with(org.gradle.api.file.CopySpec…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#with-org.gradle.api.file.CopySpec...-) method. The following example demonstrates how this is done:

[Example: Sharing copy specifications](#34g0dwd)

**build.gradle**

CopySpec webAssetsSpec = copySpec {  
 from 'src/main/webapp'  
 include '\*\*/\*.html', '\*\*/\*.png', '\*\*/\*.jpg'  
 rename '(.+)-staging(.+)', '$1$2'  
}  
  
task copyAssets(type: Copy) {  
 into "${buildDir}/inPlaceApp"  
 with webAssetsSpec  
}  
  
task distApp(type: Zip) {  
 archiveName = 'my-app-dist.zip'  
 destinationDir = file("${buildDir}/dists")  
  
 from appClasses  
 with webAssetsSpec  
}

Both the copyAssets and distApp tasks will process the static resources under src/main/webapp, as specified by webAssetsSpec.

| **✨** | The configuration defined by webAssetsSpec will *not* apply to the app classes included by the distApp task. That’s because from appClasses is its own child specification independent of with webAssetsSpec.  This can be confusing to understand, so it’s probably best to treat with() as an extra from() specification in the task. Hence it doesn’t make sense to define a standalone copy spec without at least one from() defined. |
| --- | --- |

If you encounter a scenario in which you want to apply the same copy configuration to *different* sets of files, then you can share the configuration block directly without using copySpec(). Here’s an example that has two independent tasks that happen to want to process image files only:

[Example: Sharing copy patterns only](#1jlao46)

**build.gradle**

def webAssetPatterns = {  
 include '\*\*/\*.html', '\*\*/\*.png', '\*\*/\*.jpg'  
}  
  
task copyAppAssets(type: Copy) {  
 into "${buildDir}/inPlaceApp"  
 from 'src/main/webapp', webAssetPatterns  
}  
  
task archiveDistAssets(type: Zip) {  
 archiveName = 'distribution-assets.zip'  
 destinationDir = file("${buildDir}/dists")  
  
 from 'distResources', webAssetPatterns  
}

In this case, we assign the copy configuration to its own variable and apply it to whatever from() specification we want. This doesn’t just work for inclusions, but also exclusions, file renaming, and file content filtering.

[Using child specifications](#2p2csry)

If you only use a single copy spec, the file filtering and renaming will apply to *all* the files that are copied. Sometimes this is what you want, but not always. Consider the following example that copies files into a directory structure that can be used by a Java Servlet container to deliver a website:



*Figure 4. Creating an exploded WAR for a Servlet container*

This is not a straightforward copy as the WEB-INF directory and its subdirectories don’t exist within the project, so they must be created during the copy. In addition, we only want HTML and image files going directly into the root folder — build/explodedWar — and only JavaScript files going into the js directory. So we need separate filter patterns for those two sets of files.

The solution is to use *child specifications*, which can be applied to both from() and into() declarations. The following task definition does the necessary work:

[Example: Nested copy specs](#43ky6rz)

**build.gradle**

task nestedSpecs(type: Copy) {  
 into "${buildDir}/explodedWar"  
 exclude '\*\*/\*staging\*'  
 from('src/dist') {  
 include '\*\*/\*.html', '\*\*/\*.png', '\*\*/\*.jpg'  
 }  
 from(sourceSets.main.output) {  
 into 'WEB-INF/classes'  
 }  
 into('WEB-INF/lib') {  
 from configurations.runtimeClasspath  
 }  
}

Notice how the src/dist configuration has a nested inclusion specification: that’s the child copy spec. You can of course add content filtering and renaming here as required. A child copy spec is still a copy spec.

The above example also demonstrates how you can copy files into a subdirectory of the destination either by using a child into() on a from() or a child from() on an into(). Both approaches are acceptable, but you may want to create and follow a convention to ensure consistency across your build files.

| **✨** | Don’t get your into() specifications mixed up! For a normal copy — one to the filesystem rather than an archive — there should always be *one* "root" into() that simply specifies the overall destination directory of the copy. Any other into() should have a child spec attached and its path will be relative to the root into(). |
| --- | --- |

One final thing to be aware of is that a child copy spec inherits its destination path, include patterns, exclude patterns, copy actions, name mappings and filters from its parent. So be careful where you place your configuration.

[Copying files in your own tasks](#2iq8gzs)

There might be occasions when you want to copy files or directories as *part* of a task. For example, a custom archiving task based on an unsupported archive format might want to copy files to a temporary directory before they are then archived. You still want to take advantage of Gradle’s copy API, but without introducing an extra Copy task.

The solution is to use the [Project.copy(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copy(org.gradle.api.Action)) method. It works the same way as the Copy task by configuring it with a copy spec. Here’s a trivial example:

[Example: Copying files using the copy() method without up-to-date check](#xvir7l)

**build.gradle**

task copyMethod {  
 doLast {  
 copy {  
 from 'src/main/webapp'  
 into "${buildDir}/explodedWar"  
 include '\*\*/\*.html'  
 include '\*\*/\*.jsp'  
 }  
 }  
}

The above example demonstrates the basic syntax and also highlights two major limitations of using the copy() method:

1. The copy() method is not [incremental](http://docs.google.com/more_about_tasks.html#sec:up_to_date_checks). The example’s copyMethod task will *always* execute because it has no information about what files make up the task’s inputs. You have to manually define the task inputs and outputs.
2. Using a task as a copy source, i.e. as an argument to from(), won’t set up an automatic task dependency between your task and that copy source. As such, if you are using the copy() method as part of a task action, you must explicitly declare all inputs and outputs in order to get the correct behavior.

The following example shows you how to workaround these limitations by using the [dynamic API for task inputs and outputs](http://docs.google.com/more_about_tasks.html#sec:task_input_output_runtime_api):

[Example: Copying files using the copy() method with up-to-date check](#3hv69ve)

**build.gradle**

task copyMethodWithExplicitDependencies{  
 // up-to-date check for inputs, plus add copyTask as dependency  
 inputs.files copyTask  
 outputs.dir 'some-dir' // up-to-date check for outputs  
 doLast{  
 copy {  
 // Copy the output of copyTask  
 from copyTask  
 into 'some-dir'  
 }  
 }  
}

These limitations make it preferable to use the Copy task wherever possible, because of its builtin support for incremental building and task dependency inference. That is why the copy() method is intended for use by [custom tasks](http://docs.google.com/custom_tasks.html#custom_tasks) that need to copy files as part of their function. Custom tasks that use the copy() method should declare the necessary inputs and outputs relevant to the copy action.

[Mirroring directories and file collections with the Sync task](#1x0gk37)

The [Sync](http://docs.google.com/dsl/org.gradle.api.tasks.Sync.html) task, which extends the Copy task, copies the source files into the destination directory and then removes any files from the destination directory which it did not copy. In other words, it synchronizes the contents of a directory with its source. This can be useful for doing things such as installing your application, creating an exploded copy of your archives, or maintaining a copy of the project’s dependencies.

Here is an example which maintains a copy of the project’s runtime dependencies in the build/libs directory.

[Example: Using the Sync task to copy dependencies](#4h042r0)

**build.gradle**

task libs(type: Sync) {  
 from configurations.runtime  
 into "${buildDir}/libs"  
}

You can also perform the same function in your own tasks with the [Project.sync(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:sync(org.gradle.api.Action)) method.

[Archive creation in depth](#1y810tw)

Archives are essentially self-contained file systems and Gradle treats them as such. This is why working with archives is very similar to working with files and directories, including such things as file permissions.

Out of the box, Gradle supports creation of both ZIP and TAR archives, and by extension Java’s JAR, WAR and EAR formats — Java’s archive formats are all ZIPs. Each of these formats has a corresponding task type to create them: [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html), [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html), [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html), [War](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.War.html), and [Ear](http://docs.google.com/dsl/org.gradle.plugins.ear.Ear.html). These all work the same way and are based on copy specifications, just like the Copy task.

Creating an archive file is essentially a file copy in which the destination is implicit, i.e. the archive file itself. Here’s a basic example that specifies the path and name of the target archive file:

[Example: Archiving a directory as a ZIP](#2w5ecyt)

**build.gradle**

task packageDistribution(type: Zip) {  
 archiveName = "my-distribution.zip"  
 destinationDir = file("${buildDir}/dist")  
  
 from "${buildDir}/toArchive"  
}

In the next section you’ll learn about convention-based archive names, which can save you from always configuring the destination directory and archive name.

The full power of copy specifications are available to you when creating archives, which means you can do content filtering, file renaming or anything else that is covered in the previous section. A particularly common requirement is copying files into subdirectories of the archive that don’t exist in the source folders, something that can be achieved with into() [child specifications](#2p2csry).

Gradle does of course allow you create as many archive tasks as you want, but it’s worth bearing in mind that many convention-based plugins provide their own. For example, the Java plugin adds a jar task for packaging a project’s compiled classes and resources in a JAR. Many of these plugins provide sensible conventions for the names of archives as well as the copy specifications used. We recommend you use these tasks wherever you can, rather than overriding them with your own.

[Archive naming](#3o7alnk)

Gradle has several conventions around the naming of archives and where they are created based on the plugins your project uses. The main convention is provided by the [Base Plugin](http://docs.google.com/base_plugin.html#base_plugin), which defaults to creating archives in the $buildDir/distributions directory and typically uses archive names of the form *[projectName]-[version].[type]*.

The following example comes from a project named 'zipProject', hence the myZip task creates an archive named 'zipProject-1.0.zip':

[Example: Creation of ZIP archive](#1baon6m)

**build.gradle**

plugins {  
 id 'base'  
}  
  
version = 1.0  
  
task myZip(type: Zip) {  
 from 'somedir'  
  
 doLast {  
 println archiveName  
 println relativePath(destinationDir)  
 println relativePath(archivePath)  
 }  
}

**Output of** gradle -q myZip

> gradle -q myZip  
zipProject-1.0.zip  
build/distributions  
build/distributions/zipProject-1.0.zip

Note that the name of the archive does *not* derive from the name of the task that creates it.

If you want to change the name and location of a generated archive file, you can provide values for the archiveName and destinationDir properties of the corresponding task. These override any conventions that would otherwise apply.

Alternatively, you can make use of the default archive name pattern provided by [AbstractArchiveTask.getArchiveName()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html#org.gradle.api.tasks.bundling.AbstractArchiveTask:archiveName): *[baseName]-[appendix]-[version]-[classifier].[extension]*. You can set each of these properties on the task separately if you wish. Note that the Base Plugin uses the convention of project name for *baseName*, project version for *version* and the archive type for *extension*. It does not provide values for the other properties.

This example — from the same project as the one above — configures just the baseName property, overriding the default value of the project name:

[Example: Configuration of archive task - custom archive name](#3vac5uf)

**build.gradle**

task myCustomZip(type: Zip) {  
 baseName = 'customName'  
 from 'somedir'  
  
 doLast {  
 println archiveName  
 }  
}

**Output of** gradle -q myCustomZip

> gradle -q myCustomZip  
customName-1.0.zip

You can also override the default baseName value for *all* the archive tasks in your build by using the *project* property archivesBaseName, as demonstrated by the following example:

[Example: Configuration of archive task - appendix & classifier](#2afmg28)

**build.gradle**

plugins {  
 id 'base'  
}  
  
version = 1.0  
archivesBaseName = "gradle"  
  
task myZip(type: Zip) {  
 from 'somedir'  
}  
  
task myOtherZip(type: Zip) {  
 appendix = 'wrapper'  
 classifier = 'src'  
 from 'somedir'  
}  
  
task echoNames {  
 doLast {  
 println "Project name: ${project.name}"  
 println myZip.archiveName  
 println myOtherZip.archiveName  
 }  
}

**Output of** gradle -q echoNames

> gradle -q echoNames  
Project name: zipProject  
gradle-1.0.zip  
gradle-wrapper-1.0-src.zip

You can find all the possible archive task properties in the API documentation for [AbstractArchiveTask](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html), but we have also summarized the main ones here:

archiveName — String, default: *baseName*-*appendix*-*version*-*classifier*.*extension*

The complete file name of the generated archive. If any of the properties in the default value are empty, their '-' separator is dropped.

archivePath — File, *read-only*, default: *destinationDir*/*archiveName*

The absolute file path of the generated archive.

destinationDir — File, default: depends on archive type

The target directory in which to put the generated archive. By default, JARs and WARs go into $buildDir/libs. ZIPs and TARs go into $buildDir/distributions.

baseName — String, default: *project.name*

The base name portion of the archive file name, typically a project name or some other descriptive name for what it contains.

appendix — String, default: null

The appendix portion of the archive file name that comes immediately after the base name. It is typically used to distinguish between different forms of content, such as code and docs, or a minimal distribution versus a full or complete one.

version — String, default: *project.version*

The version portion of the archive file name, typically in the form of a normal project or product version.

classifier — String, default: null

The classifier portion of the archive file name. Often used to distinguish between archives that target different platforms.

extension — String, default: depends on archive type and compression type

The filename extension for the archive. By default, this is set based on the archive task type and the compression type (if you’re creating a TAR). Will be one of: zip, jar, war, tar, tgz or tbz2. You can of course set this to a custom extension if you wish.

[Sharing content between multiple archives](#pkwqa1)

[As described earlier](#kgcv8k), you can use the [Project.copySpec(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copySpec(org.gradle.api.Action)) method to share content between archives.

[Reproducible archives](#39kk8xu)

Sometimes it’s desirable to recreate archives exactly the same, byte for byte, on different machines. You want to be sure that building an artifact from source code produces the same result no matter when and where it is built. This is necessary for projects like [reproducible-builds.org](https://reproducible-builds.org/).

Reproducing the same byte-for-byte archive poses some challenges since the order of the files in an archive is influenced by the underlying file system. Each time a ZIP, TAR, JAR, WAR or EAR is built from source, the order of the files inside the archive may change. Files that only have a different timestamp also causes differences in archives from build to build. All [AbstractArchiveTask](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html) (e.g. Jar, Zip) tasks shipped with Gradle include [incubating](http://docs.google.com/feature_lifecycle.html#feature_lifecycle) support producing reproducible archives.

For example, to make a Zip task reproducible you need to set [Zip.isReproducibleFileOrder()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html#org.gradle.api.tasks.bundling.Zip:reproducibleFileOrder) to true and [Zip.isPreserveFileTimestamps()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html#org.gradle.api.tasks.bundling.Zip:preserveFileTimestamps) to false. In order to make all archive tasks in your build reproducible, consider adding the following configuration to your build file:

[Example: Activating reproducible archives](#1opuj5n)

**build.gradle**

tasks.withType(AbstractArchiveTask) {  
 preserveFileTimestamps = false  
 reproducibleFileOrder = true  
}

Often you will want to publish an archive, so that it is usable from another project. This process is described in [Legacy Publishing](http://docs.google.com/artifact_management.html#artifact_management).

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/)