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/)

# Authoring Tasks

Contents

[Task outcomes](#4d34og8)

[Defining tasks](#2s8eyo1)

[Locating tasks](#17dp8vu)

[Configuring tasks](#3rdcrjn)

[Passing arguments to a task constructor](#26in1rg)

[Adding dependencies to a task](#lnxbz9)

[Ordering tasks](#35nkun2)

[Adding a description to a task](#1ksv4uv)

[Replacing tasks](#44sinio)

[Skipping tasks](#2jxsxqh)

[Up-to-date checks (AKA Incremental Build)](#z337ya)

[Task rules](#3j2qqm3)

[Finalizer tasks](#1y810tw)

[Lifecycle tasks](#4i7ojhp)

[Summary](#2xcytpi)

In the [introductory tutorial](http://docs.google.com/tutorial_using_tasks.html#tutorial_using_tasks) you learned how to create simple tasks. You also learned how to add additional behavior to these tasks later on, and you learned how to create dependencies between tasks. This was all about simple tasks, but Gradle takes the concept of tasks further. Gradle supports *enhanced tasks*, which are tasks that have their own properties and methods. This is really different from what you are used to with Ant targets. Such enhanced tasks are either provided by you or built into Gradle.

[Task outcomes](#4d34og8)

When Gradle executes a task, it can label the task with different outcomes in the console UI and via the [Tooling API](http://docs.google.com/embedding.html#embedding). These labels are based on if a task has actions to execute, if it should execute those actions, if it did execute those actions and if those actions made any changes.

(no label) or EXECUTED

Task executed its actions.

* Task has actions and Gradle has determined they should be executed as part of a build.
* Task has no actions and some dependencies, and any of the dependencies are executed. See also [Lifecycle Tasks](#4i7ojhp).

UP-TO-DATE

Task’s outputs did not change.

* Task has outputs and inputs and they have not changed. See [Incremental Builds](#z337ya).
* Task has actions, but the task tells Gradle it did not change its outputs.
* Task has no actions and some dependencies, but all of the dependencies are up-to-date, skipped or from cache. See also [Lifecycle Tasks](#4i7ojhp).
* Task has no actions and no dependencies.

FROM-CACHE

Task’s outputs could be found from a previous execution.

* Task has outputs restored from the build cache. See [Build Cache](http://docs.google.com/build_cache.html#build_cache).

SKIPPED

Task did not execute its actions.

* Task has been explicitly excluded from the command-line. See [Excluding tasks from execution](http://docs.google.com/command_line_interface.html#sec:excluding_tasks_from_the_command_line).
* Task has an onlyIf predicate return false. See [Using a predicate](#1ci93xb).

NO-SOURCE

Task did not need to execute its actions.

* Task has inputs and outputs, but [no sources](#3whwml4). For example, source files are .java files for [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html).

[Defining tasks](#2s8eyo1)

We have already seen how to define tasks using a keyword style in [this chapter](http://docs.google.com/tutorial_using_tasks.html#tutorial_using_tasks). There are a few variations on this style, which you may need to use in certain situations. For example, the keyword style does not work in expressions.

[Example: Defining tasks](#2bn6wsx)

**build.gradle**

task(hello) {  
 doLast {  
 println "hello"  
 }  
}  
  
task(copy, type: Copy) {  
 from(file('srcDir'))  
 into(buildDir)  
}

You can also use strings for the task names:

[Example: Defining tasks - using strings for task names](#qsh70q)

**build.gradle**

task('hello') {  
 doLast {  
 println "hello"  
 }  
}  
  
task('copy', type: Copy) {  
 from(file('srcDir'))  
 into(buildDir)  
}

There is an alternative syntax for defining tasks, which you may prefer to use:

[Example: Defining tasks with alternative syntax](#3as4poj)

**build.gradle**

tasks.create('hello') {  
 doLast {  
 println "hello"  
 }  
}  
  
tasks.create('copy', Copy) {  
 from(file('srcDir'))  
 into(buildDir)  
}

Here we add tasks to the tasks collection. Have a look at [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) for more variations of the create() method.

[Locating tasks](#17dp8vu)

You often need to locate the tasks that you have defined in the build file, for example, to configure them or use them for dependencies. There are a number of ways of doing this. Firstly, each task is available as a property of the project, using the task name as the property name:

[Example: Accessing tasks as properties](#1pxezwc)

**build.gradle**

task hello  
  
println hello.name  
println project.hello.name

Tasks are also available through the tasks collection.

[Example: Accessing tasks via tasks collection](#49x2ik5)

**build.gradle**

task hello  
  
println tasks.hello.name  
println tasks['hello'].name

You can access tasks from any project using the task’s path using the tasks.getByPath() method. You can call the getByPath() method with a task name, or a relative path, or an absolute path.

[Example: Accessing tasks by path](#2p2csry)

**build.gradle**

project(':projectA') {  
 task hello  
}  
  
task hello  
  
println tasks.getByPath('hello').path  
println tasks.getByPath(':hello').path  
println tasks.getByPath('projectA:hello').path  
println tasks.getByPath(':projectA:hello').path

**Output of** gradle -q hello

> gradle -q hello  
:hello  
:hello  
:projectA:hello  
:projectA:hello

Have a look at [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) for more options for locating tasks.

[Configuring tasks](#3rdcrjn)

As an example, let’s look at the Copy task provided by Gradle. To create a Copy task for your build, you can declare in your build script:

[Example: Creating a copy task](#147n2zr)

**build.gradle**

task myCopy(type: Copy)

This creates a copy task with no default behavior. The task can be configured using its API (see [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)). The following examples show several different ways to achieve the same configuration.

Just to be clear, realize that the name of this task is “myCopy”, but it is of *type* “Copy”. You can have multiple tasks of the same *type*, but with different names. You’ll find this gives you a lot of power to implement cross-cutting concerns across all tasks of a particular type.

[Example: Configuring a task - various ways](#3o7alnk)

**build.gradle**

Copy myCopy = task(myCopy, type: Copy)  
myCopy.from 'resources'  
myCopy.into 'target'  
myCopy.include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')

This is similar to the way we would configure objects in Java. You have to repeat the context (myCopy) in the configuration statement every time. This is a redundancy and not very nice to read.

There is another way of configuring a task. It also preserves the context and it is arguably the most readable. It is usually our favorite.

[Example: Configuring a task - with closure](#23ckvvd)

**build.gradle**

task myCopy(type: Copy)  
  
myCopy {  
 from 'resources'  
 into 'target'  
 include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')  
}

This works for *any* task. Line 3 of the example is just a shortcut for the tasks.getByName() method. It is important to note that if you pass a closure to the getByName() method, this closure is applied to *configure* the task, not when the task executes.

You can also use a configuration closure when you define a task.

[Example: Defining a task with closure](#ihv636)

**build.gradle**

task copy(type: Copy) {  
 from 'resources'  
 into 'target'  
 include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')  
}

| **💡** | Don’t forget about the build phases  A task has both configuration and actions. When using the doLast, you are simply using a shortcut to define an action. Code defined in the configuration section of your task will get executed during the configuration phase of the build regardless of what task was targeted. See [Build Lifecycle](http://docs.google.com/build_lifecycle.html#build_lifecycle) for more details about the build lifecycle. |
| --- | --- |

[Passing arguments to a task constructor](#26in1rg)

As opposed to configuring the mutable properties of a Task after creation, you can pass argument values to the Task class’s constructor. In order to pass values to the Task constructor, you must annotate the relevant constructor with @javax.inject.Inject.

[Example: Task class with @Inject constructor](#32hioqz)

**build.gradle**

class CustomTask extends DefaultTask {  
 final String message  
 final int number  
  
 @Inject  
 CustomTask(String message, int number) {  
 this.message = message  
 this.number = number  
 }

You can then create a task, passing the constructor arguments at the end of the parameter list.

[Example: Creating a task with constructor arguments using TaskContainer](#1hmsyys)

**build.gradle**

tasks.create('myTask', CustomTask, 'hello', 42)

In a Groovy build script, you can create the task using constructorArgs.

[Example: Creating a task with constructor arguments using Map](#41mghml)

**build.gradle**

task myTask(type: CustomTask, constructorArgs: ['hello', 42])

In a Kotlin build script, you can pass constructor arguments using the reified extension function on the tasks TaskContainer.

[Example: Creating a task with constructor arguments using Kotlin DSL](#2grqrue)

**build.gradle.kts**

open class CustomTask @Inject constructor(private val message: String, private val number: Int) : DefaultTask() {  
 @TaskAction fun run() = println("$message $number")  
}  
  
tasks.create<CustomTask>("myTask", "hello", 42)

In all circumstances, the values passed as constructor arguments must be non-null. If you attempt to pass a null value, Gradle will throw a NullPointerException indicating which runtime value is null.

[Adding dependencies to a task](#lnxbz9)

There are several ways you can define the dependencies of a task. In [Task dependencies](http://docs.google.com/tutorial_using_tasks.html#sec:task_dependencies) you were introduced to defining dependencies using task names. Task names can refer to tasks in the same project as the task, or to tasks in other projects. To refer to a task in another project, you prefix the name of the task with the path of the project it belongs to. The following is an example which adds a dependency from projectA:taskX to projectB:taskY:

[Example: Adding dependency on task from another project](#vx1227)

**build.gradle**

project('projectA') {  
 task taskX(dependsOn: ':projectB:taskY') {  
 doLast {  
 println 'taskX'  
 }  
 }  
}  
  
project('projectB') {  
 task taskY {  
 doLast {  
 println 'taskY'  
 }  
 }  
}

**Output of** gradle -q taskX

> gradle -q taskX  
taskY  
taskX

Instead of using a task name, you can define a dependency using a Task object, as shown in this example:

[Example: Adding dependency using task object](#3fwokq0)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
  
taskX.dependsOn taskY

**Output of** gradle -q taskX

> gradle -q taskX  
taskY  
taskX

For more advanced uses, you can define a task dependency using a closure. When evaluated, the closure is passed the task whose dependencies are being calculated. The closure should return a single Task or collection of Task objects, which are then treated as dependencies of the task. The following example adds a dependency from taskX to all the tasks in the project whose name starts with lib:

[Example: Adding dependency using closure](#1v1yuxt)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
  
taskX.dependsOn {  
 tasks.findAll { task -> task.name.startsWith('lib') }  
}  
  
task lib1 {  
 doLast {  
 println 'lib1'  
 }  
}  
  
task lib2 {  
 doLast {  
 println 'lib2'  
 }  
}  
  
task notALib {  
 doLast {  
 println 'notALib'  
 }  
}

**Output of** gradle -q taskX

> gradle -q taskX  
lib1  
lib2  
taskX

For more information about task dependencies, see the [Task](http://docs.google.com/dsl/org.gradle.api.Task.html) API.

[Ordering tasks](#35nkun2)

| **✨** | Task ordering is an [incubating](http://docs.google.com/feature_lifecycle.html#feature_lifecycle) feature. Please be aware that this feature may change in later Gradle versions. |
| --- | --- |

In some cases it is useful to control the *order* in which 2 tasks will execute, without introducing an explicit dependency between those tasks. The primary difference between a task *ordering* and a task *dependency* is that an ordering rule does not influence which tasks will be executed, only the order in which they will be executed.

Task ordering can be useful in a number of scenarios:

* Enforce sequential ordering of tasks: e.g. 'build' never runs before 'clean'.
* Run build validations early in the build: e.g. validate I have the correct credentials before starting the work for a release build.
* Get feedback faster by running quick verification tasks before long verification tasks: e.g. unit tests should run before integration tests.
* A task that aggregates the results of all tasks of a particular type: e.g. test report task combines the outputs of all executed test tasks.

There are two ordering rules available: “*must run after*” and “*should run after*”.

When you use the “must run after” ordering rule you specify that taskB must always run after taskA, whenever both taskA and taskB will be run. This is expressed as taskB.mustRunAfter(taskA). The “should run after” ordering rule is similar but less strict as it will be ignored in two situations. Firstly if using that rule introduces an ordering cycle. Secondly when using parallel execution and all dependencies of a task have been satisfied apart from the “should run after” task, then this task will be run regardless of whether its “should run after” dependencies have been run or not. You should use “should run after” where the ordering is helpful but not strictly required.

With these rules present it is still possible to execute taskA without taskB and vice-versa.

[Example: Adding a 'must run after' task ordering](#4f1mdlm)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
taskY.mustRunAfter taskX

**Output of** gradle -q taskY taskX

> gradle -q taskY taskX  
taskX  
taskY

[Example: Adding a 'should run after' task ordering](#2u6wntf)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
taskY.shouldRunAfter taskX

**Output of** gradle -q taskY taskX

> gradle -q taskY taskX  
taskX  
taskY

In the examples above, it is still possible to execute taskY without causing taskX to run:

[Example: Task ordering does not imply task execution](#19c6y18)

**Output of** gradle -q taskY

> gradle -q taskY  
taskY

To specify a “must run after” or “should run after” ordering between 2 tasks, you use the [Task.mustRunAfter(java.lang.Object...)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:mustRunAfter(java.lang.Object%5B%5D)) and [Task.shouldRunAfter(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/Task.html#shouldRunAfter-java.lang.Object...-) methods. These methods accept a task instance, a task name or any other input accepted by [Task.dependsOn(java.lang.Object...)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:dependsOn(java.lang.Object%5B%5D)).

Note that “B.mustRunAfter(A)” or “B.shouldRunAfter(A)” does not imply any execution dependency between the tasks:

* It is possible to execute tasks A and B independently. The ordering rule only has an effect when both tasks are scheduled for execution.
* When run with --continue, it is possible for B to execute in the event that A fails.

As mentioned before, the “should run after” ordering rule will be ignored if it introduces an ordering cycle:

[Example: A 'should run after' task ordering is ignored if it introduces an ordering cycle](#3tbugp1)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
task taskZ {  
 doLast {  
 println 'taskZ'  
 }  
}  
taskX.dependsOn taskY  
taskY.dependsOn taskZ  
taskZ.shouldRunAfter taskX

**Output of** gradle -q taskX

> gradle -q taskX  
taskZ  
taskY  
taskX

[Adding a description to a task](#1ksv4uv)

You can add a description to your task. This description is displayed when executing gradle tasks.

[Example: Adding a description to a task](#28h4qwu)

**build.gradle**

task copy(type: Copy) {  
 description 'Copies the resource directory to the target directory.'  
 from 'resources'  
 into 'target'  
 include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')  
}

[Replacing tasks](#44sinio)

Sometimes you want to replace a task. For example, if you want to exchange a task added by the Java plugin with a custom task of a different type. You can achieve this with:

[Example: Overwriting a task](#nmf14n)

**build.gradle**

task copy(type: Copy)  
  
task copy(overwrite: true) {  
 doLast {  
 println('I am the new one.')  
 }  
}

**Output of** gradle -q copy

> gradle -q copy  
I am the new one.

This will replace a task of type Copy with the task you’ve defined, because it uses the same name. When you define the new task, you have to set the overwrite property to true. Otherwise Gradle throws an exception, saying that a task with that name already exists.

[Skipping tasks](#2jxsxqh)

Gradle offers multiple ways to skip the execution of a task.

[Using a predicate](#1ci93xb)

You can use the onlyIf() method to attach a predicate to a task. The task’s actions are only executed if the predicate evaluates to true. You implement the predicate as a closure. The closure is passed the task as a parameter, and should return true if the task should execute and false if the task should be skipped. The predicate is evaluated just before the task is due to be executed.

[Example: Skipping a task using a predicate](#37m2jsg)

**build.gradle**

task hello {  
 doLast {  
 println 'hello world'  
 }  
}  
  
hello.onlyIf { !project.hasProperty('skipHello') }

**Output of** gradle hello -PskipHello

> gradle hello -PskipHello  
> Task :hello SKIPPED  
  
BUILD SUCCESSFUL in 0s

[Using StopExecutionException](#1mrcu09)

If the logic for skipping a task can’t be expressed with a predicate, you can use the [StopExecutionException](http://docs.google.com/javadoc/org/gradle/api/tasks/StopExecutionException.html). If this exception is thrown by an action, the further execution of this action as well as the execution of any following action of this task is skipped. The build continues with executing the next task.

[Example: Skipping tasks with StopExecutionException](#46r0co2)

**build.gradle**

task compile {  
 doLast {  
 println 'We are doing the compile.'  
 }  
}  
  
compile.doFirst {  
 // Here you would put arbitrary conditions in real life.  
 // But this is used in an integration test so we want defined behavior.  
 if (true) { throw new StopExecutionException() }  
}  
task myTask(dependsOn: 'compile') {  
 doLast {  
 println 'I am not affected'  
 }  
}

**Output of** gradle -q myTask

> gradle -q myTask  
I am not affected

This feature is helpful if you work with tasks provided by Gradle. It allows you to add *conditional* execution of the built-in actions of such a task.[[1](#2lwamvv)]

[Enabling and disabling tasks](#111kx3o)

Every task has an enabled flag which defaults to true. Setting it to false prevents the execution of any of the task’s actions. A disabled task will be labelled SKIPPED.

[Example: Enabling and disabling tasks](#3l18frh)

**build.gradle**

task disableMe {  
 doLast {  
 println 'This should not be printed if the task is disabled.'  
 }  
}  
disableMe.enabled = false

**Output of** gradle disableMe

> gradle disableMe  
> Task :disableMe SKIPPED  
  
BUILD SUCCESSFUL in 0s

[Up-to-date checks (AKA Incremental Build)](#z337ya)

An important part of any build tool is the ability to avoid doing work that has already been done. Consider the process of compilation. Once your source files have been compiled, there should be no need to recompile them unless something has changed that affects the output, such as the modification of a source file or the removal of an output file. And compilation can take a significant amount of time, so skipping the step when it’s not needed saves a lot of time.

Gradle supports this behavior out of the box through a feature it calls incremental build. You have almost certainly already seen it in action: it’s active nearly every time the UP-TO-DATE text appears next to the name of a task when you run a build. Task outcomes are described in [Task outcomes](#4d34og8).

How does incremental build work? And what does it take to make use of it in your own tasks? Let’s take a look.

[Task inputs and outputs](#206ipza)

In the most common case, a task takes some inputs and generates some outputs. If we use the compilation example from earlier, we can see that the source files are the inputs and, in the case of Java, the generated class files are the outputs. Other inputs might include things like whether debug information should be included.



*Figure 1. Example task inputs and outputs*

An important characteristic of an input is that it affects one or more outputs, as you can see from the previous figure. Different bytecode is generated depending on the content of the source files and the minimum version of the Java runtime you want to run the code on. That makes them task inputs. But whether compilation has 500MB or 600MB of maximum memory available, determined by the memoryMaximumSize property, has no impact on what bytecode gets generated. In Gradle terminology, memoryMaximumSize is just an internal task property.

As part of incremental build, Gradle tests whether any of the task inputs or outputs have changed since the last build. If they haven’t, Gradle can consider the task up to date and therefore skip executing its actions. Also note that incremental build won’t work unless a task has at least one task output, although tasks usually have at least one input as well.

What this means for build authors is simple: you need to tell Gradle which task properties are inputs and which are outputs. If a task property affects the output, be sure to register it as an input, otherwise the task will be considered up to date when it’s not. Conversely, don’t register properties as inputs if they don’t affect the output, otherwise the task will potentially execute when it doesn’t need to. Also be careful of non-deterministic tasks that may generate different output for exactly the same inputs: these should not be configured for incremental build as the up-to-date checks won’t work.

Let’s now look at how you can register task properties as inputs and outputs.

[Custom task types](#4k668n3)

If you’re implementing a custom task as a class, then it takes just two steps to make it work with incremental build:

1. Create typed properties (via getter methods) for each of your task inputs and outputs
2. Add the appropriate annotation to each of those properties

| **✨** | Annotations must be placed on getters or on Groovy properties. Annotations placed on setters, or on a Java field without a corresponding annotated getter are ignored. |
| --- | --- |

Gradle supports three main categories of inputs and outputs:

* Simple values  
  Things like strings and numbers. More generally, a simple value can have any type that implements Serializable.
* Filesystem types  
  These consist of the standard File class but also derivatives of Gradle’s [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) type and anything else that can be passed to either 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 single file/directory properties - or 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.
* Nested values  
  Custom types that don’t conform to the other two categories but have their own properties that are inputs or outputs. In effect, the task inputs or outputs are nested inside these custom types.

As an example, imagine you have a task that processes templates of varying types, such as FreeMarker, Velocity, Moustache, etc. It takes template source files and combines them with some model data to generate populated versions of the template files.

This task will have three inputs and one output:

* Template source files
* Model data
* Template engine
* Where the output files are written

When you’re writing a custom task class, it’s easy to register properties as inputs or outputs via annotations. To demonstrate, here is a skeleton task implementation with some suitable inputs and outputs, along with their annotations:

[Example: Custom task class](#2zbgiuw)

**buildSrc/src/main/java/org/example/ProcessTemplates.java**

package org.example;  
  
import java.io.File;  
import java.util.HashMap;  
import org.gradle.api.\*;  
import org.gradle.api.file.\*;  
import org.gradle.api.tasks.\*;  
  
public class ProcessTemplates extends DefaultTask {  
 private TemplateEngineType templateEngine;  
 private FileCollection sourceFiles;  
 private TemplateData templateData;  
 private File outputDir;  
  
 @Input  
 public TemplateEngineType getTemplateEngine() {  
 return this.templateEngine;  
 }  
  
 @InputFiles  
 public FileCollection getSourceFiles() {  
 return this.sourceFiles;  
 }  
  
 @Nested  
 public TemplateData getTemplateData() {  
 return this.templateData;  
 }  
  
 @OutputDirectory  
 public File getOutputDir() { return this.outputDir; }  
  
 // + setter methods for the above - assume we’ve defined them  
  
 @TaskAction  
 public void processTemplates() {  
 // ...  
 }  
}

**buildSrc/src/main/java/org/example/TemplateData.java**

package org.example;  
  
import java.util.HashMap;  
import java.util.Map;  
import org.gradle.api.tasks.Input;  
  
public class TemplateData {  
 private String name;  
 private Map<String, String> variables;  
  
 public TemplateData(String name, Map<String, String> variables) {  
 this.name = name;  
 this.variables = new HashMap<>(variables);  
 }  
  
 @Input  
 public String getName() { return this.name; }  
  
 @Input  
 public Map<String, String> getVariables() {  
 return this.variables;  
 }  
}

**Output of gradle processTemplates**

> gradle processTemplates  
> Task :processTemplates  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

**Output of gradle processTemplates (run again)**

> gradle processTemplates  
> Task :processTemplates UP-TO-DATE  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

There’s plenty to talk about in this example, so let’s work through each of the input and output properties in turn:

* templateEngine  
  Represents which engine to use when processing the source templates, e.g. FreeMarker, Velocity, etc. You could implement this as a string, but in this case we have gone for a custom enum as it provides greater type information and safety. Since enums implement Serializable automatically, we can treat this as a simple value and use the @Input annotation, just as we would with a String property.
* sourceFiles  
  The source templates that the task will be processing. Single files and collections of files need their own special annotations. In this case, we’re dealing with a collection of input files and so we use the @InputFiles annotation. You’ll see more file-oriented annotations in a table later.
* templateData  
  For this example, we’re using a custom class to represent the model data. However, it does not implement Serializable, so we can’t use the @Input annotation. That’s not a problem as the properties within TemplateData - a string and a hash map with serializable type parameters - are serializable and can be annotated with @Input. We use @Nested on templateData to let Gradle know that this is a value with nested input properties.
* outputDir  
  The directory where the generated files go. As with input files, there are several annotations for output files and directories. A property representing a single directory requires @OutputDirectory. You’ll learn about the others soon.

These annotated properties mean that Gradle will skip the task if none of the source files, template engine, model data or generated files have changed since the previous time Gradle executed the task. This will often save a significant amount of time. You can learn how Gradle detects [changes later](#1egqt2p).

This example is particularly interesting because it works with collections of source files. What happens if only one source file changes? Does the task process all the source files again or just the modified one? That depends on the task implementation. If the latter, then the task itself is incremental, but that’s a different feature to the one we’re discussing here. Gradle does help task implementers with this via its [incremental task inputs](http://docs.google.com/custom_tasks.html#incremental_tasks) feature.

Now that you have seen some of the input and output annotations in practice, let’s take a look at all the annotations available to you and when you should use them. The table below lists the available annotations and the corresponding property type you can use with each one.

Table 1. Incremental build property type annotations

| **Annotation** | **Expected property type** | **Description** |
| --- | --- | --- |
| @[Input](http://docs.google.com/javadoc/org/gradle/api/tasks/Input.html) | Any Serializable type | A simple input value |
| @[InputFile](http://docs.google.com/javadoc/org/gradle/api/tasks/InputFile.html) | File\* | A single input file (not directory) |
| @[InputDirectory](http://docs.google.com/javadoc/org/gradle/api/tasks/InputDirectory.html) | File\* | A single input directory (not file) |
| @[InputFiles](http://docs.google.com/javadoc/org/gradle/api/tasks/InputFiles.html) | Iterable<File>\* | An iterable of input files and directories |
| @[Classpath](http://docs.google.com/javadoc/org/gradle/api/tasks/Classpath.html) | Iterable<File>\* | An iterable of input files and directories that represent a Java classpath. This allows the task to ignore irrelevant changes to the property, such as different names for the same files. It is similar to annotating the property @PathSensitive(RELATIVE) but it will ignore the names of JAR files directly added to the classpath, and it will consider changes in the order of the files as a change in the classpath. Gradle will inspect the contents of jar files on the classpath and ignore changes that do not affect the semantics of the classpath (such as file dates and entry order). See also [Using the classpath annotations](#3ygebqi).  Note: The @Classpath annotation was introduced in Gradle 3.2. To stay compatible with earlier Gradle versions, classpath properties should also be annotated with @InputFiles. |
| @[CompileClasspath](http://docs.google.com/javadoc/org/gradle/api/tasks/CompileClasspath.html) | Iterable<File>\* | An iterable of input files and directories that represent a Java compile classpath. This allows the task to ignore irrelevant changes that do not affect the API of the classes in classpath. See also [Using the classpath annotations](#3ygebqi).  The following kinds of changes to the classpath will be ignored:   * Changes to the path of jar or top level directories. * Changes to timestamps and the order of entries in Jars. * Changes to resources and Jar manifests, including adding or removing resources. * Changes to private class elements, such as private fields, methods and inner classes. * Changes to code, such as method bodies, static initializers and field initializers (except for constants). * Changes to debug information, for example when a change to a comment affects the line numbers in class debug information. * Changes to directories, including directory entries in Jars.  | **✨** | The @CompileClasspath annotation was introduced in Gradle 3.4. To stay compatible with Gradle 3.3 and 3.2, compile classpath properties should also be annotated with @Classpath. For compatibility with Gradle versions before 3.2 the property should also be annotated with @InputFiles. | | --- | --- | |
| @[OutputFile](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputFile.html) | File\* | A single output file (not directory) |
| @[OutputDirectory](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputDirectory.html) | File\* | A single output directory (not file) |
| @[OutputFiles](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputFiles.html) | Map<String, File>\*\* or Iterable<File>\* | An iterable of output files (no directories). The task outputs can only be [cached](http://docs.google.com/build_cache.html#sec:task_output_caching) if a Map is provided. |
| @[OutputDirectories](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputDirectories.html) | Map<String, File>\*\* or Iterable<File>\* | An iterable of output directories (no files). The task outputs can only be [cached](http://docs.google.com/build_cache.html#sec:task_output_caching) if a Map is provided. |
| @[Destroys](http://docs.google.com/javadoc/org/gradle/api/tasks/Destroys.html) | File or Iterable<File>\* | Specifies one or more files that are removed by this task. Note that a task can define either inputs/outputs or destroyables, but not both. |
| @[LocalState](http://docs.google.com/javadoc/org/gradle/api/tasks/LocalState.html) | File or Iterable<File>\* | Specifies one or more files that represent the [local state of the task](http://docs.google.com/custom_tasks.html#sec:storing_incremental_task_state). These files are removed when the task is loaded from cache. |
| @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) | Any custom type | A custom type that may not implement Serializable but does have at least one field or property marked with one of the annotations in this table. It could even be another @Nested. |
| @[Console](http://docs.google.com/javadoc/org/gradle/api/tasks/Console.html) | Any type | Indicates that the property is neither an input nor an output. It simply affects the console output of the task in some way, such as increasing or decreasing the verbosity of the task. |
| @[Internal](http://docs.google.com/javadoc/org/gradle/api/tasks/Internal.html) | Any type | Indicates that the property is used internally but is neither an input nor an output. |
| @[SkipWhenEmpty](http://docs.google.com/javadoc/org/gradle/api/tasks/SkipWhenEmpty.html) | File\* | Used with @InputFiles or @InputDirectory to tell Gradle to skip the task if the corresponding files or directory are empty, along with all other input files declared with this annotation. Tasks that have been skipped due to all of their input files that were declared with this annotation being empty will result in a distinct “no source” outcome. For example, NO-SOURCE will be emitted in the console output. |
| @[Optional](http://docs.google.com/javadoc/org/gradle/api/tasks/Optional.html) | Any type | Used with any of the property type annotations listed in the [Optional](http://docs.google.com/javadoc/org/gradle/api/tasks/Optional.html) API documentation. This annotation disables validation checks on the corresponding property. See [the section on validation](#2dlolyb) for more details. |
| @[PathSensitive](http://docs.google.com/javadoc/org/gradle/api/tasks/PathSensitive.html) | File\* | Used with any input file property to tell Gradle to only consider the given part of the file paths as important. For example, if a property is annotated with @PathSensitive(PathSensitivity.NAME\_ONLY), then moving the files around without changing their contents will not make the task out-of-date. |

| **✨** | | **\*** | In fact, File can be any type accepted by [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) and Iterable<File> can be any type accepted by [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...-), or [ProjectLayout.configurableFiles(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-). This includes instances of Callable, such as closures, allowing for lazy evaluation of the property values. Be aware that the types FileCollection and FileTree are Iterable<File>s. | | --- | --- | | **\*\*** | Similar to the above, File can be any type accepted by [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)). The Map itself can be wrapped in Callables, such as closures. | |
| --- | --- | --- | --- | --- | --- |

Annotations are inherited from all parent types including implemented interfaces. Property type annotations override any other property type annotation declared in a parent type. This way an @InputFile property can be turned into an @InputDirectory property in a child task type.

Annotations on a property declared in a type override similar annotations declared by the superclass and in any implemented interfaces. Superclass annotations take precedence over annotations declared in implemented interfaces.

The [Console](http://docs.google.com/javadoc/org/gradle/api/tasks/Console.html) and [Internal](http://docs.google.com/javadoc/org/gradle/api/tasks/Internal.html) annotations in the table are special cases as they don’t declare either task inputs or task outputs. So why use them? It’s so that you can take advantage of the [Java Gradle Plugin Development plugin](http://docs.google.com/java_gradle_plugin.html#java_gradle_plugin) to help you develop and publish your own plugins. This plugin checks whether any properties of your custom task classes lack an incremental build annotation. This protects you from forgetting to add an appropriate annotation during development.

[Using the classpath annotations](#3ygebqi)

Besides @InputFiles, for JVM-related tasks Gradle understands the concept of classpath inputs. Both runtime and compile classpaths are treated differently when Gradle is looking for changes.

As opposed to input properties annotated with @[InputFiles](http://docs.google.com/javadoc/org/gradle/api/tasks/InputFiles.html), for classpath properties the order of the entries in the file collection matter. On the other hand, the names and paths of the directories and jar files on the classpath itself are ignored. Timestamps and the order of class files and resources inside jar files on a classpath are ignored, too, thus recreating a jar file with different file dates will not make the task out of date.

Runtime classpaths are marked with @[Classpath](http://docs.google.com/javadoc/org/gradle/api/tasks/Classpath.html), and they offer further customization via [classpath normalization](#sqyw64).

Input properties annotated with @[CompileClasspath](http://docs.google.com/javadoc/org/gradle/api/tasks/CompileClasspath.html) are considered Java compile classpaths. Additionally to the aforementioned general classpath rules, compile classpaths ignore changes to everything but class files. Gradle uses the same class analysis described in [Java compile avoidance](http://docs.google.com/java_plugin.html#sec:java_compile_avoidance) to further filter changes that don’t affect the class' ABIs. This means that changes which only touch the implementation of classes do not make the task out of date.

[Nested inputs](#3cqmetx)

When analyzing @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) task properties for declared input and output sub-properties Gradle uses the type of the actual value. Hence it can discover all sub-properties declared by a runtime sub-type.

When adding @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) to a @[Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html), the value of the Provider is treated as a nested input.

When adding @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) to an iterable, each element is treated as a separate nested input. Each nested input in the iterable is assigned a name, which by default is the dollar sign followed by the index in the iterable, e.g. $2. If an element of the iterable implements [Named](http://docs.google.com/javadoc/org/gradle/api/Named.html), then the name is used as property name. The ordering of the elements in the iterable is crucial for for reliable up-to-date checks and caching if not all of the elements implement [Named](http://docs.google.com/javadoc/org/gradle/api/Named.html). Multiple elements which have the same name are not allowed.

When adding @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) to a map, then for each value a nested input is added, using the key as name.

The type and classpath of nested inputs is tracked, too. This ensures that changes to the implementation of a nested input causes the build to be out of date. By this it is also possible to add user provided code as an input, e.g. by annotating an @[Action](http://docs.google.com/javadoc/org/gradle/api/Action.html) property with @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html). Note that any inputs to such actions should be tracked, either by annotated properties on the action or by manually registering them with the task.

Using nested inputs allows richer modeling and extensibility for tasks, as e.g. shown by [Test.getJvmArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:jvmArgumentProviders).

This allows us to [model the JaCoCo Java agent](https://github.com/gradle/gradle/blob/7b047c7cbb4932743243a76123f5347be6d07856/subprojects/jacoco/src/main/java/org/gradle/testing/jacoco/plugins/JacocoPluginExtension.java#L138-L157), thus declaring the necessary JVM arguments and providing the inputs and outputs to Gradle:

**JacocoAgent.java**

class JacocoAgent implements CommandLineArgumentProvider {  
 private final JacocoTaskExtension jacoco;  
  
 public JacocoAgent(JacocoTaskExtension jacoco) {  
 this.jacoco = jacoco;  
 }  
  
 @Nested  
 @Optional  
 public JacocoTaskExtension getJacoco() {  
 return jacoco.isEnabled() ? jacoco : null;  
 }  
  
 @Override  
 public Iterable<String> asArguments() {  
 return jacoco.isEnabled() ? ImmutableList.of(jacoco.getAsJvmArg()) : Collections.<String>emptyList();  
 }  
}  
  
test.getJvmArgumentProviders().add(new JacocoAgent(extension));

For this to work, [JacocoTaskExtension](http://docs.google.com/dsl/org.gradle.testing.jacoco.plugins.JacocoTaskExtension.html) needs to have the correct input and output annotations.

The approach works for Test JVM arguments, since [Test.getJvmArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:jvmArgumentProviders) is an Iterable annotated with @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html).

There are other task types where this kind of nested inputs are available:

* [JavaExec.getArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.JavaExec.html#org.gradle.api.tasks.JavaExec:argumentProviders) - model e.g. custom tools
* [JavaExec.getJvmArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.JavaExec.html#org.gradle.api.tasks.JavaExec:jvmArgumentProviders) - used for Jacoco Java agent
* [CompileOptions.getCompilerArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.compile.CompileOptions.html#org.gradle.api.tasks.compile.CompileOptions:compilerArgumentProviders) - model e.g annotation processors
* [Exec.getArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.Exec.html#org.gradle.api.tasks.Exec:argumentProviders) - model e.g custom tools

In the same way, this kind of modelling is available to custom tasks.

[Runtime API](#1rvwp1q)

Custom task classes are an easy way to bring your own build logic into the arena of incremental build, but you don’t always have that option. That’s why Gradle also provides an alternative API that can be used with any tasks, which we look at next.

When you don’t have access to the source for a custom task class, there is no way to add any of the annotations we covered in the previous section. Fortunately, Gradle provides a runtime API for scenarios just like that. It can also be used for ad-hoc tasks, as you’ll see next.

[Using it for ad-hoc tasks](#4bvk7pj)

This runtime API is provided through a couple of aptly named properties that are available on every Gradle task:

* [Task.getInputs()](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:inputs) of type [TaskInputs](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskInputs.html)
* [Task.getOutputs()](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:outputs) of type [TaskOutputs](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskOutputs.html)
* [Task.getDestroyables()](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:destroyables) of type [TaskDestroyables](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskDestroyables.html)

These objects have methods that allow you to specify files, directories and values which constitute the task’s inputs and outputs. In fact, the runtime API has almost feature parity with the annotations. All it lacks is validation of whether declared files are actually files and declared directories are directories. Nor will it create output directories if they don’t exist. But that’s it.

Let’s take the template processing example from before and see how it would look as an ad-hoc task that uses the runtime API:

[Example: Ad-hoc task](#2r0uhxc)

**build.gradle**

task processTemplatesAdHoc {  
 inputs.property("engine", TemplateEngineType.FREEMARKER)  
 inputs.files(fileTree("src/templates"))  
 inputs.property("templateData.name", "docs")  
 inputs.property("templateData.variables", [year: 2013])  
 outputs.dir("$buildDir/genOutput2")  
  
 doLast {  
 // Process the templates here  
 }  
}

**Output of** gradle processTemplatesAdHoc

> gradle processTemplatesAdHoc  
> Task :processTemplatesAdHoc  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

As before, there’s much to talk about. To begin with, you should really write a custom task class for this as it’s a non-trivial implementation that has several configuration options. In this case, there are no task properties to store the root source folder, the location of the output directory or any of the other settings. That’s deliberate to highlight the fact that the runtime API doesn’t require the task to have any state. In terms of incremental build, the above ad-hoc task will behave the same as the custom task class.

All the input and output definitions are done through the methods on inputs and outputs, such as property(), files(), and dir(). Gradle performs up-to-date checks on the argument values to determine whether the task needs to run again or not. Each method corresponds to one of the incremental build annotations, for example inputs.property() maps to @Input and outputs.dir() maps to @OutputDirectory. The only difference is that the file(), files(), dir() and dirs() methods don’t validate the type of file object at the given path (file or directory), unlike the annotations.

The files that a task removes can be specified through destroyables.register().

[Example: Ad-hoc task declaring a destroyable](#1664s55)

**build.gradle**

task removeTempDir {  
 destroyables.register("$projectDir/tmpDir")  
 doLast {  
 delete("$projectDir/tmpDir")  
 }  
}

One notable difference between the runtime API and the annotations is the lack of a method that corresponds directly to @Nested. That’s why the example uses two property() declarations for the template data, one for each TemplateData property. You should utilize the same technique when using the runtime API with nested values. Any given task can either declare destroyables or inputs/outputs, but cannot declare both.

[Using it for custom task types](#3q5sasy)

Another type of example involves adding input and output definitions to instances of a custom task class that lacks the requisite annotations. For example, imagine that the ProcessTemplates task is provided by a plugin and that it’s missing the incremental build annotations. In order to make up for that deficiency, you can use the runtime API:

[Example: Using runtime API with custom task type](#25b2l0r)

**build.gradle**

task processTemplatesRuntime(type: ProcessTemplatesNoAnnotations) {  
 templateEngine = TemplateEngineType.FREEMARKER  
 sourceFiles = fileTree("src/templates")  
 templateData = new TemplateData("test", [year: 2014])  
 outputDir = file("$buildDir/genOutput3")  
  
 inputs.property("engine",templateEngine)  
 inputs.files(sourceFiles)  
 inputs.property("templateData.name", templateData.name)  
 inputs.property("templateData.variables", templateData.variables)  
 outputs.dir(outputDir)  
}

**Output of gradle processTemplatesRuntime**

> gradle processTemplatesRuntime  
> Task :processTemplatesRuntime  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

**Output of gradle processTemplatesRuntime (run again)**

> gradle processTemplatesRuntime  
> Task :processTemplatesRuntime UP-TO-DATE  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

As you can see, we can both configure the tasks properties and use those properties as arguments to the incremental build runtime API. Using the runtime API like this is a little like using doLast() and doFirst() to attach extra actions to a task, except in this case we’re attaching information about inputs and outputs. Note that if the task type is already using the incremental build annotations, the runtime API will add inputs and outputs rather than replace them.

[Fine-grained configuration](#kgcv8k)

The runtime API methods only allow you to declare your inputs and outputs in themselves. However, the file-oriented ones return a builder - of type [TaskInputFilePropertyBuilder](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskInputFilePropertyBuilder.html) - that lets you provide additional information about those inputs and outputs.

You can learn about all the options provided by the builder in its API documentation, but we’ll show you a simple example here to give you an idea of what you can do.

Let’s say we don’t want to run the processTemplates task if there are no source files, regardless of whether it’s a clean build or not. After all, if there are no source files, there’s nothing for the task to do. The builder allows us to configure this like so:

[Example: Using skipWhenEmpty() via the runtime API](#34g0dwd)

**build.gradle**

task processTemplatesRuntimeConf(type: ProcessTemplatesNoAnnotations) {  
 // ...  
 sourceFiles = fileTree("src/templates") {  
 include "\*\*/\*.fm"  
 }  
  
 inputs.files(sourceFiles).skipWhenEmpty()  
 // ...  
}

**Output of** gradle clean processTemplatesRuntimeConf

> gradle clean processTemplatesRuntimeConf  
> Task :processTemplatesRuntimeConf NO-SOURCE  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

The TaskInputs.files() method returns a builder that has a skipWhenEmpty() method. Invoking this method is equivalent to annotating to the property with [@SkipWhenEmpty](#3whwml4).

Prior to Gradle 3.0, you had to use the TaskInputs.source() and TaskInputs.sourceDir() methods to get the same behavior as with skipWhenEmpty(). These methods are now deprecated and should not be used with Gradle 3.0 and above.

Now that you have seen both the annotations and the runtime API, you may be wondering which API you should be using. Our recommendation is to use the annotations wherever possible, and it’s sometimes worth creating a custom task class just so that you can make use of them. The runtime API is more for situations in which you can’t use the annotations.

[Important beneficial side effects](#1jlao46)

Once you declare a task’s formal inputs and outputs, Gradle can then infer things about those properties. For example, if an input of one task is set to the output of another, that means the first task depends on the second, right? Gradle knows this and can act upon it.

We’ll look at this feature next and also some other features that come from Gradle knowing things about inputs and outputs.

[Inferred task dependencies](#43ky6rz)

Consider an archive task that packages the output of the processTemplates task. A build author will see that the archive task obviously requires processTemplates to run first and so may add an explicit dependsOn. However, if you define the archive task like so:

[Example: Inferred task dependency via task outputs](#2iq8gzs)

**build.gradle**

task packageFiles(type: Zip) {  
 from processTemplates.outputs  
}

**Output of** gradle clean packageFiles

> gradle clean packageFiles  
> Task :processTemplates  
> Task :packageFiles  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

Gradle will automatically make packageFiles depend on processTemplates. It can do this because it’s aware that one of the inputs of packageFiles requires the output of the processTemplates task. We call this an inferred task dependency.

The above example can also be written as

[Example: Inferred task dependency via a task argument](#xvir7l)

**build.gradle**

task packageFiles2(type: Zip) {  
 from processTemplates  
}

**Output of** gradle clean packageFiles2

> gradle clean packageFiles2  
> Task :processTemplates  
> Task :packageFiles2  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

This is because the from() method can accept a task object as an argument. Behind the scenes, from() uses the project.files() method to wrap the argument, which in turn exposes the task’s formal outputs as a file collection. In other words, it’s a special case!

[Input and output validation](#2dlolyb)

The incremental build annotations provide enough information for Gradle to perform some basic validation on the annotated properties. In particular, it does the following for each property before the task executes:

* @InputFile - verifies that the property has a value and that the path corresponds to a file (not a directory) that exists.
* @InputDirectory - same as for @InputFile, except the path must correspond to a directory.
* @OutputDirectory - verifies that the path doesn’t match a file and also creates the directory if it doesn’t already exist.

Such validation improves the robustness of the build, allowing you to identify issues related to inputs and outputs quickly.

You will occasionally want to disable some of this validation, specifically when an input file may validly not exist. That’s why Gradle provides the @Optional annotation: you use it to tell Gradle that a particular input is optional and therefore the build should not fail if the corresponding file or directory doesn’t exist.

[Continuous build](#3hv69ve)

Another benefit of defining task inputs and outputs is continuous build. Since Gradle knows what files a task depends on, it can automatically run a task again if any of its inputs change. By activating continuous build when you run Gradle - through the --continuous or -t options - you will put Gradle into a state in which it continually checks for changes and executes the requested tasks when it encounters such changes.

You can find out more about this feature in [Continuous build](http://docs.google.com/command_line_interface.html#sec:continuous_build).

[Task parallelism](#1x0gk37)

One last benefit of defining task inputs and outputs is that Gradle can use this information to make decisions about how to run tasks when the "--parallel" option is used. For instance, Gradle will inspect the outputs of tasks when selecting the next task to run and will avoid concurrent execution of tasks that write to the same output directory. Similarly, Gradle will use the information about what files a task destroys (e.g. specified by the Destroys annotation) and avoid running a task that removes a set of files while another task is running that consumes or creates those same files (and vice versa). It can also determine that a task that creates a set of files has already run and that a task that consumes those files has yet to run and will avoid running a task that removes those files in between. By providing task input and output information in this way, Gradle can infer creation/consumption/destruction relationships between tasks and can ensure that task execution does not violate those relationships.

[How does it work?](#1egqt2p)

Before a task is executed for the first time, Gradle takes a snapshot of the inputs. This snapshot contains the paths of input files and a hash of the contents of each file. Gradle then executes the task. If the task completes successfully, Gradle takes a snapshot of the outputs. This snapshot contains the set of output files and a hash of the contents of each file. Gradle persists both snapshots for the next time the task is executed.

Each time after that, before the task is executed, Gradle takes a new snapshot of the inputs and outputs. If the new snapshots are the same as the previous snapshots, Gradle assumes that the outputs are up to date and skips the task. If they are not the same, Gradle executes the task. Gradle persists both snapshots for the next time the task is executed.

Gradle also considers the *code* of the task as part of the inputs to the task. When a task, its actions, or its dependencies change between executions, Gradle considers the task as out-of-date.

Gradle understands if a file property (e.g. one holding a Java classpath) is order-sensitive. When comparing the snapshot of such a property, even a change in the order of the files will result in the task becoming out-of-date.

Note that if a task has an output directory specified, any files added to that directory since the last time it was executed are ignored and will NOT cause the task to be out of date. This is so unrelated tasks may share an output directory without interfering with each other. If this is not the behaviour you want for some reason, consider using [TaskOutputs.upToDateWhen(groovy.lang.Closure)](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskOutputs.html#upToDateWhen-groovy.lang.Closure-)

The inputs for the task are also used to calculate the [build cache](http://docs.google.com/build_cache.html#build_cache) key used to load task outputs when enabled. For more details see [Task output caching](http://docs.google.com/build_cache.html#sec:task_output_caching).

[Advanced techniques](#4h042r0)

Everything you’ve seen so far in this section will cover most of the use cases you’ll encounter, but there are some scenarios that need special treatment. We’ll present a few of those next with the appropriate solutions.

[Adding your own cached input/output methods](#2w5ecyt)

Have you ever wondered how the from() method of the Copy task works? It’s not annotated with @InputFiles and yet any files passed to it are treated as formal inputs of the task. What’s happening?

The implementation is quite simple and you can use the same technique for your own tasks to improve their APIs. Write your methods so that they add files directly to the appropriate annotated property. As an example, here’s how to add a sources() method to the custom ProcessTemplates class we introduced earlier:

[Example: Declaring a method to add task inputs](#1baon6m)

**ProcessTemplates.java**

public class ProcessTemplates extends DefaultTask {  
 // ...  
 private FileCollection sourceFiles = getProject().getLayout().files();  
  
 @SkipWhenEmpty  
 @InputFiles  
 @PathSensitive(PathSensitivity.NONE)  
 public FileCollection getSourceFiles() {  
 return this.sourceFiles;  
 }  
  
 public void sources(FileCollection sourceFiles) {  
 this.sourceFiles = this.sourceFiles.plus(sourceFiles);  
 }  
  
 // ...  
}

**build.gradle**

task processTemplates(type: ProcessTemplates) {  
 templateEngine = TemplateEngineType.FREEMARKER  
 templateData = new TemplateData("test", [year: 2012])  
 outputDir = file("$buildDir/genOutput")  
  
 sources fileTree("src/templates")  
}

**Output of gradle processTemplates**

> gradle processTemplates  
> Task :processTemplates  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

In other words, as long as you add values and files to formal task inputs and outputs during the configuration phase, they will be treated as such regardless from where in the build you add them.

If we want to support tasks as arguments as well and treat their outputs as the inputs, we can use the project.layout.files() method like so:

[Example: Declaring a method to add a task as an input](#3vac5uf)

**ProcessTemplates.java**

// ...  
 public void sources(Task inputTask) {  
 this.sourceFiles = this.sourceFiles.plus(getProject().getLayout().files(inputTask));  
 }  
 // ...

**build.gradle**

task copyTemplates(type: Copy) {  
 into "$buildDir/tmp"  
 from "src/templates"  
}  
  
task processTemplates2(type: ProcessTemplates) {  
 // ...  
 sources copyTemplates  
}

**Output of gradle processTemplates2**

> gradle processTemplates2  
> Task :copyTemplates  
> Task :processTemplates2  
  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

This technique can make your custom task easier to use and result in cleaner build files. As an added benefit, our use of getProject().getLayout().files() means that our custom method can set up an inferred task dependency.

One last thing to note: if you are developing a task that takes collections of source files as inputs, like this example, consider using the built-in [SourceTask](http://docs.google.com/dsl/org.gradle.api.tasks.SourceTask.html). It will save you having to implement some of the plumbing that we put into ProcessTemplates.

[Linking an @OutputDirectory to an @InputFiles](#2afmg28)

When you want to link the output of one task to the input of another, the types often match and a simple property assignment will provide that link. For example, a File output property can be assigned to a File input.

Unfortunately, this approach breaks down when you want the files in a task’s @OutputDirectory (of type File) to become the source for another task’s @InputFiles property (of type FileCollection). Since the two have different types, property assignment won’t work.

As an example, imagine you want to use the output of a Java compilation task - via the destinationDir property - as the input of a custom task that instruments a set of files containing Java bytecode. This custom task, which we’ll call Instrument, has a classFiles property annotated with @InputFiles. You might initially try to configure the task like so:

[Example: Failed attempt at setting up an inferred task dependency](#pkwqa1)

**build.gradle**

apply plugin: "java"  
  
task badInstrumentClasses(type: Instrument) {  
 classFiles = fileTree(compileJava.destinationDir)  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean badInstrumentClasses

> gradle clean badInstrumentClasses  
> Task :clean UP-TO-DATE  
> Task :badInstrumentClasses NO-SOURCE  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

There’s nothing obviously wrong with this code, but you can see from the console output that the compilation task is missing. In this case you would need to add an explicit task dependency between instrumentClasses and compileJava via dependsOn. The use of fileTree() means that Gradle can’t infer the task dependency itself.

One solution is to use the TaskOutputs.files property, as demonstrated by the following example:

[Example: Setting up an inferred task dependency between output dir and input files](#39kk8xu)

**build.gradle**

task instrumentClasses(type: Instrument) {  
 classFiles = compileJava.outputs.files  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean instrumentClasses

> gradle clean instrumentClasses  
> Task :clean UP-TO-DATE  
> Task :compileJava  
> Task :instrumentClasses  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

Alternatively, you can get Gradle to access the appropriate property itself by using one of project.files(), project.layout.files() or project.layout.configurableFiles() in place of project.fileTree():

[Example: Setting up an inferred task dependency with filesFor()](#1opuj5n)

**build.gradle**

task instrumentClasses2(type: Instrument) {  
 classFiles = layout.files(compileJava)  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean instrumentClasses2

> gradle clean instrumentClasses2  
> Task :clean UP-TO-DATE  
> Task :compileJava  
> Task :instrumentClasses2  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

Remember that files(), layout.files() and layout.configurableFiles() can take tasks as arguments, whereas fileTree() cannot.

The downside of this approach is that all file outputs of the source task become the input files of the target - instrumentClasses in this case. That’s fine as long as the source task only has a single file-based output, like the JavaCompile task. But if you have to link just one output property among several, then you need to explicitly tell Gradle which task generates the input files using the builtBy method:

[Example: Setting up an inferred task dependency with builtBy()](#48pi1tg)

**build.gradle**

task instrumentClassesBuiltBy(type: Instrument) {  
 classFiles = fileTree(compileJava.destinationDir) {  
 builtBy compileJava  
 }  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean instrumentClassesBuiltBy

> gradle clean instrumentClassesBuiltBy  
> Task :clean UP-TO-DATE  
> Task :compileJava  
> Task :instrumentClassesBuiltBy  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

You can of course just add an explicit task dependency via dependsOn, but the above approach provides more semantic meaning, explaining why compileJava has to run beforehand.

[Providing custom up-to-date logic](#2nusc19)

Gradle automatically handles up-to-date checks for output files and directories, but what if the task output is something else entirely? Perhaps it’s an update to a web service or a database table. Gradle has no way of knowing how to check whether the task is up to date in such cases.

That’s where the upToDateWhen() method on TaskOutputs comes in. This takes a predicate function that is used to determine whether a task is up to date or not. One use case is to disable up-to-date checks completely for a task, like so:

[Example: Ignoring up-to-date checks](#1302m92)

**build.gradle**

task alwaysInstrumentClasses(type: Instrument) {  
 classFiles = layout.files(compileJava)  
 destinationDir = file("$buildDir/instrumented")  
 outputs.upToDateWhen { false }  
}

**Output of gradle clean alwaysInstrumentClasses**

> gradle clean alwaysInstrumentClasses  
> Task :compileJava  
> Task :alwaysInstrumentClasses  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

**Output of gradle alwaysInstrumentClasses**

> gradle alwaysInstrumentClasses  
> Task :compileJava UP-TO-DATE  
> Task :alwaysInstrumentClasses  
  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 1 executed, 1 up-to-date

The { false } closure ensures that copyResources will always perform the copy, irrespective of whether there is no change in the inputs or outputs.

You can of course put more complex logic into the closure. You could check whether a particular record in a database table exists or has changed for example. Just be aware that up-to-date checks should *save* you time. Don’t add checks that cost as much or more time than the standard execution of the task. In fact, if a task ends up running frequently anyway, because it’s rarely up to date, then it may not be worth having an up-to-date check at all. Remember that your checks will always run if the task is in the execution task graph.

One common mistake is to use upToDateWhen() instead of Task.onlyIf(). If you want to skip a task on the basis of some condition unrelated to the task inputs and outputs, then you should use onlyIf(). For example, in cases where you want to skip a task when a particular property is set or not set.

[Configure input normalization](#sqyw64)

For up to date checks and the [build cache](http://docs.google.com/build_cache.html#build_cache) Gradle needs to determine if two task input properties have the same value. In order to do so, Gradle first normalizes both inputs and then compares the result. For example, for a compile classpath, Gradle extracts the ABI signature from the classes on the classpath and then compares signatures between the last Gradle run and the current Gradle run as described in [Java compile avoidance](http://docs.google.com/java_plugin.html#sec:java_compile_avoidance).

It is possible to customize Gradle’s built-in strategy for runtime classpath normalization. All inputs annotated with @[Classpath](http://docs.google.com/javadoc/org/gradle/api/tasks/Classpath.html) are considered to be runtime classpaths.

Let’s say you want to add a file build-info.properties to all your produced jar files which contains information about the build, e.g. the timestamp when the build started or some ID to identify the CI job that published the artifact. This file is only for auditing purposes, and has no effect on the outcome of running tests. Nonetheless, this file is part of the runtime classpath for the test task and changes on every build invocation. Therefore, the test would be never up-to-date or pulled from the build cache. In order to benefit from incremental builds again, you are able tell Gradle to ignore this file on the runtime classpath at the project level by using [Project.normalization(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:normalization(org.gradle.api.Action)):

[Example: Runtime classpath normalization](#3mzq4wv)

**build.gradle**

normalization {  
 runtimeClasspath {  
 ignore 'build-info.properties'  
 }  
}

The effect of this configuration would be that changes to build-info.properties would be ignored for up-to-date checks and [build cache](http://docs.google.com/build_cache.html#build_cache) key calculations. Note that this will not change the runtime behavior of the test task - i.e. any test is still able to load build-info.properties and the runtime classpath is still the same as before.

[Stale task outputs](#2250f4o)

When the Gradle version changes, Gradle detects that outputs from tasks that ran with older versions of Gradle need to be removed to ensure that the newest version of the tasks are starting from a known clean state.

| **✨** | Automatic clean-up of stale output directories has only been implemented for the output of source sets (Java/Groovy/Scala compilation). |
| --- | --- |

[Task rules](#3j2qqm3)

Sometimes you want to have a task whose behavior depends on a large or infinite number value range of parameters. A very nice and expressive way to provide such tasks are task rules:

[Example: Task rule](#haapch)

**build.gradle**

tasks.addRule("Pattern: ping<ID>") { String taskName ->  
 if (taskName.startsWith("ping")) {  
 task(taskName) {  
 doLast {  
 println "Pinging: " + (taskName - 'ping')  
 }  
 }  
 }  
}

**Output of** gradle -q pingServer1

> gradle -q pingServer1  
Pinging: Server1

The String parameter is used as a description for the rule, which is shown with gradle tasks.

Rules are not only used when calling tasks from the command line. You can also create dependsOn relations on rule based tasks:

[Example: Dependency on rule based tasks](#319y80a)

**build.gradle**

tasks.addRule("Pattern: ping<ID>") { String taskName ->  
 if (taskName.startsWith("ping")) {  
 task(taskName) {  
 doLast {  
 println "Pinging: " + (taskName - 'ping')  
 }  
 }  
 }  
}  
  
task groupPing {  
 dependsOn pingServer1, pingServer2  
}

**Output of** gradle -q groupPing

> gradle -q groupPing  
Pinging: Server1  
Pinging: Server2

If you run “gradle -q tasks” you won’t find a task named “pingServer1” or “pingServer2”, but this script is executing logic based on the request to run those tasks.

[Finalizer tasks](#1y810tw)

| **✨** | Finalizers tasks are an *incubating* feature (see [more about the Gradle feature lifecycle](http://docs.google.com/feature_lifecycle.html#sec:incubating_state)). |
| --- | --- |

Finalizer tasks are automatically added to the task graph when the finalized task is scheduled to run.

[Example: Adding a task finalizer](#1gf8i83)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
  
taskX.finalizedBy taskY

**Output of** gradle -q taskX

> gradle -q taskX  
taskX  
taskY

Finalizer tasks will be executed even if the finalized task fails.

[Example: Task finalizer for a failing task](#40ew0vw)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 throw new RuntimeException()  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
  
taskX.finalizedBy taskY

**Output of** gradle -q taskX

> gradle -q taskX  
taskX  
taskY  
  
FAILURE: Build failed with an exception.  
  
\* Where:  
Build file '/home/user/gradle/samples/build.gradle' line: 4  
  
\* What went wrong:  
Execution failed for task ':taskX'.  
> java.lang.RuntimeException (no error message)  
  
\* Try:  
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get full insights.  
  
\* Get more help at https://help.gradle.org  
  
BUILD FAILED in 0s

On the other hand, finalizer tasks are not executed if the finalized task didn’t do any work, for example if it is considered up to date or if a dependent task fails.

Finalizer tasks are useful in situations where the build creates a resource that has to be cleaned up regardless of the build failing or succeeding. An example of such a resource is a web container that is started before an integration test task and which should be always shut down, even if some of the tests fail.

To specify a finalizer task you use the [Task.finalizedBy(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:finalizedBy(java.lang.Object%5B%5D)) method. This method accepts a task instance, a task name, or any other input accepted by [Task.dependsOn(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:dependsOn(java.lang.Object%5B%5D)).

[Lifecycle tasks](#4i7ojhp)

Lifecycle tasks are tasks that do not do work themselves. They typically do not have any task actions. Lifecycle tasks can represent several concepts:

* a work-flow step (e.g., run all checks with check)
* a buildable thing (e.g., create a debug 32-bit executable for native components with debug32MainExecutable)
* a convenience task to execute many of the same logical tasks (e.g., run all compilation tasks with compileAll)

Many Gradle plug-ins define their own lifecycle tasks to make it convenient to do specific things. When developing your own plugins, you should consider using your own lifecycle tasks or hooking into some of the tasks already provided by Gradle. See the Java plugin [tasks](http://docs.google.com/java_plugin.html#sec:java_tasks) for an example.

Unless a lifecycle task has actions, its outcome is determined by its dependencies. If any of the task’s dependencies are executed, the lifecycle task will be considered executed. If all of the task’s dependencies are up-to-date, skipped or from cache, the lifecycle task will be considered up-to-date.

[Summary](#2xcytpi)

If you are coming from Ant, an enhanced Gradle task like *Copy* seems like a cross between an Ant target and an Ant task. Although Ant’s tasks and targets are really different entities, Gradle combines these notions into a single entity. Simple Gradle tasks are like Ant’s targets, but enhanced Gradle tasks also include aspects of Ant tasks. All of Gradle’s tasks share a common API and you can create dependencies between them. These tasks are much easier to configure than an Ant task. They make full use of the type system, and are more expressive and easier to maintain.

[1](#2fk6b3p). You might be wondering why there is neither an import for the StopExecutionException nor do we access it via its fully qualified name. The reason is, that Gradle adds a set of default imports to your script (see [Default imports](http://docs.google.com/writing_build_scripts.html#script-default-imports)).

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/)