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/)

# Writing Custom Plugins

Contents

[Packaging a plugin](#4d34og8)

[Writing a simple plugin](#2s8eyo1)

[Making the plugin configurable](#17dp8vu)

[Working with files in custom tasks and plugins](#3rdcrjn)

[Mapping extension properties to task properties](#26in1rg)

[A standalone project](#lnxbz9)

[Providing a configuration DSL for the plugin](#35nkun2)

A Gradle plugin packages up reusable pieces of build logic, which can be used across many different projects and builds. Gradle allows you to implement your own plugins, so you can reuse your build logic, and share it with others.

You can implement a Gradle plugin in any language you like, provided the implementation ends up compiled as bytecode. In our examples, we are going to use Groovy as the implementation language. Groovy, Java or Kotlin are all good choices as the language to use to implement a plugin, as the Gradle API has been designed to work well with these languages. In general, a plugin implemented using Java or Kotlin, which are statically typed, will perform better than the same plugin implemented using Groovy.

[Packaging a plugin](#4d34og8)

There are several places where you can put the source for the plugin.

Build script

You can include the source for the plugin directly in the build script. This has the benefit that the plugin is automatically compiled and included in the classpath of the build script without you having to do anything. However, the plugin is not visible outside the build script, and so you cannot reuse the plugin outside the build script it is defined in.

buildSrc project

You can put the source for the plugin in the *rootProjectDir*/buildSrc/src/main/groovy directory. Gradle will take care of compiling and testing the plugin and making it available on the classpath of the build script. The plugin is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the plugin outside the build it is defined in.

See [Organizing Gradle Projects](http://docs.google.com/organizing_gradle_projects.html#organizing_gradle_projects) for more details about the buildSrc project.

Standalone project

You can create a separate project for your plugin. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the plugin in the build script, to keep things simple. Then we will look at creating a standalone project.

[Writing a simple plugin](#2s8eyo1)

To create a Gradle plugin, you need to write a class that implements the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) interface. When the plugin is applied to a project, Gradle creates an instance of the plugin class and calls the instance’s [Plugin.apply()](http://docs.google.com/javadoc/org/gradle/api/Plugin.html#apply-T-) method. The project object is passed as a parameter, which the plugin can use to configure the project however it needs to. The following sample contains a greeting plugin, which adds a hello task to the project.

[Example: A custom plugin](#1ksv4uv)

**build.gradle**

class GreetingPlugin implements Plugin<Project> {  
 void apply(Project project) {  
 project.task('hello') {  
 doLast {  
 println 'Hello from the GreetingPlugin'  
 }  
 }  
 }  
}  
  
// Apply the plugin  
apply plugin: GreetingPlugin

**Output of** gradle -q hello

> gradle -q hello  
Hello from the GreetingPlugin

One thing to note is that a new instance of a plugin is created for each project it is applied to. Also note that the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) class is a generic type. This example has it receiving the [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) type as a type parameter. A plugin can instead receive a parameter of type [Settings](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html), in which case the plugin can be applied in a settings script, or a parameter of type [Gradle](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html), in which case the plugin can be applied in an initialization script.

[Making the plugin configurable](#17dp8vu)

Most plugins need to obtain some configuration from the build script. One method for doing this is to use *extension objects*. The Gradle [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) has an associated [ExtensionContainer](http://docs.google.com/javadoc/org/gradle/api/plugins/ExtensionContainer.html) object that contains all the settings and properties for the plugins that have been applied to the project. You can provide configuration for your plugin by adding an extension object to this container. An extension object is simply a Java Bean compliant class. Groovy is a good language choice to implement an extension object because plain old Groovy objects contain all the getter and setter methods that a Java Bean requires. Java and Kotlin are other good choices.

Let’s add a simple extension object to the project. Here we add a greeting extension object to the project, which allows you to configure the greeting.

[Example: A custom plugin extension](#44sinio)

**build.gradle**

class GreetingPluginExtension {  
 String message = 'Hello from GreetingPlugin'  
}  
  
class GreetingPlugin implements Plugin<Project> {  
 void apply(Project project) {  
 // Add the 'greeting' extension object  
 def extension = project.extensions.create('greeting', GreetingPluginExtension)  
 // Add a task that uses configuration from the extension object  
 project.task('hello') {  
 doLast {  
 println extension.message  
 }  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
// Configure the extension  
greeting.message = 'Hi from Gradle'

**Output of** gradle -q hello

> gradle -q hello  
Hi from Gradle

In this example, GreetingPluginExtension is a plain old Groovy object with a property called message. The extension object is added to the plugin list with the name greeting. This object then becomes available as a project property with the same name as the extension object.

Oftentimes, you have several related properties you need to specify on a single plugin. Gradle adds a configuration closure block for each extension object, so you can group settings together. The following example shows you how this works.

[Example: A custom plugin with configuration closure](#2jxsxqh)

**build.gradle**

class GreetingPluginExtension {  
 String message  
 String greeter  
}  
  
class GreetingPlugin implements Plugin<Project> {  
 void apply(Project project) {  
 def extension = project.extensions.create('greeting', GreetingPluginExtension)  
 project.task('hello') {  
 doLast {  
 println "${extension.message} from ${extension.greeter}"  
 }  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
// Configure the extension using a DSL block  
greeting {  
 message = 'Hi'  
 greeter = 'Gradle'  
}

**Output of** gradle -q hello

> gradle -q hello  
Hi from Gradle

In this example, several settings can be grouped together within the greeting closure. The name of the closure block in the build script (greeting) needs to match the extension object name. Then, when the closure is executed, the fields on the extension object will be mapped to the variables within the closure based on the standard Groovy closure delegate feature.

[Working with files in custom tasks and plugins](#3rdcrjn)

When developing custom tasks and plugins, it’s a good idea to be very flexible when accepting input configuration for file locations. To do this, you can leverage 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 to resolve values to files as late as possible.

[Example: Evaluating file properties lazily](#z337ya)

**build.gradle**

class GreetingToFileTask extends DefaultTask {  
  
 def destination  
  
 File getDestination() {  
 project.file(destination)  
 }  
  
 @TaskAction  
 def greet() {  
 def file = getDestination()  
 file.parentFile.mkdirs()  
 file.write 'Hello!'  
 }  
}  
task greet(type: GreetingToFileTask) {  
 destination = { project.greetingFile }  
}  
  
task sayGreeting(dependsOn: greet) {  
 doLast {  
 println file(greetingFile).text  
 }  
}  
  
ext.greetingFile = "$buildDir/hello.txt"

**Output of** gradle -q sayGreeting

> gradle -q sayGreeting  
Hello!

In this example, we configure the greet task destination property as a closure, which is evaluated 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 to turn the return value of the closure into a File object at the last minute. You will notice that in the example above we specify the greetingFile property value after we have configured to use it for the task. This kind of lazy evaluation is a key benefit of accepting any value when setting a file property, then resolving that value when reading the property.

[Mapping extension properties to task properties](#26in1rg)

Capturing user input from the build script through an extension and mapping it to input/output properties of a custom task is considered a best practice. The end user only interacts with the exposed DSL defined by the extension. The imperative logic is hidden in the plugin implementation.

The extension declaration in the build script as well as the mapping between extension properties and custom task properties occurs during Gradle’s configuration phase of the build lifecycle. To avoid evaluation order issues, the actual value of a mapped property has to be resolved during the execution phase. For more information please see [the section on build phases](http://docs.google.com/build_lifecycle.html#sec:build_phases). Gradle’s API offers types for representing a property that should be lazily evaluated e.g. during execution time. Refer to [Lazy Configuration](http://docs.google.com/lazy_configuration.html#lazy_configuration) for more information.

The following demonstrates the usage of the type for mapping an extension property to a task property:

[Example: Mapping extension properties to task properties](#3j2qqm3)

**build.gradle**

class GreetingPlugin implements Plugin<Project> {  
 void apply(Project project) {  
 def extension = project.extensions.create('greeting', GreetingPluginExtension, project)  
 project.tasks.create('hello', Greeting) {  
 message = extension.message  
 outputFiles = extension.outputFiles  
 }  
 }  
}  
  
class GreetingPluginExtension {  
 final Property<String> message  
 final ConfigurableFileCollection outputFiles  
  
 GreetingPluginExtension(Project project) {  
 message = project.objects.property(String)  
 message.set('Hello from GreetingPlugin')  
 outputFiles = project.layout.configurableFiles()  
 }  
  
 void setOutputFiles(FileCollection outputFiles) {  
 this.outputFiles.setFrom(outputFiles)  
 }  
}  
  
class Greeting extends DefaultTask {  
 final Property<String> message = project.objects.property(String)  
 final ConfigurableFileCollection outputFiles = project.layout.configurableFiles()  
  
 void setOutputFiles(FileCollection outputFiles) {  
 this.outputFiles.setFrom(outputFiles)  
 }  
  
 @TaskAction  
 void printMessage() {  
 outputFiles.each {  
 logger.quiet "Writing message 'Hi from Gradle' to file"  
 it.text = message.get()  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
greeting {  
 message = 'Hi from Gradle'  
 outputFiles = layout.files('a.txt', 'b.txt')  
}

| **✨** | The code for this example can be found at samples/userguide/tasks/mapExtensionPropertiesToTaskProperties in the ‘-all’ distribution of Gradle. |
| --- | --- |

**Output of gradle -q hello**

> gradle -q hello  
Writing message 'Hi from Gradle' to file  
Writing message 'Hi from Gradle' to file

[A standalone project](#lnxbz9)

Now we will move our plugin to a standalone project, so we can publish it and share it with others. This project is simply a Groovy project that produces a JAR containing the plugin classes. Here is a simple build script for the project. It applies the Groovy plugin, and adds the Gradle API as a compile-time dependency.

[Example: A build for a custom plugin](#1y810tw)

**build.gradle**

plugins {  
 id 'groovy'  
}  
  
dependencies {  
 compile gradleApi()  
 compile localGroovy()  
}

| **✨** | The code for this example can be found at samples/customPlugin/plugin in the ‘-all’ distribution of Gradle. |
| --- | --- |

So how does Gradle find the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) implementation? The answer is you need to provide a properties file in the jar’s META-INF/gradle-plugins directory that matches the id of your plugin.

[Example: Wiring for a custom plugin](#4i7ojhp)

**src/main/resources/META-INF/gradle-plugins/org.samples.greeting.properties**

implementation-class=org.gradle.GreetingPlugin

Notice that the properties filename matches the plugin id and is placed in the resources folder, and that the implementation-class property identifies the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) implementation class.

[Creating a plugin id](#2xcytpi)

Plugin ids are fully qualified in a manner similar to Java packages (i.e. a reverse domain name). This helps to avoid collisions and provides a way to group plugins with similar ownership.

Your plugin id should be a combination of components that reflect namespace (a reasonable pointer to you or your organization) and the name of the plugin it provides. For example if you had a Github account named "foo" and your plugin was named "bar", a suitable plugin id might be com.github.foo.bar. Similarly, if the plugin was developed at the baz organization, the plugin id might be org.baz.bar.

Plugin ids should conform to the following:

* May contain any alphanumeric character, '.', and '-'.
* Must contain at least one '.' character separating the namespace from the name of the plugin.
* Conventionally use a lowercase reverse domain name convention for the namespace.
* Conventionally use only lowercase characters in the name.
* org.gradle and com.gradleware namespaces may not be used.
* Cannot start or end with a '.' character.
* Cannot contain consecutive '.' characters (i.e. '..').

Although there are conventional similarities between plugin ids and package names, package names are generally more detailed than is necessary for a plugin id. For instance, it might seem reasonable to add "gradle" as a component of your plugin id, but since plugin ids are only used for Gradle plugins, this would be superfluous. Generally, a namespace that identifies ownership and a name are all that are needed for a good plugin id.

[Publishing your plugin](#1ci93xb)

If you are publishing your plugin internally for use within your organization, you can publish it like any other code artifact. See the [Ivy](http://docs.google.com/publishing_ivy.html#publishing_ivy) and [Maven](http://docs.google.com/publishing_maven.html#publishing_maven) chapters on publishing artifacts.

If you are interested in publishing your plugin to be used by the wider Gradle community, you can publish it to the [Gradle Plugin Portal](http://plugins.gradle.org). This site provides the ability to search for and gather information about plugins contributed by the Gradle community. Please refer to the corresponding [guide](https://guides.gradle.org/publishing-plugins-to-gradle-plugin-portal/) on how to make your plugin available on this site.

[Using your plugin in another project](#3whwml4)

To use a plugin in a build script, you need to add the plugin classes to the build script’s classpath. To do this, you use a “buildscript { }” block, as described in see [Applying plugins using the buildscript block](http://docs.google.com/plugins.html#sec:applying_plugins_buildscript). The following example shows how you might do this when the JAR containing the plugin has been published to a local repository:

[Example: Using a custom plugin in another project](#2bn6wsx)

**build.gradle**

buildscript {  
 repositories {  
 maven {  
// END SNIPPET use-plugin  
// END SNIPPET use-task  
 def producerName = findProperty('producerName') ?: 'plugin'  
 def repoLocation = "../$producerName/build/repo"  
// START SNIPPET use-plugin  
// START SNIPPET use-task  
 url = uri(repoLocation)  
 }  
 }  
 dependencies {  
 classpath group: 'org.gradle', name: 'customPlugin',  
 version: '1.0-SNAPSHOT'  
 }  
}  
apply plugin: 'org.samples.greeting'

Alternatively, if your plugin is published to the plugin portal, you can use the incubating plugins DSL (see [Applying plugins using the plugins DSL](http://docs.google.com/plugins.html#sec:plugins_block)) to apply the plugin:

[Example: Applying a community plugin with the plugins DSL](#qsh70q)

**build.gradle**

plugins {  
 id 'com.jfrog.bintray' version '0.4.1'  
}

[Writing tests for your plugin](#3as4poj)

You can use the [ProjectBuilder](http://docs.google.com/javadoc/org/gradle/testfixtures/ProjectBuilder.html) class to create [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) instances to use when you test your plugin implementation.

[Example: Testing a custom plugin](#1pxezwc)

**src/test/groovy/org/gradle/GreetingPluginTest.groovy**

class GreetingPluginTest {  
 @Test  
 public void greeterPluginAddsGreetingTaskToProject() {  
 Project project = ProjectBuilder.builder().build()  
 project.pluginManager.apply 'org.samples.greeting'  
  
 assertTrue(project.tasks.hello instanceof GreetingTask)  
 }  
}

[Using the Java Gradle Plugin Development Plugin](#49x2ik5)

You can use the incubating [Java Gradle Plugin Development Plugin](http://docs.google.com/java_gradle_plugin.html#java_gradle_plugin) to eliminate some of the boilerplate declarations in your build script and provide some basic validations of plugin metadata. This plugin will automatically apply the [Java Plugin](http://docs.google.com/java_plugin.html#java_plugin), add the gradleApi() dependency to the compile configuration, and perform plugin metadata validations as part of the jar task execution, and generate plugin descriptors in the resulting JAR’s META-INF directory.

[Example: Using the Java Gradle Plugin Development plugin](#2p2csry)

**build.gradle**

plugins {  
 id 'java-gradle-plugin'  
 id 'groovy'  
}  
  
gradlePlugin {  
 plugins {  
 simplePlugin {  
 id = 'org.samples.greeting'  
 implementationClass = 'org.gradle.GreetingPlugin'  
 }  
 }  
}

When publishing plugins to custom plugin repositories using the [Ivy](http://docs.google.com/publishing_ivy.html#publishing_ivy) or [Maven](http://docs.google.com/publishing_maven.html#publishing_maven) publish plugins, the [Java Gradle Plugin Development Plugin](http://docs.google.com/java_gradle_plugin.html#java_gradle_plugin) will also generate plugin marker artifacts named based on the plugin id which depend on the plugin’s implementation artifact.

[Providing a configuration DSL for the plugin](#35nkun2)

As we saw above, you can use an extension object to provide configuration for your plugin. Using an extension object also extends the Gradle DSL to add a project property and DSL block for the plugin. An extension object is simply a regular object, and so you can provide DSL elements nested inside this block by adding properties and methods to the extension object.

Gradle provides several conveniences to help create a well-behaved DSL for your plugin.

[Nested DSL elements](#147n2zr)

When Gradle creates a task or extension object, Gradle *decorates* the implementation class to mix in DSL support. To create a nested DSL element you can use the [ObjectFactory](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html) type to create objects that are similarly decorated. These decorated objects can then be made visible to the DSL through properties and methods of the plugin’s extension:

[Example: Nested DSL elements](#3o7alnk)

**build.gradle**

class Person {  
 String name  
}  
  
class GreetingPluginExtension {  
 String message  
 final Person greeter  
  
 @javax.inject.Inject  
 GreetingPluginExtension(ObjectFactory objectFactory) {  
 // Create a Person instance  
 greeter = objectFactory.newInstance(Person)  
 }  
  
 void greeter(Action<? super Person> action) {  
 action.execute(greeter)  
 }  
}  
  
class GreetingPlugin implements Plugin<Project> {  
 void apply(Project project) {  
 // Create the extension, passing in an ObjectFactory for it to use  
 def extension = project.extensions.create('greeting', GreetingPluginExtension, project.objects)  
 project.task('hello') {  
 doLast {  
 println "${extension.message} from ${extension.greeter.name}"  
 }  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
greeting {  
 message = 'Hi'  
 greeter {  
 name = 'Gradle'  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
Hi from Gradle

In this example, the plugin passes the project’s ObjectFactory to the extension object through its constructor. The constructor uses this to create a nested object and makes this object available to the DSL through the greeter property.

[Configuring a collection of objects](#23ckvvd)

Gradle provides some utility classes for maintaining collections of objects, intended to work well with the Gradle DSL.

[Example: Managing a collection of objects](#ihv636)

**build.gradle**

class Book {  
 final String name  
 File sourceFile  
  
 Book(String name) {  
 this.name = name  
 }  
}  
  
class DocumentationPlugin implements Plugin<Project> {  
 void apply(Project project) {  
 // Create a container of Book instances  
 def books = project.container(Book)  
 books.all {  
 sourceFile = project.file("src/docs/$name")  
 }  
 // Add the container as an extension object  
 project.extensions.books = books  
 }  
}  
  
apply plugin: DocumentationPlugin  
  
// Configure the container  
books {  
 quickStart {  
 sourceFile = file('src/docs/quick-start')  
 }  
 userGuide {  
  
 }  
 developerGuide {  
  
 }  
}  
  
task books {  
 doLast {  
 books.each { book ->  
 println "$book.name -> $book.sourceFile"  
 }  
 }  
}

**Output of** gradle -q books

> gradle -q books  
developerGuide -> /home/user/gradle/samples/src/docs/developerGuide  
quickStart -> /home/user/gradle/samples/src/docs/quick-start  
userGuide -> /home/user/gradle/samples/src/docs/userGuide

The [Project.container(java.lang.Class)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:container(java.lang.Class)) methods create instances of [NamedDomainObjectContainer](http://docs.google.com/dsl/org.gradle.api.NamedDomainObjectContainer.html), that have many useful methods for managing and configuring the objects. In order to use a type with any of the project.container methods, it MUST expose a property named “name” as the unique, and constant, name for the object. The project.container(Class) variant of the container method creates new instances by attempting to invoke the constructor of the class that takes a single string argument, which is the desired name of the object. See the above link for project.container method variants that allow custom instantiation strategies.

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/)