Gradle Kotlin DSL Primer

Gradle's Kotlin DSL offers an alternative to the traditional Groovy DSL, delivering an enhanced editing experience in supported IDEs with features like better content assist, refactoring, and documentation.

This chapter explores the key Kotlin DSL constructs and demonstrates how to use them to interact with the Gradle API.

TIP

If you are interested in migrating an existing Gradle build to the Kotlin DSL, please also check out the dedicated migration page.

Prerequisites

- The embedded Kotlin compiler works on Linux, macOS, Windows, Cygwin, FreeBSD, and Solaris on x86-64 architectures.
- Familiarity with Kotlin syntax and basic language features is recommended. Refer to the Kotlin documentation and Kotlin Koans to learn the basics.
- Using the `plugins {}` block to declare Gradle plugins is highly recommended as it significantly improves the editing experience.

IDE support

The Kotlin DSL is fully supported by IntelliJ IDEA and Android Studio. While other IDEs lack advanced tools for editing Kotlin DSL files, you can still import Kotlin-DSL-based builds and work with them as usual.

	Build import	Syntax highlighting [1]	Semantic editor [2]
IntelliJ IDEA	✓	✓	✓
Android Studio	✓	✓	✓
Eclipse IDE	✓	✓	✖

CLion	✓	✓	✖
Apache NetBeans	✓	✓	✖
Visual Studio Code (LSP)	✓	✓	✖
Visual Studio	✓	✖	✖

1 Kotlin syntax highlighting in Gradle Kotlin DSL scripts

2 Code completion, navigation to sources, documentation, refactorings etc… in Gradle Kotlin DSL scripts

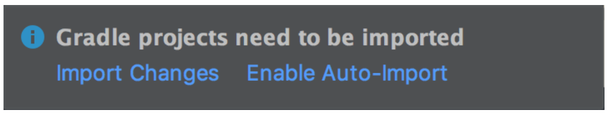
As noted in the limitations, you must import your project using the Gradle model to enable content assist and refactoring tools for Kotlin DSL scripts in IntelliJ IDEA.

Builds with slow configuration time might affect the IDE responsiveness, so please check out the performance section to help resolve such issues.

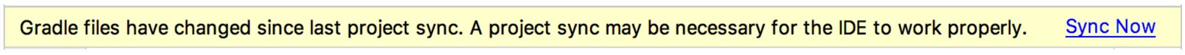
Automatic build import vs. automatic reloading of script dependencies

Both IntelliJ IDEA and Android Studio will detect when you make changes to your build logic and offer two suggestions:

1. Import the whole build again:

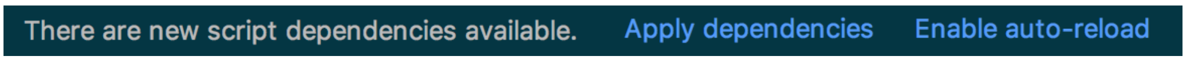


Gradle projects need to be imported



Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

2. Reload script dependencies when editing a build script:



There are new script dependencies available.

We recommend *disabling automatic build import* while *enabling automatic reloading of script dependencies*. This approach provides early feedback when editing Gradle scripts while giving you control over when the entire build setup synchronizes with your IDE.

See the Troubleshooting section to learn more.

Kotlin DSL scripts

Just like its Groovy-based counterpart, the Kotlin DSL is built on Gradle's Java API. Everything in a Kotlin DSL script is Kotlin code, compiled and executed by Gradle. Many of the objects, functions, and properties in your build scripts come from the Gradle API and the APIs of applied plugins.

TIP

Use the `Kotlin DSL reference` search to explore available members.

Script file names

- Groovy DSL script files use the `.gradle` file name extension.
- Kotlin DSL script files use the `.gradle.kts` file name extension.

To activate the Kotlin DSL, use the `.gradle.kts` extension for your build scripts instead of `.gradle`. This also applies to the settings file (e.g., `settings.gradle.kts`) and initialization scripts.

You can mix Groovy DSL and Kotlin DSL scripts within the same build. For example, a Kotlin DSL build script can apply a Groovy DSL one, and different projects in a multi-project build can use either.

To improve IDE support, we recommend following these conventions:

- Name settings scripts (or any script backed by a Gradle `Settings` object) using the pattern `*.settings.gradle.kts`. This includes script plugins applied from settings scripts.
- Name initialization scripts using the pattern `*.init.gradle.kts` or simply `init.gradle.kts`.

This helps the IDE identify the object "backing" the script, whether it's `Project`, `Settings`, or `Gradle`.

Implicit imports

All Kotlin DSL build scripts come with implicit imports, including:

- The default Gradle API imports
- The Kotlin DSL API, which includes types from the following packages:
  - `org.gradle.kotlin.dsl`
  - `org.gradle.kotlin.dsl.plugins.dsl`

- `org.gradle.kotlin.dsl.precompile`
- `java.util.concurrent.Callable`
- `java.util.concurrent.TimeUnit`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.io.File`
- `javax.inject.Inject`

Avoid Using Internal Kotlin DSL APIs

Using internal Kotlin DSL APIs in plugins and build scripts can break builds when either Gradle or plugins are updated.

The `Kotlin DSL API` extends the public Gradle API with types listed in the `corresponding API docs` found in the packages above (but not in their subpackages).

Compilation warnings

Gradle Kotlin DSL scripts are compiled by Gradle during the configuration phase of your build.

Deprecation warnings found by the Kotlin compiler are reported on the console when compiling the scripts:

```text
> Configure project :
w: build.gradle.kts:4:5: 'getter for uploadTaskName: String!' is deprecated. Deprecated in
Java
```

It is possible to configure your build to fail on any warning emitted during script compilation by setting the `org.gradle.kotlin.dsl.allWarningsAsErrors` Gradle property to `true`:

gradle.properties

```properties
org.gradle.kotlin.dsl.allWarningsAsErrors=true
```

Type-safe model accessors

The Groovy DSL allows you to reference many build model elements by name, even if they are defined at runtime, such as named configurations or source sets.

For example, when the `Java` plugin is applied, you can access the `implementation` configuration via `configurations.implementation`.

The Kotlin DSL replaces this dynamic resolution with type-safe model accessors, which work with model elements contributed by plugins.

Understanding when type-safe model accessors are available

The Kotlin DSL currently provides various sets of type-safe model accessors, each tailored to different scopes.

For the main project build scripts and precompiled project script plugins:

Type-safe model accessors	Example
Dependency and artifact configurations	`implementation` and `runtimeOnly` (contributed by the Java Plugin)
Project extensions and conventions, and extensions on them	`sourceSets`
Extensions on the `dependencies` and `repositories` containers, and extensions on them	`testImplementation` (contributed by the Java Plugin), `mavenCentral`
Elements in the `tasks` and `configurations` containers	`compileJava` (contributed by the Java Plugin), `test`
Elements in project-extension containers	Source sets contributed by the Java Plugin that are added to the `sourceSets` container: `sourceSets.main.java { setSrcDirs(listOf("src/main/java")) }`

For the main project settings script and precompiled settings script plugins:

Type-safe model accessors	Example
Project extensions and conventions, contributed by `Settings` plugins, and extensions on them	`pluginManagement`, `dependencyResolutionManagement`

IMPORTANT

Initialization scripts and script plugins do not have type-safe model accessors. These limitations will be removed in a future Gradle release.

The set of type-safe model accessors available is determined right before evaluating the script body, immediately after the `plugins {}` block. Model elements contributed after that point, such as configurations defined in your build script, **will not work** with type-safe model accessors:

build.gradle.kts

```kotlin
// Applies the Java plugin
plugins {
    id("java")
}

repositories {
    mavenCentral()
}

// Access to 'implementation' (contributed by the Java plugin) works here:
dependencies {
    implementation("org.apache.commons:commons-lang3:3.12.0")
    testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher") // Add this if needed for
runtime
}

// Add a custom configuration
configurations.create("customConfiguration")
// Type-safe accessors for 'customConfiguration' will NOT be available because it was
created after the plugins block
dependencies {
    customConfiguration("com.google.guava:guava:32.1.2-jre") // ❌ Error: No type-safe
accessor for 'customConfiguration'
}
```

However, this means you can use type-safe accessors for any model elements contributed by plugins that are *applied by parent projects*.

The following project build script demonstrates how you can access various configurations, extensions and other elements using type-safe accessors:

build.gradle.kts

```kotlin
plugins {
    `java-library`
}

dependencies {                                  ❶
    api("junit:junit:4.13")
    implementation("junit:junit:4.13")
    testImplementation("junit:junit:4.13")
}

configurations {                                ❶
    implementation {
        resolutionStrategy.failOnVersionConflict()
    }
}

sourceSets {                                    ❷
    main {                                      ❸
        java.srcDir("src/core/java")
    }
}

java {                                          ❹
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    test {                                      ❺
        testLogging.showExceptions = true
        useJUnit()
    }
}
```

❶ Uses type-safe accessors for the `api`, `implementation` and `testImplementation` dependency configurations contributed by the Java Library Plugin

❷ Uses an accessor to configure the `sourceSets` project extension

❸ Uses an accessor to configure the `main` source set

❹ Uses an accessor to configure the `java` source for the `main` source set

❺ Uses an accessor to configure the `test` task

TIP

Your IDE is aware of the type-safe accessors and will include them in its suggestions.

This applies both at the top level of your build scripts, where most plugin extensions are added to the `Project` object, and within the blocks that configure an extension.

Note that accessors for elements of containers such as `configurations`, `tasks`, and `sourceSets` leverage Gradle's configuration avoidance APIs. For example, on `tasks`, accessors are of type `TaskProvider<T>` and provide a lazy reference and lazy configuration of the underlying task.

Here are some examples illustrating when configuration avoidance applies:

build.gradle.kts

KOTLIN

```kotlin
tasks.test {
    // lazy configuration
    useJUnitPlatform()
}

// Lazy reference
val testProvider: TaskProvider<Test> = tasks.test

testProvider {
    // lazy configuration
}

// Eagerly realized Test task, defeats configuration avoidance if done out of a lazy context
val test: Test = tasks.test.get()
```

For all other containers, accessors for elements are of type `NamedDomainObjectProvider<T>`, providing the same behavior:

build.gradle.kts

KOTLIN

```kotlin
val mainSourceSetProvider: NamedDomainObjectProvider<SourceSet> = sourceSets.named("main")
```

Understanding what to do when type-safe model accessors are not available

Consider the sample build script shown above, which demonstrates the use of type-safe accessors. The following sample is identical, except it uses the `apply()` method to apply the plugin.

In this case, the build script cannot use type-safe accessors because the `apply()` call occurs in the body of the build script. You must use another techniques instead, as demonstrated here:

build.gradle.kts

KOTLIN

```kotlin
apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.13")
    "implementation"("junit:junit:4.13")
    "testImplementation"("junit:junit:4.13")
}

configurations {
    "implementation" {
        resolutionStrategy.failOnVersionConflict()
    }
}

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}

configure<JavaPluginExtension> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}
```

Type-safe accessors are unavailable for model elements contributed by the following:

- Plugins applied via the `apply(plugin = "id")` method.
- The project build script.
- Script plugins, via `apply(from = "script-plugin.gradle.kts")`.
- Plugins applied via cross-project configuration.

You cannot use type-safe accessors in binary Gradle plugins implemented in Kotlin.

If you can’t find a type-safe accessor, *fall back to using the normal API* for the corresponding types. To do so, you need to know the names and/or types of the configured model elements. We will now show you how these can be discovered by examining the script in detail.

Artifact configurations

The following sample demonstrates how to reference and configure artifact configurations without type-safe accessors:

build.gradle.kts

KOTLIN

```kotlin
apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.13")
    "implementation"("junit:junit:4.13")
    "testImplementation"("junit:junit:4.13")
}

configurations {
    "implementation" {
        resolutionStrategy.failOnVersionConflict()
    }
}
```

The code looks similar to that of the type-safe accessors, except that the configuration names are string literals. You can use string literals for configuration names in dependency declarations and within the `configurations {}` block.

While the IDE won't be able to help you discover the available configurations, you can look them up either in the corresponding plugin's documentation or by running `./gradlew dependencies`.

Project extensions

Project extensions have both a name and a unique type. However, the Kotlin DSL only needs to know the type to configure them.

The following sample shows the `sourceSets {}` and `java {}` blocks from the original example build script. The `configure<T>()` function is used with the corresponding type:

build.gradle.kts

KOTLIN

```kotlin
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}

configure<JavaPluginExtension> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}
```

Note that `sourceSets` is a Gradle extension on `Project` of type `SourceSetContainer` and `java` is an extension on `Project` of type `JavaPluginExtension`.

You can discover available extensions by either reviewing the documentation for the applied plugins or running `./gradlew kotlinDslAccessorsReport`. The report generates the Kotlin code needed to access the model elements contributed by the applied plugins, providing both names and types.

As a last resort, you can check the plugin's source code, though this should not be necessary in most cases.

You can also use the `the<T>()` function if you only need a reference to the extension without configuring it, or if you want to perform a one-line configuration:

build.gradle.kts

KOTLIN

```
the<SourceSetContainer>()["main"].java.srcDir("src/main/java")
```

The snippet above also demonstrates one way to configure elements of a project extension that is a container.

Elements in project-extension containers

Container-based project extensions, such as `SourceSetContainer`, allow you to configure the elements they hold.

In our sample build script, we want to configure a source set named `main` within the source set container. We can do this by using the `named()` method instead of an accessor:

build.gradle.kts

KOTLIN

```
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}
```

All elements within a container-based project extension have a name, so you can use this technique in all such cases.

For project extensions and conventions, you can discover what elements are present in any container by either checking the documentation for the applied plugins or by running `./gradlew kotlinDslAccessorsReport`.

As a last resort, you may also review the plugin's source code to find out what it does.

Tasks

Tasks are not managed through a container-based project extension, but they are part of a container that behaves in a similar way.

This means that you can configure tasks in the same way as you do for source sets. The following example illustrates this approach:

build.gradle.kts

KOTLIN

```kotlin
apply(plugin = "java-library")

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}
```

We are using the Gradle API to refer to tasks by name and type, rather than using accessors.

Note that it is necessary to specify the type of the task explicitly. If you don't, the script won't compile because the inferred type will be `Task`, not `Test`, and the `testLogging` property is specific to the `Test` task type.

However, you can omit the type if you only need to configure properties or call methods that are common to all tasks, i.e., those declared on the `Task` interface.

You can discover what tasks are available by running `./gradlew tasks`.

To find out the type of a given task, run `./gradlew help --task <taskName>`, as demonstrated here:

TEXT

```
❯ ./gradlew help --task test
...
```

```
Type
    Test (org.gradle.api.tasks.testing.Test)
```

The IDE can assist you with the required imports, so you only need the simple names of the types, without the package name part. In this case, there's no need to import the `Test` task type, as it is part of the Gradle API and is therefore imported implicitly.

Working with container objects

The Gradle build model makes extensive use of container objects (or simply "containers").

For example, `configurations` and `tasks` are containers that hold `Configuration` and `Task` objects, respectively. Community plugins also contribute containers, such as the `android.buildTypes` container contributed by the Android Plugin.

The Kotlin DSL provides multiple ways for build authors to interact with containers. We will explore each of these methods, using the `tasks` container as an example.

TIP

You can leverage the type-safe accessors described in another section when configuring existing elements on supported containers. That section also explains which containers support type-safe accessors.

Using the container API

All containers in Gradle implement `NamedDomainObjectContainer<DomainObjectType>`. Some containers can hold objects of different types and implement `PolymorphicDomainObjectContainer<BaseType>`. The simplest way to interact with containers is through these interfaces.

The following example demonstrates how you can use the `named()` method to configure existing tasks, and the `register()` method to create new tasks:

build.gradle.kts

KOTLIN

```kotlin
tasks.named("check")                    1
tasks.register("myTask1")               2

tasks.named<JavaCompile>("compileJava") 3
tasks.register<Copy>("myCopy1")         4

tasks.named("assemble") {               5
    dependsOn(":myTask1")
}
tasks.register("myTask2") {             6
```

```
    description = "Some meaningful words"
}

tasks.named<Test>("test") {            7
    testLogging.showStackTraces = true
}
tasks.register<Copy>("myCopy2") {      8
    from("source")
    into("destination")
}
```

1. Gets a reference of type `Task` to the existing task named `check`
2. Registers a new untyped task named `myTask1`
3. Gets a reference to the existing task named `compileJava` of type `JavaCompile`
4. Registers a new task named `myCopy1` of type `Copy`
5. Gets a reference to the existing (untyped) task named `assemble` and configures it — you can only configure properties and methods that are available on `Task` with this syntax
6. Registers a new untyped task named `myTask2` and configures it — you can only configure properties and methods that are available on `Task` in this case
7. Gets a reference to the existing task named `test` of type `Test` and configures it — in this case you have access to the properties and methods of the specified type
8. Registers a new task named `myCopy2` of type `Copy` and configures it

NOTE

The above sample relies on the configuration avoidance APIs. If you need or want to eagerly configure or register container elements, simply replace `named()` with `getByName()` and `register()` with `create()`.

Using Kotlin delegated properties

Another way to interact with containers is via Kotlin delegated properties. These are particularly useful if you need a reference to a container element that you can use elsewhere in the build. Additionally, Kotlin delegated properties can easily be renamed via IDE refactoring.

The following example achieves the same result as the one in the previous section, but it uses delegated properties and reuses those references instead of string-literal task paths:

build.gradle.kts

KOTLIN

```kotlin
val check by tasks.existing
val myTask1 by tasks.registering

val compileJava by tasks.existing(JavaCompile::class)
val myCopy1 by tasks.registering(Copy::class)

val assemble by tasks.existing {
    dependsOn(myTask1)  1
}
val myTask2 by tasks.registering {
    description = "Some meaningful words"
}

val test by tasks.existing(Test::class) {
    testLogging.showStackTraces = true
}
val myCopy2 by tasks.registering(Copy::class) {
    from("source")
    into("destination")
}
```

1 Uses the reference to the `myTask1` task rather than a task path

NOTE

The above sample relies on the configuration avoidance APIs. If you need or want to eagerly configure or register container elements, simply replace `existing()` with `getting()` and `registering()` with `creating()`.

Configuring multiple container elements together

When configuring several elements of a container, you can group interactions in a block to avoid repeating the container's name on each interaction.

The following example demonstrates a combination of type-safe accessors, the container API, and Kotlin delegated properties:

build.gradle.kts

KOTLIN

```kotlin
tasks {
    test {
        testLogging.showStackTraces = true
    }
    val myCheck by registering {
        doLast { /* assert on something meaningful */ }
    }
    check {
        dependsOn(myCheck)
    }
    register("myHelp") {
        doLast { /* do something helpful */ }
    }
}
```

Working with runtime properties

Gradle has two main sources of properties defined at runtime: *project properties* and *extra properties*.

The Kotlin DSL provides specific syntax for working with these property types, which we will explore in the following sections.

Project properties

The Kotlin DSL allows you to access project properties by binding them via Kotlin delegated properties.

The following snippet demonstrates this technique for a couple of project properties, one of which *must* be defined:

build.gradle.kts

KOTLIN

```kotlin
val myProperty: String by project (1)
val myNullableProperty: String? by project (2)
```

1. Makes the `myProperty` project property available via a `myProperty` delegated property — the project property must exist in this case, otherwise the build will fail when the build script attempts to use the `myProperty` value
2. Does the same for the `myNullableProperty` project property, but the build won't fail on using the `myNullableProperty` value as long as you check for null (standard Kotlin rules for null safety apply)

The same approach works in both settings and initialization scripts, except you use `by settings` and `by gradle` respectively in place of `by project`.

Extra properties

Extra properties are available on any object that implements the `ExtensionAware` interface.

In Kotlin DSL, you can access and create extra properties via delegated properties, using the `by extra` syntax as demonstrated in the following sample:

build.gradle.kts

```kotlin
val myNewProperty by extra("initial value") ❶
val myOtherNewProperty by extra { "calculated initial value" } ❷

val myExtraProperty: String by extra ❸
val myExtraNullableProperty: String? by extra ❹
```

❶ Creates a new extra property called `myNewProperty` in the current context (the project in this case) and initializes it with the value `"initial value"`, which also determines the property's *type*

❷ Create a new extra property whose initial value is calculated by the provided lambda

❸ Binds an existing extra property from the current context (the project in this case) to a `myProperty` reference

❹ Does the same as the previous line but allows the property to have a null value

This approach works for all Gradle scripts: project build scripts, script plugins, settings scripts, and initialization scripts.

You can also access extra properties on a root project from a subproject using the following syntax:

my-sub-project/build.gradle.kts

```kotlin
val myNewProperty: String by rootProject.extra ❶
```

❶ Binds the root project's `myNewProperty` extra property to a reference of the same name

Extra properties aren't just limited to projects. For example, `Task` extends `ExtensionAware`, so you can attach extra properties to tasks as well.

Here's an example that defines a new `myNewTaskProperty` on the `test` task and then uses that property to initialize another task:

build.gradle.kts

```kotlin
tasks {
    test {
        val reportType by extra("dev") (1)
        doLast {
            // Use 'suffix' for post-processing of reports
        }
    }

    register<Zip>("archiveTestReports") {
        val reportType: String by test.get().extra (2)
        archiveAppendix = reportType
        from(test.get().reports.html.outputLocation)
    }
}
```

1. Creates a new `reportType` extra property on the `test` task
2. Makes the `test` task's `reportType` extra property available to configure the `archiveTestReports` task

If you're happy to use eager configuration rather than the configuration avoidance APIs, you could use a single, "global" property for the report type, like this:

build.gradle.kts

```kotlin
tasks.test {
    doLast { /* ... */ }
}

val testReportType by tasks.test.get().extra("dev") (1)

tasks.create<Zip>("archiveTestsReports") {
    archiveAppendix = testReportType (2)
    from(test.reports.html.outputLocation)
}
```

1. Creates and initializes an extra property on the `test` task, binding it to a "global" property
2. Uses the "global" property to initialize the `archiveTestReports` task

There is one last syntax for extra properties that treats `extra` as a map. We generally recommend against using this, as it bypasses Kotlin's type checking and limits IDE support. However, it is more succinct than the delegated properties syntax and can be used if you only need to set an extra property without referencing it later.

Here is a simple example demonstrating how to set and read extra properties using the map syntax:

build.gradle.kts

KOTLIN

```kotlin
extra["myNewProperty"] = "initial value" (1)

tasks.register("myTask") {
    doLast {
        println("Property: ${project.extra["myNewProperty"]}") (2)
    }
}
```

1. Creates a new project extra property called `myNewProperty` and sets its value
2. Reads the value from the project extra property we created — note the `project.` qualifier on `extra[…]`, otherwise Gradle will assume we want to read an extra property from the *task*

Working with Gradle types

`Property`, `Provider`, and `NamedDomainObjectProvider` are types that represent deferred and lazy evaluation of values and objects. The Kotlin DSL provides a specialized syntax for working with these types.

Using a `Property`

A property represents a value that can be set and read lazily:

- Setting a value: `property.set(value)` or `property = value`
- Accessing the value: `property.get()`
- Using the delegate syntax: `val propValue: String by property`

build.gradle.kts

KOTLIN

```kotlin
val myProperty: Property<String> = project.objects.property(String::class.java)

myProperty.set("Hello, Gradle!") // Set the value
println(myProperty.get())        // Access the value

// Using delegate syntax
val propValue: String by myProperty
println(propValue)

// Using lazy syntax
```

```
myProperty = "Hi, Gradle!" // Set the value
println(myProperty.get())  // Access the value
```

Using a `Provider`

A provider represents a read-only, lazily-evaluated value:

- Accessing the value: `provider.get()`
- Chaining: `provider.map { transform(it) }`

build.gradle.kts

```kotlin
val versionProvider: Provider<String> = project.provider { "1.0.0" }

println(versionProvider.get()) // Access the value

// Chaining transformations
val majorVersion: Provider<String> = versionProvider.map { it.split(".")[0] }
println(majorVersion.get()) // Prints: "1"
```

Using a `NamedDomainObjectProvider`

A named domain object provider represents a lazily-evaluated named object from a Gradle container (like tasks or extensions):

- Accessing the object: `namedObjectProvider.get()`
- Configuring the object: `namedObjectProvider.configure { … }`

build.gradle.kts

```kotlin
val myTaskProvider: NamedDomainObjectProvider<Task> = tasks.named("build")

// Configuring the task
myTaskProvider.configure {
    doLast {
        println("Build task completed!")
    }
}

// Accessing the task
val myTask: Task = myTaskProvider.get()
```

Lazy property assignment

Gradle's Kotlin DSL supports lazy property assignment using the `=` operator.

Lazy property assignment reduces verbosity when lazy properties are used. It works for properties that are publicly seen as `final` (without a setter) and have type `Property` or `ConfigurableFileCollection`. Since properties must be `final`, we generally recommend avoiding custom setters for properties with lazy types and, if possible, implementing such properties via an abstract getter.

Using the `=` operator is the preferred way to call `set()` in the Kotlin DSL:

build.gradle.kts

KOTLIN

```kotlin
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

abstract class WriteJavaVersionTask : DefaultTask() {
    @get:Input
    abstract val javaVersion: Property<String>
    @get:OutputFile
    abstract val output: RegularFileProperty

    @TaskAction
    fun execute() {
        output.get().asFile.writeText("Java version: ${javaVersion.get()}")
    }
}

tasks.register<WriteJavaVersionTask>("writeJavaVersion") {
    javaVersion.set("17") (1)
    javaVersion = "17" (2)
    javaVersion = java.toolchain.languageVersion.map { it.toString() } (3)
    output = layout.buildDirectory.file("writeJavaVersion/javaVersion.txt")
}
```

1. Set value with the `.set()` method
2. Set value with lazy property assignment using the `=` operator
3. The `=` operator can be used also for assigning lazy values

IDE support

Lazy property assignment is supported from IntelliJ 2022.3 and from Android Studio Giraffe.

Kotlin DSL Plugin

The Kotlin DSL Plugin provides a convenient way to develop Kotlin-based projects that contribute build logic. This includes buildSrc projects, included builds, and Gradle plugins.

The plugin achieves this by doing the following:

- Applies the Kotlin Plugin, which adds support for compiling Kotlin source files.
- Adds the `kotlin-stdlib`, `kotlin-reflect`, and `gradleKotlinDsl()` dependencies to the `compileOnly` and `testImplementation` configurations, enabling the use of those Kotlin libraries and the Gradle API in your Kotlin code.
- Configures the Kotlin compiler with the same settings used for Kotlin DSL scripts, ensuring consistency between your build logic and those scripts:
    - Adds Kotlin compiler arguments,
    - Registers the SAM-with-receiver Kotlin compiler plugin.
- Enables support for precompiled script plugins.

Each Gradle release is meant to be used with a specific version of the `kotlin-dsl` plugin. Compatibility between arbitrary Gradle releases and `kotlin-dsl` plugin versions is not guaranteed. Using an unexpected version of the `kotlin-dsl` plugin will emit a warning and can cause hard-to-diagnose problems.

This is the basic configuration you need to use the plugin:

buildSrc/build.gradle.kts

KOTLIN

```kotlin
plugins {
    `kotlin-dsl`
}

repositories {
    // The org.jetbrains.kotlin.jvm plugin requires a repository
    // where to download the Kotlin compiler dependencies from.
    mavenCentral()
}
```

The Kotlin DSL Plugin leverages Java Toolchains. By default, the code will target Java 8. You can change that by defining a Java toolchain to be used by the project:

buildSrc/src/main/kotlin/myproject.java-conventions.gradle.kts

KOTLIN

```kotlin
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
    }
}
```

Embedded Kotlin

Gradle embeds Kotlin in order to provide support for Kotlin-based scripts.

Kotlin versions

Gradle ships with `kotlin-compiler-embeddable` plus matching versions of `kotlin-stdlib` and `kotlin-reflect` libraries. For details, see the Kotlin section of Gradle's compatibility matrix. The `kotlin` package from those modules is visible through the Gradle classpath.

The compatibility guarantees provided by Kotlin apply for both backward and forward compatibility.

Backward compatibility

Our approach is to only make backward-incompatible Kotlin upgrades with major Gradle releases. We clearly document the Kotlin version shipped with each release and announce upgrade plans ahead of major releases.

Plugin authors aiming to maintain compatibility with older Gradle versions must limit their API usage to what is supported by those versions. This is no different from working with any new API in Gradle. For example, if a new API for dependency resolution is introduced, a plugin must either drop support for older Gradle versions or organize its code to conditionally execute the new code path on compatible versions.

Forward compatibility

The primary compatibility concern lies between the external `kotlin-gradle-plugin` version and the `kotlin-stdlib` version shipped with Gradle. More broadly, this applies to any plugin that transitively depends on `kotlin-stdlib` and its version provided by Gradle. As long as the versions are compatible, everything should work as expected. This issue will diminish as the Kotlin language matures.

Kotlin compiler arguments

The following Kotlin compiler arguments are used for compiling Kotlin DSL scripts, as well as Kotlin sources and scripts in projects with the `kotlin-dsl` plugin applied:

`-java-parameters`

Generate metadata for Java >= 1.8 reflection on method parameters. See Kotlin/JVM compiler options in the Kotlin documentation for more information.

`-Xjvm-default=all`

Makes all non-abstract members of Kotlin interfaces default for the Java classes implementing them. This is to provide a better interoperability with Java and Groovy for plugins written in Kotlin. See Default methods in interfaces in the Kotlin documentation for more information.

`-Xsam-conversions=class`

Sets up the implementation strategy for SAM (single abstract method) conversion to always generate anonymous classes, instead of using the `invokedynamic` JVM instruction. This is to provide a better support for configuration cache and incremental build. See KT-44912 in the Kotlin issue tracker for more information.

`-Xjsr305=strict` & `-Xjspecify-annotations=strict`

Sets up Kotlin's Java interoperability to strictly follow JSR-305 and JSpecify annotations for increased null safety. See Calling Java code from Kotlin in the Kotlin documentation for more information.

Interoperability

When mixing languages in your build logic, you may have to cross language boundaries. An extreme example would be a build that uses tasks and plugins that are implemented in Java, Groovy and Kotlin, while also using both Kotlin DSL and Groovy DSL build scripts.

> *Kotlin is designed with Java Interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well.*
>
> — *Kotlin reference documentation*

Both calling Java from Kotlin and calling Kotlin from Java are very well covered in the Kotlin reference documentation.

The same mostly applies to interoperability with Groovy code. In addition, the Kotlin DSL provides several ways to opt into Groovy semantics, which we look at next.

Static extensions

Both the Groovy and Kotlin languages support extending existing classes via Groovy Extension modules and Kotlin extensions.

To call a Kotlin extension function from Groovy, call it as a static function, passing the receiver as the first parameter:

build.gradle

GROOVY

```groovy
TheTargetTypeKt.kotlinExtensionFunction(receiver, "parameters", 42, aReference)
```

Kotlin extension functions are package-level functions. You can learn how to locate the name of the type declaring a given Kotlin extension in the Package-Level Functions section of the Kotlin reference documentation.

To call a Groovy extension method from Kotlin, the same approach applies: call it as a static function passing the receiver as the first parameter:

build.gradle.kts

KOTLIN

```kotlin
TheTargetTypeGroovyExtension.groovyExtensionMethod(receiver, "parameters", 42, aReference)
```

Named parameters and default arguments

Both the Groovy and Kotlin languages support named function parameters and default arguments, although they are implemented very differently. Kotlin has fully-fledged support for both, as described in the Kotlin language reference under named arguments and default arguments. Groovy implements named arguments in a non-type-safe way based on a `Map<String, ?>` parameter, which means they cannot be combined with default arguments. In other words, you can only use one or the other in Groovy for any given method.

Calling Kotlin from Groovy

To call a Kotlin function that has named arguments from Groovy, just use a normal method call with positional parameters:

build.gradle

GROOVY

```groovy
kotlinFunction("value1", "value2", 42)
```

There is no way to provide values by argument name.

To call a Kotlin function that has default arguments from Groovy, always pass values for all the function parameters.

Calling Groovy from Kotlin

To call a Groovy function with named arguments from Kotlin, you need to pass a `Map<String, ?>`, as shown in this example:

build.gradle.kts

```kotlin
groovyNamedArgumentTakingMethod(mapOf(
    "parameterName" to "value",
    "other" to 42,
    "and" to aReference))
```

To call a Groovy function with default arguments from Kotlin, always pass values for all the parameters.

Groovy closures from Kotlin

You may sometimes have to call Groovy methods that take Closure arguments from Kotlin code. For example, some third-party plugins written in Groovy expect closure arguments.

NOTE

Gradle plugins written in any language should prefer the type `Action<T>` type in place of closures. Groovy closures and Kotlin lambdas are automatically mapped to arguments of that type.

In order to provide a way to construct closures while preserving Kotlin's strong typing, two helper methods exist:

- `closureOf<T> {}`
- `delegateClosureOf<T> {}`

Both methods are useful in different circumstances and depend upon the method you are passing the `Closure` instance into.

Some plugins expect simple closures, as with the Bintray plugin:

build.gradle.kts

```kotlin
bintray {
    pkg(closureOf<PackageConfig> {
        // Config for the package here
    })
}
```

In other cases, like with the Gretty Plugin when configuring farms, the plugin expects a delegate closure:

build.gradle.kts

KOTLIN

```kotlin
farms {
    farm("OldCoreWar", delegateClosureOf<FarmExtension> {
        // Config for the war here
    })
}
```

There sometimes isn't a good way to tell, from looking at the source code, which version to use. Usually, if you get a `NullPointerException` with `closureOf<T> {}`, using `delegateClosureOf<T> {}` will resolve the problem.

These two utility functions are useful for *configuration closures*, but some plugins might expect Groovy closures for other purposes. The `KotlinClosure0` to `KotlinClosure2` types allows adapting Kotlin functions to Groovy closures with more flexibility:

build.gradle.kts

KOTLIN

```kotlin
somePlugin {

    // Adapt parameter-less function
    takingParameterLessClosure(KotlinClosure0({
        "result"
    }))

    // Adapt unary function
    takingUnaryClosure(KotlinClosure1<String, String>({
        "result from single parameter $this"
    }))

    // Adapt binary function
    takingBinaryClosure(KotlinClosure2<String, String, String>({ a, b ->
        "result from parameters $a and $b"
    }))
}
```

The Kotlin DSL Groovy Builder

If some plugin makes heavy use of Groovy metaprogramming, then using it from Kotlin or Java or any statically-compiled language can be very cumbersome.

The Kotlin DSL provides a `withGroovyBuilder {}` utility extension that attaches the Groovy metaprogramming semantics to objects of type `Any`.

The following example demonstrates several features of the method on the object `target`:

build.gradle.kts

KOTLIN

```kotlin
target.withGroovyBuilder {                                                   (1)

    // GroovyObject methods available                                        (2)
    if (hasProperty("foo")) { /*...*/ }
    val foo = getProperty("foo")
    setProperty("foo", "bar")
    invokeMethod("name", arrayOf("parameters", 42, aReference))

    // Kotlin DSL utilities
    "name"("parameters", 42, aReference)                                     (3)
        "blockName" {                                                        (4)
            // Same Groovy Builder semantics on `blockName`
        }
    "another"("name" to "example", "url" to "https://example.com/")         (5)
}
```

1. The receiver is a GroovyObject and provides Kotlin helpers
2. The `GroovyObject` API is available
3. Invoke the `methodName` method, passing some parameters
4. Configure the `blockName` property, maps to a `Closure` taking method invocation
5. Invoke `another` method taking named arguments, maps to a Groovy named arguments `Map<String, ?>` taking method invocation

Using a Groovy script

Another option when dealing with problematic plugins that assume a Groovy DSL build script is to configure them in a Groovy DSL build script that is applied from the main Kotlin DSL build script:

dynamic-groovy-plugin-configuration.gradle

GROOVY

```groovy
native {    (1)
    dynamic {
        groovy as Usual
    }
}
```

build.gradle.kts

KOTLIN

```kotlin
plugins {
    id("dynamic-groovy-plugin") version "1.0"    ②
}
apply(from = "dynamic-groovy-plugin-configuration.gradle")  ③
```

① The Groovy script uses dynamic Groovy to configure plugin

② The Kotlin build script requests and applies the plugin

③ The Kotlin build script applies the Groovy script

Troubleshooting

The IDE support is provided by two components:

1. Kotlin Plugin (used by IntelliJ IDEA/Android Studio).
2. Gradle.

The level of support varies based on the versions of each.

If you encounter issues, first run `./gradlew tasks` from the command line to determine if the problem is specific to the IDE. If the issue persists on the command line, it likely originates from the build itself rather than IDE integration.

However, if the build runs successfully on the command line but your script editor reports errors, try restarting your IDE and invalidating its caches.

If the issue persists, and you suspect a problem with the Kotlin DSL script editor, try the following:

- Run `./gradlew tasks` to gather more details.
- Check the logs in one of these locations:
    - `$HOME/Library/Logs/gradle-kotlin-dsl` on macOS
    - `$HOME/.gradle-kotlin-dsl/log` on Linux
    - `$HOME/AppData/Local/gradle-kotlin-dsl/log` on Windows
- Report the issue on the Gradle issue tracker, including as much detail as possible.

From version 5.1 onward, the log directory is automatically cleaned. Logs are checked periodically (at most, every 24 hours), and files are deleted if unused for 7 days.

If this doesn’t help pinpoint the problem, you can enable the `org.gradle.kotlin.dsl.logging.tapi` system property in your IDE. This causes the Gradle Daemon to log additional details in its log file located at `$HOME/.gradle/daemon`.

In IntelliJ IDEA, enable this property by navigating to `Help > Edit Custom VM Options…` and adding: `-Dorg.gradle.kotlin.dsl.logging.tapi=true`.

For IDE problems outside the Kotlin DSL script editor, please open issues in the corresponding IDE’s issue tracker:

- JetBrains’s IDEA issue tracker
- Google’s Android Studio issue tracker

Lastly, if you face problems with Gradle itself or with the Kotlin DSL, please open issues on the Gradle issue tracker.

Limitations

- The Kotlin DSL is known to be slower than the Groovy DSL on first use, for example with clean checkouts or on ephemeral continuous integration agents. Changing something in the *buildSrc* directory also has an impact as it invalidates build-script caching. The main reason for this is the slower script compilation for Kotlin DSL.
- In IntelliJ IDEA, you must import your project from the Gradle model in order to get content assist and refactoring support for your Kotlin DSL build scripts.
- Kotlin DSL script compilation avoidance has known issues. If you encounter problems, it can be disabled by setting the `org.gradle.kotlin.dsl.scriptCompilationAvoidance` system property to `false`.
- The Kotlin DSL will not support the `model {}` block, which is part of the discontinued Gradle Software Model.

If you run into trouble or discover a suspected bug, please report the issue in the Gradle issue tracker.