


[Multiplatform development](#) / [Sharing code principles](#) / Hierarchical project structure

# Hierarchical project structure

 [Edit page](#) Last modified: 02 July 2023

Multiplatform projects support hierarchical structures. This means you can arrange a hierarchy of intermediate source sets for sharing the common code among some, but not all, [supported targets](#). Using intermediate source sets has some important advantages:

- If you're a library author and you want to provide a specialized API, you can use an intermediate source set for some, but not all, targets – for example, an intermediate source set for Kotlin/Native targets but not for Kotlin/JVM ones.
- If you want to use platform-dependent libraries in your project, you can use an intermediate source set to use that specific API in several native targets. For example, you can have access to iOS-specific dependencies, such as Foundation, when sharing code across all iOS targets.
- Some libraries aren't available for particular platforms. Specifically, native libraries are only available for source sets that compile to Kotlin/Native. Using an intermediate source set will solve this issue.

The Kotlin toolchain ensures that each source set has access only to the API that is available for all targets to which that source set compiles. This prevents cases like using a Windows-specific API and then compiling it to macOS, resulting in linkage errors or undefined behavior at runtime.

There are 3 ways to create a target hierarchy:

- [Specify all targets and enable the default hierarchy](#)

- [Use target shortcuts available for typical cases](#)
- [Manually declare and connect the source sets](#)

## Default hierarchy



The default target hierarchy is [Experimental](#). It may be changed in future Kotlin releases without prior notice. For Kotlin Gradle build scripts, opting in is required with `@OptIn(ExperimentalKotlinGradlePluginApi::class)`.

Starting with Kotlin 1.8.20, you can set up a source set hierarchy in your multiplatform projects with the default target hierarchy. It's a [template](#) for all possible targets and their shared source sets hardcoded in the Kotlin Gradle plugin.

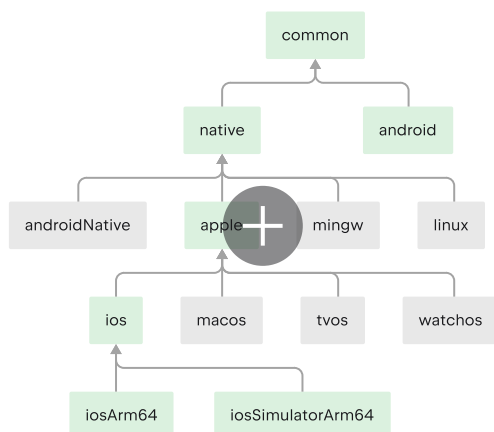
### Set up your project

To set up a hierarchy, call `targetHierarchy.default()` in the `kotlin` block of your `build.gradle(.kts)` file and list all of the targets you need. For example:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
    targetHierarchy.default()

    android()
    iosArm64()
    iosSimulatorArm64()
}
```

When you declare the final targets `android`, `iosArm64`, and `iosSimulatorArm64` in your code, the Kotlin Gradle plugin finds suitable shared source sets from the template and creates them for you. The resulting hierarchy looks like this:



Green source sets are actually created and present in the project, while gray ones from the default template are ignored. The Kotlin Gradle plugin hasn't created the `watchos` source set, for example, because there are no watchOS targets in the project.

If you add a watchOS target, like `watchosArm64`, the `watchos` source set is created, and the code from the `apple`, `native`, and `common` source sets is compiled to `watchosArm64` as well.

**i** In this example, the `apple` and `native` source sets compile only to the `iosArm64` and `iosSimulatorArm64` targets. Despite their names, they have access to the full iOS API. This can be counter-intuitive for source sets like `native`, as you might expect that only APIs available on all native targets are accessible in this source set. This behavior may change in the future.

## Adjust the resulting hierarchy

You can further configure the resulting hierarchy manually using the `dependsOn` relation. To do so, apply the `by getting` construction for the source sets created with `targetHierarchy.default()`.

Consider this example of a project with a source set shared between the `jvm` and `native` targets only:

```
@OptIn(ExperimentalKotlinGradlePluginApi::class)
kotlin {
    // Enable the default target hierarchy:
```

```
targetHierarchy.default()

jvm()
iosArm64()
// the rest of the necessary targets...

sourceSets {
    val commonMain by getting

    val jvmAndNativeMain by creating {
        dependsOn(commonMain)
    }

    val nativeMain by getting {
        dependsOn(jvmAndNativeMain)
    }

    val jvmMain by getting {
        dependsOn(jvmAndNativeMain)
    }
}
```

It can be cumbersome to remove `dependsOn` relations that are automatically created by the `targetHierarchy.default()` call. In that case, use an entirely manual configuration instead of calling the default hierarchy.



We're currently working on an API to create your own target hierarchies. It will be useful for projects whose hierarchy configurations are significantly different from the default template.

This API is not ready yet, but if you're eager to try it, look into the `targetHierarchy.custom { ... }` block and the declaration of `targetHierarchy.default()` as an example. Keep in mind that this API is still in development. It might not be tested, and can change in further releases.

**+ See the full hierarchy template**

# Target shortcuts

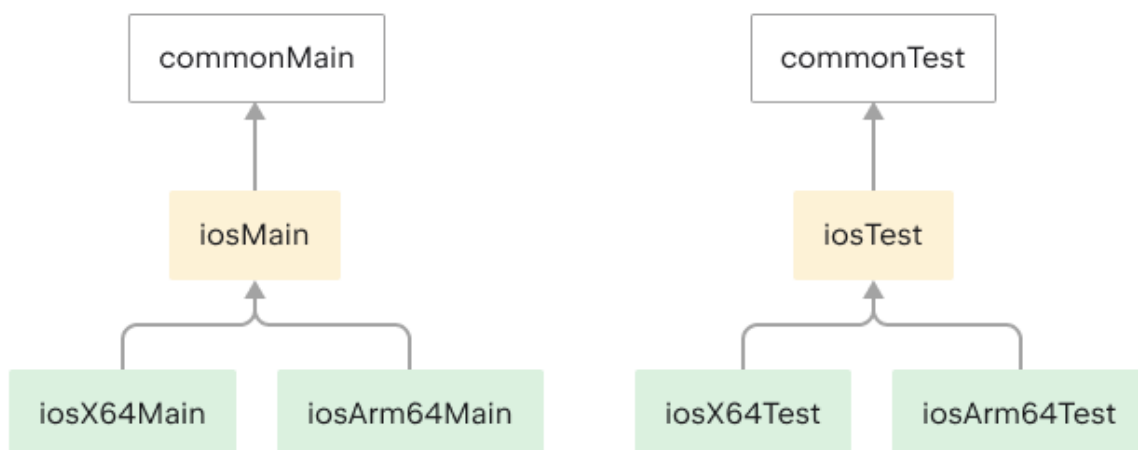
The Kotlin Multiplatform plugin provides some predefined target shortcuts for creating structures for common target combinations:

Target shortcut	Targets
<code>ios</code>	<code>iosArm64</code> , <code>iosX64</code>
<code>watchos</code>	<code>watchosArm32</code> , <code>watchosArm64</code> , <code>watchosX64</code>
<code>tvos</code>	<code>tvosArm64</code> , <code>tvosX64</code>

All shortcuts create similar hierarchical structures in the code. For example, you can use the `ios()` shortcut to create a multiplatform project with 2 iOS-related targets, `iosArm64` and `iosX64` , and a shared source set:

```
kotlin {  
    ios() // iOS device and simulator targets; iosMain and iosTest so  
}
```

In this case, the hierarchical structure includes the intermediate source sets `iosMain` and `iosTest` , which are used by the platform-specific source sets:



The resulting hierarchical structure will be equivalent to the code below:

Kotlin

Groovy

```
kotlin {  
    iosX64()  
    iosArm64()  
  
    sourceSets {  
        val commonMain by getting  
        val iosX64Main by getting  
        val iosArm64Main by getting  
        val iosMain by creating {  
            dependsOn(commonMain)  
            iosX64Main.dependsOn(this)  
            iosArm64Main.dependsOn(this)  
        }  
    }  
}
```

## Target shortcuts and ARM64 (Apple Silicon) simulators

The `ios`, `watchos`, and `tvos` target shortcuts don't include the simulator targets for ARM64 (Apple Silicon) platforms: `iosSimulatorArm64`, `watchosSimulatorArm64`, and `tvosSimulatorArm64`. If you use the target shortcuts and want to build the project for an Apple Silicon simulator, make the following adjustment to the build script:

1. Add the `*SimulatorArm64` simulator target you need.
2. Connect the simulator target with the shortcut using the `dependsOn` relation between source sets.

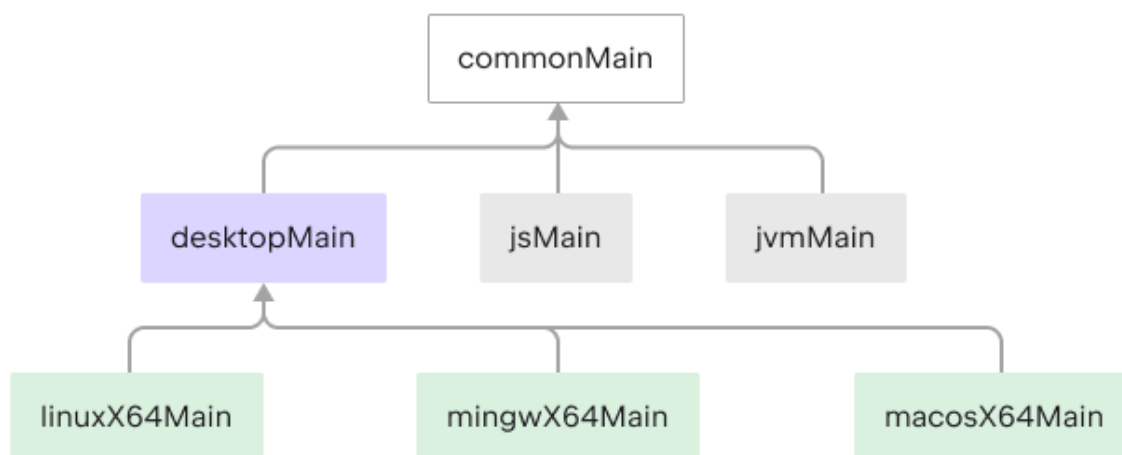
[Kotlin](#)[Groovy](#)

```
kotlin {  
    ios()  
    // Add the ARM64 simulator target  
    iosSimulatorArm64()  
  
    val iosMain by sourceSets.getting  
    val iosTest by sourceSets.getting  
    val iosSimulatorArm64Main by sourceSets.getting  
    val iosSimulatorArm64Test by sourceSets.getting  
  
    // Set up dependencies between the source sets  
    iosSimulatorArm64Main.dependsOn(iosMain)  
    iosSimulatorArm64Test.dependsOn(iosTest)  
}
```

## Manual configuration

You can manually introduce an intermediate source in the source set structure. It will hold the shared code for several targets.

For example, here's what to do if you want to share code among native Linux, Windows, and macOS targets ( `linuxX64` , `mingwX64` , and `macosX64` ):



1. Add the intermediate source set `desktopMain` , which holds the shared logic for these targets.
2. Specify the source set hierarchy using the `dependsOn` relation.

The resulting hierarchical structure will look like this:

Kotlin Groovy

```

kotlin {
    linuxX64()
    mingwX64()
    macosX64()

    sourceSets {
        val desktopMain by creating {
            dependsOn(commonMain.get())
        }
        val linuxX64Main by getting {
            dependsOn(desktopMain)
        }
        val mingwX64Main by getting {
            dependsOn(desktopMain)
        }
    }
}
  
```



```
    }  
    val macOSX64Main by getting {  
        dependsOn(desktopMain)  
    }  
}  
}
```

You can have a shared source set for the following combinations of targets:

- JVM or Android + JS + Native
- JVM or Android + Native
- JS + Native
- JVM or Android + JS
- Native

Kotlin doesn't currently support sharing a source set for these combinations:

- Several JVM targets
- JVM + Android targets
- Several JS targets

If you need to access platform-specific APIs from a shared native source set, IntelliJ IDEA will help you detect common declarations that you can use in the shared native code. For other cases, use the Kotlin mechanism of expected and actual declarations.