Conceptos avanzados de la estructura del proyecto multiplataforma

Este artículo explica los conceptos avanzados de la estructura del proyecto multiplataforma de Kotlin y cómo se asignan a la implementación de Gradle.

Esta información será útil si usted:

- Tener código compartido entre un conjunto específico de objetivos para los que Kotlin no crea un conjunto de fuentes de este tipo de forma predeterminada. En este caso, necesitas una API de nivel inferior que exponga algunas abstracciones nuevas.
- Necesidad de trabajar con abstracciones de bajo nivel de la compilación de Gradle, como configuraciones, tareas, publicaciones y otras.
- Están creando un complemento Gradle para las compilaciones multiplataforma de Kotlin.

Antes de sumergirnos en conceptos avanzados, recomendamos aprender los conceptos básicos de la estructura del proyecto multiplataforma.

Dependencias y dependencias

Esta sección describe dos tipos de dependencias:

- Depende de: una relación multiplataforma Kotlin específica entre dos conjuntos de fuentes de Kotlin.
- Dependencias regulares: dependencias de una biblioteca publicada, como kotlinx-coroutines, o de otro proyecto de Gradle en su compilación.

Por lo general, trabajarás con dependencias y no con la relación dependenciesOn. Sin embargo, examinar depende de On es crucial para entender cómo funcionan los proyectos multiplataforma de Kotlin bajo el capó.

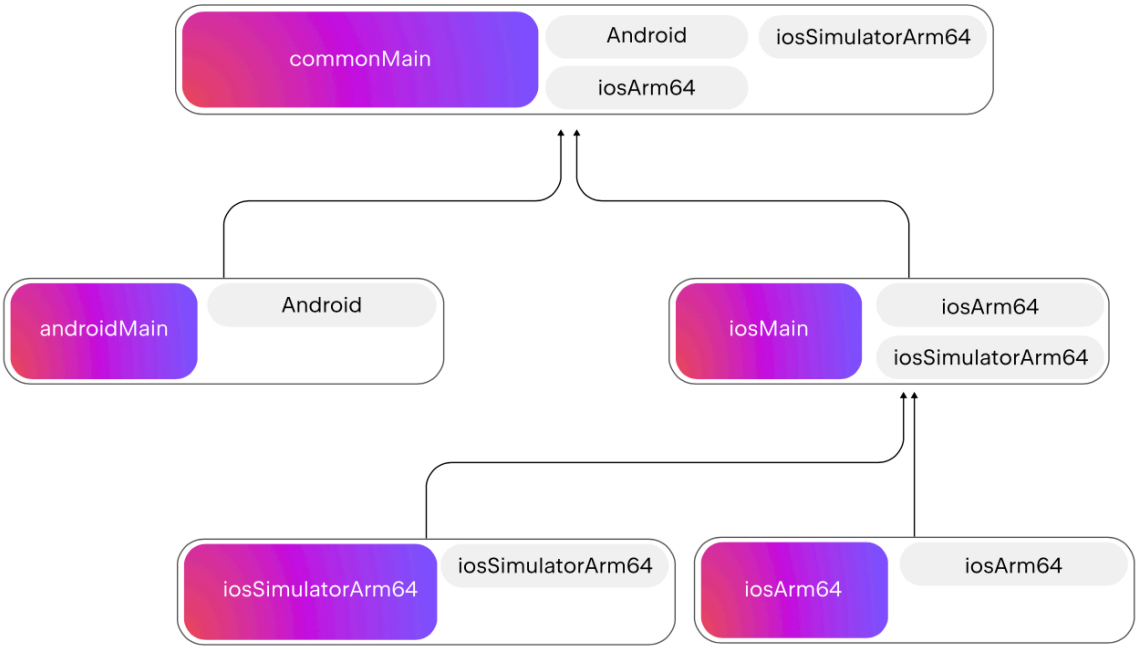
Jerarquías de dependencias y conjuntos de fuentes

dependsOn es una relación específica de Kotlin entre dos conjuntos de fuentes de Kotlin. Esto podría ser una conexión entre conjuntos de fuentes comunes y específicas de la plataforma, por ejemplo, cuando el conjunto de fuentes jvmMain depende de commonMain, iosArm64Main depende de iosMain, y así sucede.

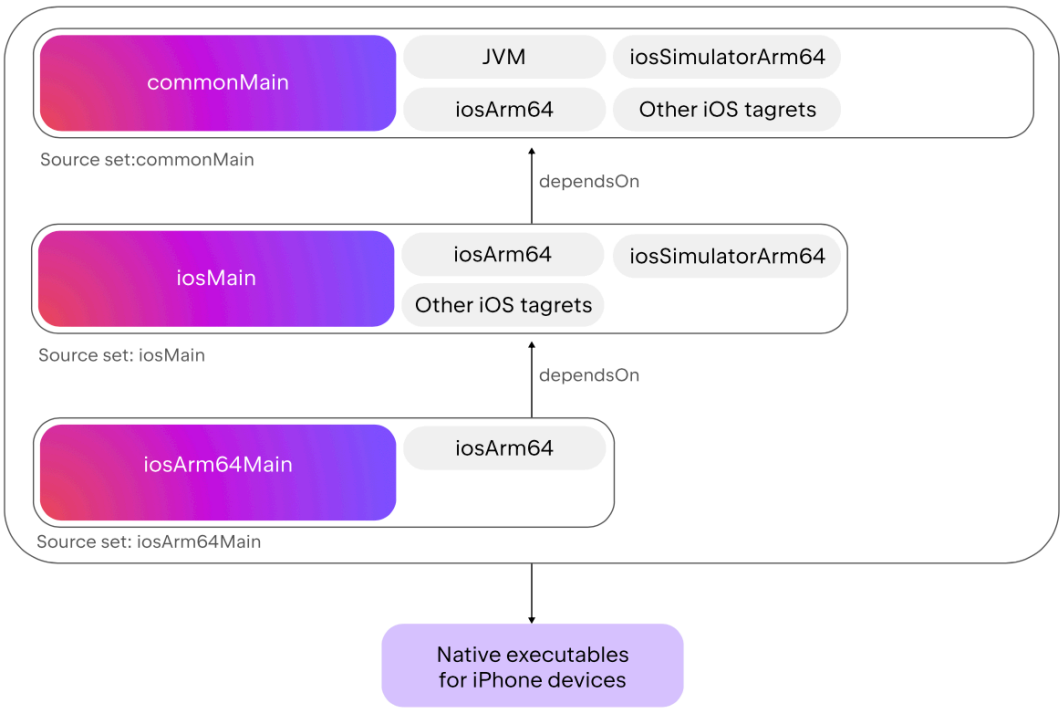
Considere un ejemplo general con los conjuntos de fuentes de Kotlin A y B. La expresión A.dependsOn(B) le indica a Kotlin que:

1. A observa la API de B, incluidas las declaraciones internas.
2. A puede proporcionar implementaciones reales para las declaraciones esperadas de B. Esta es una condición necesaria y suficiente, ya que A puede proporcionar datos reales para B si y solo si A. depende de On(B), ya sea directa o indirectamente.
3. B debe compilar en todos los objetivos a los que A compila, además de sus propios objetivos.
4. A hereda todas las dependencias regulares de B.

La relación dependsOn crea una estructura similar a un árbol conocida como jerarquía de conjuntos de fuentes. Aquí hay un ejemplo de un proyecto típico para el desarrollo móvil con androidTarget, iosArm64 (dispositivos iPhone) e iosSimulatorArm64 (simulador de iPhone para Apple Silicon Mac):



Las flechas representan depende de las relaciones. Estas relaciones se conservan durante la compilación de binarios de plataforma. Así es como Kotlin entiende que se supone que iosMain debe ver la API de commonMain, pero no de iosArm64Main:



Las relaciones dependsOn se configuran con la llamada KotlinSourceSet.dependsOn(KotlinSourceSet), por ejemplo:

```
kotlin {
    // Targets declaration
    sourceSets {
        // Example of configuring the dependsOn relation
        iosArm64Main.dependsOn(commonMain)
    }
}
```

- Este ejemplo muestra cómo se pueden definir las relaciones de dependencia en el script de compilación. Sin embargo, el complemento Kotlin Gradle crea conjuntos de fuentes y configura estas relaciones de forma predeterminada, por lo que no es necesario hacerlo manualmente.

- Las relaciones dependsOn se declaran por separado del bloque de dependencias {} en los scripts de compilación. Esto se debe a que dependsOn no es una dependencia regular; en cambio, es una relación específica entre los conjuntos de fuentes de Kotlin necesarios para compartir código entre diferentes objetivos.

No puede usar dependsOn para expresar dependencias regulares en una biblioteca publicada u otro proyecto de Gradle. Por ejemplo, no puede configurar commonMain para que dependa del commonMain de la biblioteca kotlinx-coroutines-core o llamar a commonTest.dependsOn(commonMain).

Dependencias de otras bibliotecas o proyectos

En proyectos multiplataforma, puede configurar dependencias regulares en una biblioteca publicada o en otro proyecto de Gradle.

Kotlin Multiplatform generalmente declara dependencias de una manera típica de Gradle. De manera similar a Gradle, usted:

Utilice el bloque de dependencias {} en su script de compilación.

Elija el alcance adecuado para las dependencias, por ejemplo, la implementación o la API.

Haga referencia a la dependencia especificando sus coordenadas si se publica en un repositorio, como "com.google.guava:guava:32.1.2-jre", o su ruta si es un proyecto Gradle en la misma construcción, como project(":utils:concurrency").

La configuración de dependencias en proyectos multiplataforma tiene algunas características especiales. Cada conjunto de fuentes de Kotlin tiene su propio bloque de dependencias {}. Esto le permite declarar dependencias específicas de la plataforma en conjuntos de fuentes específicas de la plataforma:

```
 kotlin {
    // Targets declaration
    sourceSets {
        jvmMain.dependencies {
            // This is jvmMain's dependencies, so it's OK to add a JVM-
specific dependency
            implementation("com.google.guava:guava:32.1.2-jre")
        }
    }
}
```

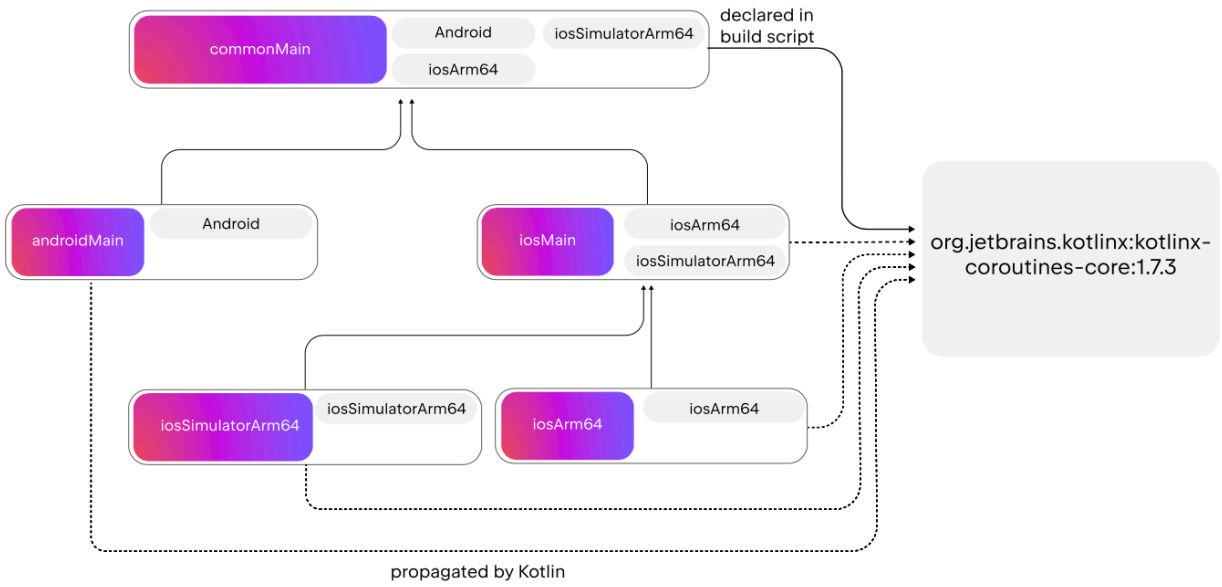
Las dependencias comunes son más complicadas. Considere un proyecto multiplataforma que declare una dependencia de una biblioteca multiplataforma, por ejemplo, kotlinx.coroutines:

```
kotlin {
    androidTarget()     // Android
    iosArm64()          // iPhone devices
    iosSimulatorArm64() // iPhone simulator on Apple Silicon Mac

    sourceSets {
        commonMain.dependencies {
            implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:1.7.3")
        }
    }
}
```

Hay tres conceptos importantes en la resolución de dependencias:

1 Las dependencias multiplataforma se propagan por la estructura de dependencias. Cuando agregue una dependencia a commonMain, se agregará automáticamente a todos los conjuntos de fuentes que declaren relaciones dependientes directa o indirectamente en commonMain.

   En este caso, la dependencia se agregó automáticamente a todos los conjuntos de fuentes *Main: iosMain, jvmMain, iosSimulatorArm64Main e iosX64Main. Todos estos conjuntos de fuentes heredan la dependencia kotlin-coroutines-core del conjunto de fuentes commonMain, por lo que no tiene que copiarla y pegarla en todas ellas manualmente:

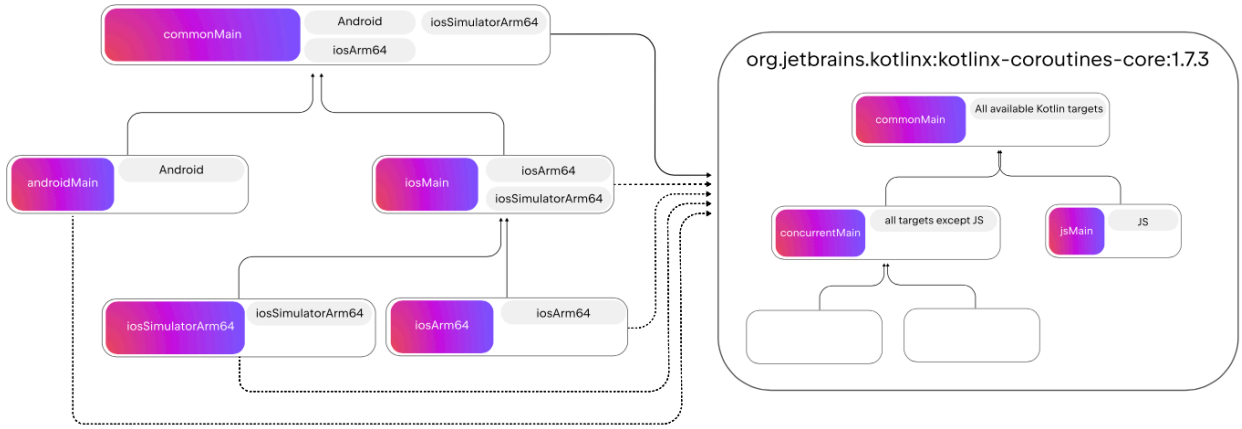


El mecanismo de propagación le permite elegir un alcance que recibirá la dependencia declarada seleccionando un conjunto de fuentes específico. Por ejemplo, si quieres usar kotlinx.coroutines en iOS pero no en Android, puedes añadir esta dependencia solo a iOSMain.

El conjunto de fuentes → dependencias de la biblioteca multiplataforma, como commonMain a org.jetbrians.kotlinx:kotlinx-coroutines-core:1.7.3 anterior, representan el estado intermedio de resolución de dependencias. El estado final de resolución siempre está representado por el conjunto de fuentes → dependencias del conjunto de fuentes.

El conjunto de fuentes finales → las dependencias del conjunto de fuentes no dependen de las relaciones.

Para inferir el conjunto de fuentes granulares → dependencias del conjunto de fuentes, Kotlin lee la estructura del conjunto de fuentes que se publica junto con cada biblioteca multiplataforma. Después de este paso, cada biblioteca estará representada internamente no como un todo, sino como una colección de sus conjuntos de fuentes. Vea este ejemplo para kotlinx-coroutines-core:



3. Kotlin toma cada relación de dependencia y la resuelve en una colección de conjuntos de fuentes de una dependencia. Cada fuente de dependencia establecida en esa colección debe tener objetivos compatibles. Un conjunto de fuentes de dependencia tiene objetivos compatibles si se compila con al menos los mismos objetivos que el conjunto de fuentes del consumidor.

   Considere un ejemplo en el que commonMain en el proyecto de muestra se compila en androidTarget, iosX64 e iosSimulatorArm64:

   En primer lugar, resuelve una dependencia de kotlinx-coroutines-core.commonMain. Esto sucede porque kotlinx-coroutines-core se compila en todos los objetivos posibles de Kotlin. Por lo tanto, su commonMain se compila en todos los objetivos posibles, incluidos los necesarios androidTarget, iosX64 e iosSimulatorArm64.

   En segundo lugar, commonMain depende de kotlinx-coroutines-core.concurrentMain. Dado que concurrentMain en kotlinx-coroutines-core se compila con todos los objetivos, excepto JS, coincide con los objetivos del commonMain del proyecto del consumidor.

   Sin embargo, los conjuntos de fuentes como iosX64Main de las corrutinas son incompatibles con el commonMain del consumidor. A pesar de que iosX64Main se compila en uno de los objetivos de commonMain, a saber, iosX64, no se compila ni en androidTarget ni en iosSimulatorArm64.

   Los resultados de la resolución de dependencias afectan directamente a cuál del código en kotlinx-coroutines-core es visible:

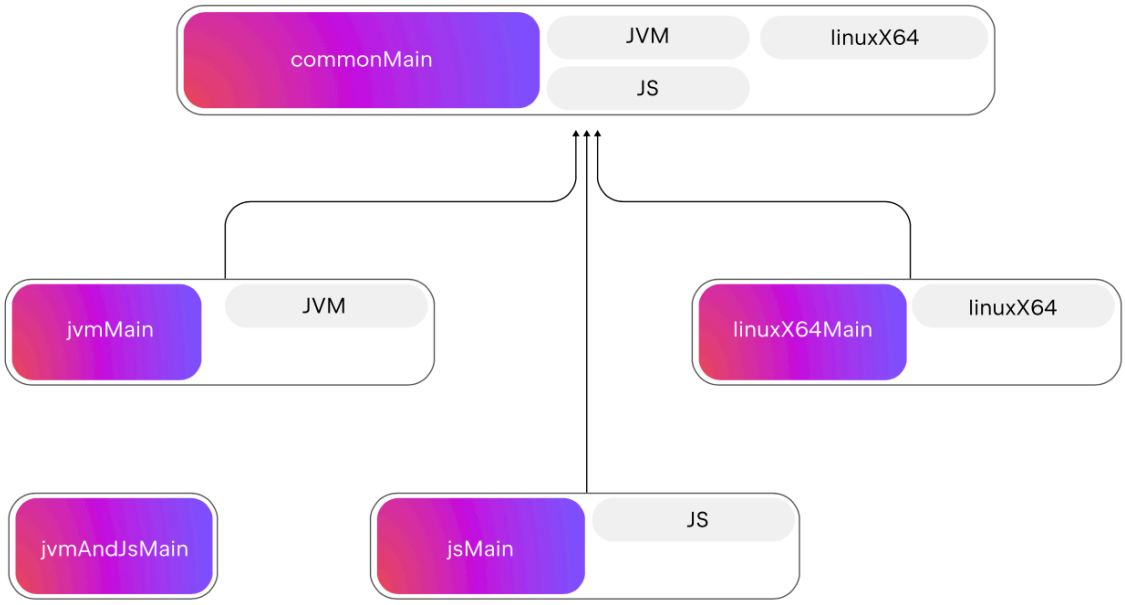
Declarar conjuntos de fuentes personalizados

En algunos casos, es posible que necesite tener una fuente intermedia personalizada en su proyecto. Considere un proyecto que se compile en JVM, JS y Linux, y desea compartir algunas fuentes solo entre JVM y JS. En este caso, debería encontrar un conjunto de fuentes específico para este par de objetivos, como se describe en Los conceptos básicos de la estructura del proyecto multiplataforma.

Kotlin no crea un conjunto de fuentes de este tipo automáticamente. Esto significa que debes crearlo manualmente con el mediante la creación de la construcción:

```
kotlin {
    jvm()
    js()
    linuxX64()

    sourceSets {
        // Create a source set named "jvmAndJs"
        val jvmAndJsMain by creating {
            // …
        }
    }
}
```

However, Kotlin still doesn't know how to treat or compile this source set. If you drew a diagram, this source set would be isolated and wouldn't have any target labels:

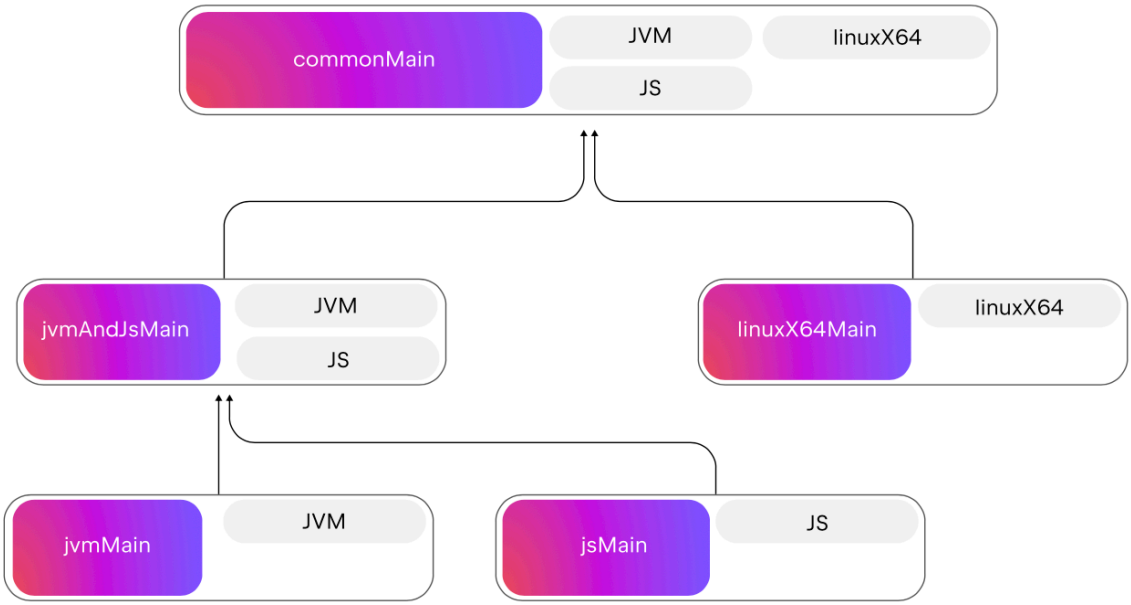


Para solucionar esto, incluya jvmAndJsMain en la jerarquía añadiendo varias relaciones dependientes:

```
kotlin {
    jvm()
    js()
    linuxX64()

    sourceSets {
        val jvmAndJsMain by creating {
            // Don't forget to add dependsOn to commonMain
            dependsOn(commonMain.get())
        }

        jvmMain {
            dependsOn(jvmAndJsMain)
        }

        jsMain {
            dependsOn(jvmAndJsMain)
        }
    }
}
```

Aquí, jvmMain.dependsOn(jvmAndJsMain) agrega el objetivo JVM a jvmAndJsMain, y jsMain.dependsOn(jvmAndJsMain) agrega el objetivo JS a jvmAndJsMain.

La estructura final del proyecto se verá así:

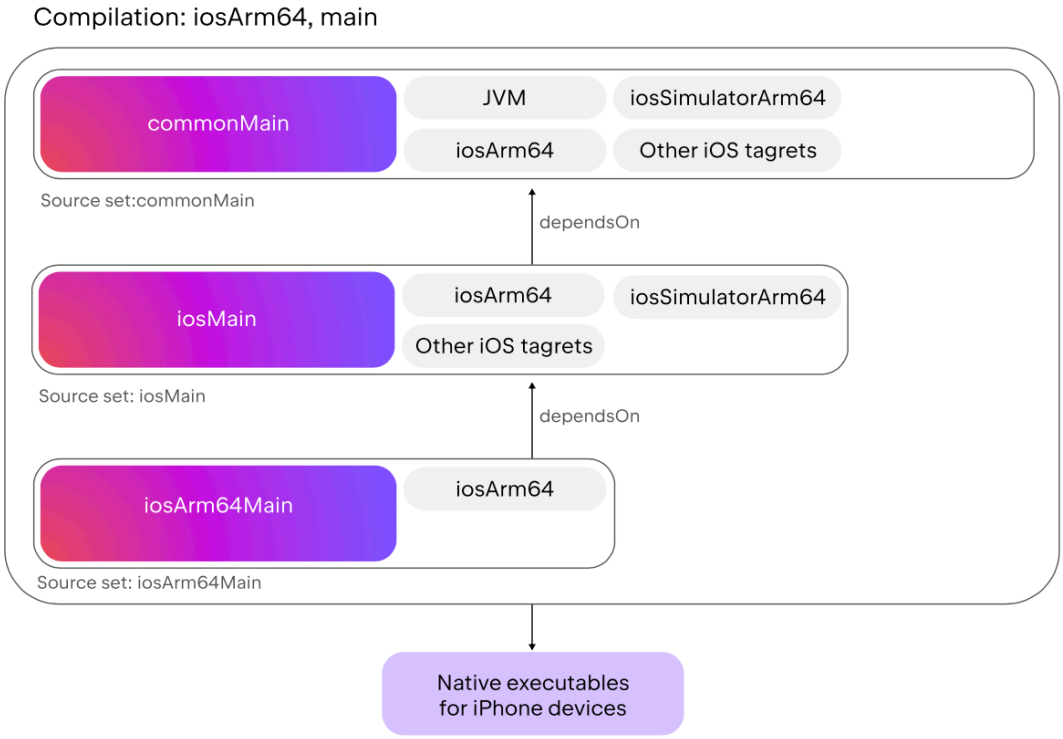


Manual configuration of dependsOn relations disables automatic application of the default hierarchy template. See Additional configuration to learn more about such cases and how to handle them.

Compilaciones

A diferencia de los proyectos de plataforma única, los proyectos multiplataforma de Kotlin requieren múltiples lanzamientos de compiladores para construir todos los artefactos. Cada lanzamiento del compilador es una compilación de Kotlin.

Por ejemplo, así es como se generan los binarios para dispositivos iPhone durante esta compilación de Kotlin mencionada anteriormente:



Las compilaciones de Kotlin se agrupan bajo objetivos. De forma predeterminada, Kotlin crea dos compilaciones para cada objetivo, la compilación principal para las fuentes de producción y la compilación de prueba para las fuentes de prueba.

Se accede a las compilaciones en scripts de compilación de una manera similar. Primero selecciona un objetivo de Kotlin, luego accede al contenedor de compilaciones dentro y, finalmente, elige la compilación necesaria por su nombre:

```
kotlin {
    // Declare and configure the JVM target
    jvm {
        val mainCompilation: KotlinJvmCompilation =
compilations.getByName("main")
    }
}
```