

Los conceptos básicos de la estructura del proyecto multiplataforma de Kotlin

Con Kotlin Multiplatform, puedes compartir código entre diferentes plataformas. Este artículo explica las restricciones del código compartido, cómo distinguir entre partes compartidas y específicas de la plataforma de su código, y cómo especificar las plataformas en las que funciona este código compartido.

También aprenderá los conceptos básicos de la configuración del proyecto multiplataforma de Kotlin, como el código común, los objetivos, los conjuntos de fuentes intermedias y específicas de la plataforma, y la integración de pruebas. Eso te ayudará a configurar tus proyectos multiplataforma en el futuro.

El modelo que se presenta aquí se simplifica en comparación con el utilizado por Kotlin. Sin embargo, este modelo básico debería ser adecuado para la mayoría de los casos.

Código común

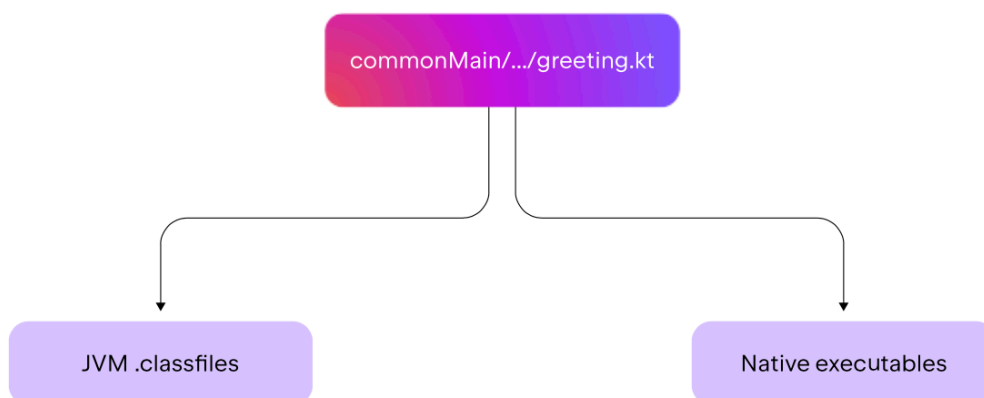
El código común es el código Kotlin compartido entre diferentes plataformas.

Considere el simple ejemplo de "Hola, mundo":

```
fun greeting() {  
    println("Hello, Kotlin Multiplatform!")  
}
```

El código de Kotlin compartido entre las plataformas se encuentra normalmente en el directorio principal común. La ubicación de los archivos de código es importante, ya que afecta a la lista de plataformas en las que se compila este código.

El compilador de Kotlin obtiene el código fuente como entrada y produce un conjunto de binarios específicos de la plataforma como resultado. Al compilar proyectos multiplataforma, puede producir varios binarios a partir del mismo código. Por ejemplo, el compilador puede producir archivos JVM .class y archivos ejecutables nativos desde el mismo archivo Kotlin:



No todas las piezas de código de Kotlin se pueden compilar en todas las plataformas. El compilador Kotlin le impide utilizar funciones o clases específicas de la plataforma en su código común, ya que este código no se puede compilar en una plataforma diferente.

Por ejemplo, no puedes usar la dependencia de `java.io.File` del código común. Es parte del JDK, mientras que el código común también se compila en código nativo, donde las clases JDK no están disponibles:




En el código común, puedes usar las bibliotecas multiplataforma de Kotlin. Estas bibliotecas proporcionan una API común que se puede implementar de manera diferente en diferentes plataformas. En este caso, las API específicas de la plataforma sirven como partes adicionales, y tratar de usar dicha API en código común da como resultado un error.

Por ejemplo, `kotlinx.coroutines` es una biblioteca multiplataforma de Kotlin que admite todos los objetivos, pero también tiene una parte específica de la plataforma que convierte las primitivas concurrentes de `kotlinx.coroutines` en primitivas concurrentes de JDK, como la diversión `CoroutinesDispatcher.asExecutor(): Executor`. Esta parte adicional de la API no está disponible en `CommonMain`.

Objetivos

Los objetivos definen las plataformas en las que Kotlin compila el código común. Estos podrían ser, por ejemplo, la JVM, JS, Android, iOS o Linux. El ejemplo anterior compiló el código común para la JVM y los objetivos nativos.

Un objetivo Kotlin es un identificador que describe un objetivo de compilación. Define el formato de los binarios producidos, las construcciones de idiomas disponibles y las dependencias permitidas.

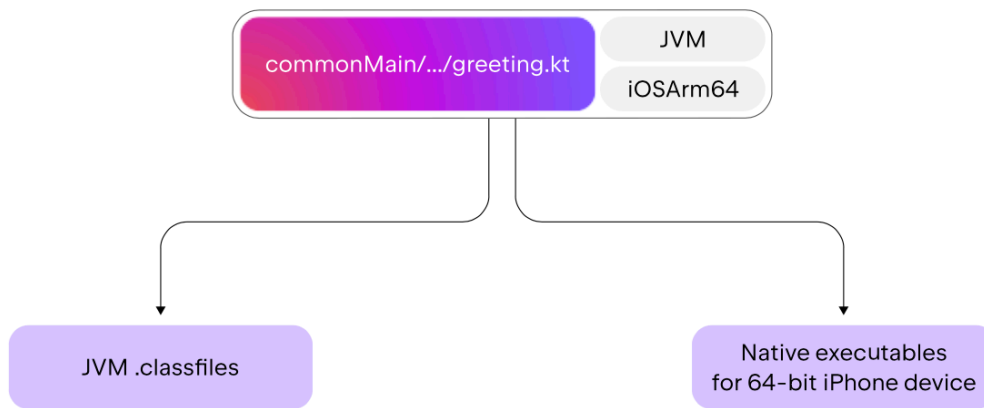
 Los objetivos también se pueden denominar plataformas. Vea la lista completa de objetivos compatibles.

Los objetivos también se pueden denominar plataformas. Vea la lista completa de objetivos compatibles.

```
kotlin {  
    jvm() // Declares a JVM target  
    iosArm64() // Declares a target that corresponds to 64-bit iPhones  
}
```

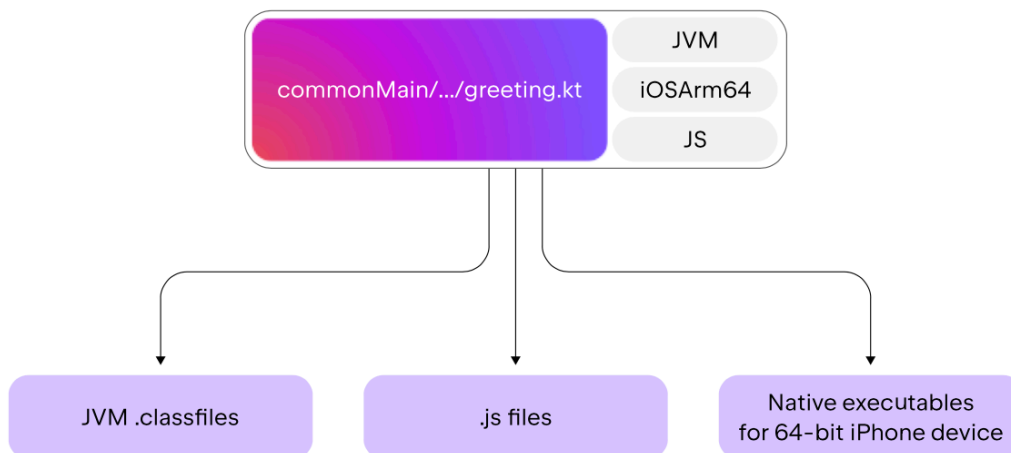
De esta manera, cada proyecto multiplataforma define un conjunto de objetivos compatibles. Consulte la sección Estructura jerárquica del proyecto para obtener más información sobre la declaración de objetivos en sus scripts de compilación.

Con los objetivos `jvm` e `iosArm64` declarados, el código común en `commonMain` se compilará con estos objetivos:



Para entender qué código se va a compilar para un objetivo específico, puedes pensar en un objetivo como una etiqueta adjunta a los archivos fuente de Kotlin. Kotlin utiliza estas etiquetas para determinar cómo compilar su código, qué binarios producir y qué construcciones y dependencias de lenguaje están permitidas en ese código.

Si también quieres compilar el archivo `greeting.kt` en `.js`, solo tienes que declarar el destino de JS. El código en `commonMain` recibe una etiqueta `js` adicional, correspondiente al objetivo JS, que indica a Kotlin que produzca archivos `.js`:



Así es como funciona el compilador Kotlin con el código común compilado en todos los destinos declarados. Consulte los conjuntos de fuentes para aprender a escribir código específico de la plataforma.

Conjuntos de fuentes

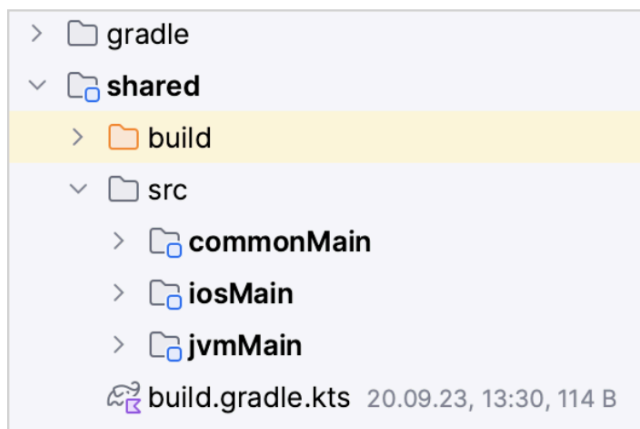
Un conjunto de fuentes Kotlin es un conjunto de archivos fuente con sus propios objetivos, dependencias y opciones de compilador. Es la forma principal de compartir código en proyectos multiplataforma.

- Cada fuente establecida en un proyecto multiplataforma:
- Tiene un nombre que es único para un proyecto determinado.
- Contiene un conjunto de archivos y recursos de origen, generalmente almacenados en el directorio con el nombre del conjunto de origen.
- Especifica un conjunto de objetivos a los que se compila el código de este conjunto de fuentes. Estos objetivos afectan a las construcciones y dependencias del lenguaje que están disponibles en este conjunto de fuentes.
- Define sus propias dependencias y las opciones del compilador.

Kotlin proporciona un montón de conjuntos de fuentes predefinidos. Uno de ellos es `commonMain`, que está presente en todos los proyectos multiplataforma y compila en todos los objetivos declarados.

Interactúas con conjuntos de fuentes como directorios dentro de `src` en los proyectos multiplataforma de Kotlin. Por ejemplo, un proyecto con los conjuntos de fuentes `commonMain`,

`iosMain` y `jvmMain` tiene la siguiente estructura:



En los scripts de Gradle, se accede a los conjuntos de fuentes por nombre dentro del bloque `kotlin.sourceSets {}`:

```
kotlin {
    // Targets declaration:
    // ...

    // Source set declaration:
    sourceSets {
        commonMain {
            // configure the commonMain source set
        }
    }
}
```

Además de `commonMain`, otros conjuntos de fuentes pueden ser específicos de la plataforma o intermedios.

Conjuntos de fuentes específicos de la plataforma

Si bien tener solo código común es conveniente, no siempre es posible. El código en `commonMain` se compila en todos los objetivos declarados, y Kotlin no le permite utilizar ninguna API específica de la plataforma allí.

En un proyecto multiplataforma con objetivos nativos y JS, el siguiente código en `commonMain` no se compila:

```
// commonMain/kotlin/common.kt
// Doesn't compile in common code
fun greeting() {
    java.io.File("greeting.txt").writeText("Hello, Multiplatform!")
}
```

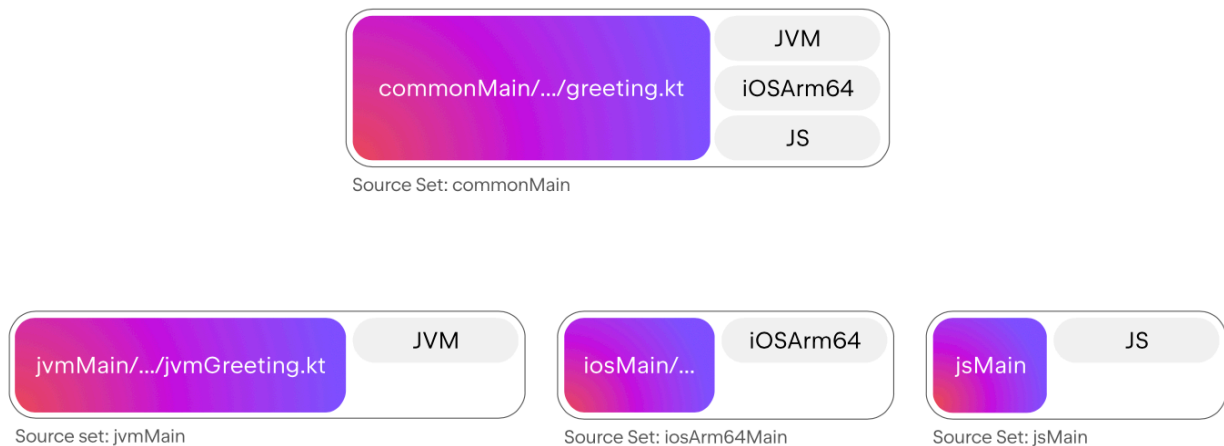
Como solución, Kotlin crea conjuntos de fuentes específicos de la plataforma, también conocidos como conjuntos de fuentes de la plataforma. Cada objetivo tiene un conjunto de fuentes de plataforma correspondiente que se compila solo para ese objetivo. Por ejemplo, un objetivo de `jvm` tiene el conjunto de fuentes `jvmMain` correspondiente que se compila solo en la JVM. Kotlin permite el uso de dependencias específicas de la plataforma en estos conjuntos de fuentes, por ejemplo, `JDK` en `jvmMain`:

```
// jvmMain/kotlin/jvm.kt
// You can use Java dependencies in the `jvmMain` source set
fun jvmGreeting() {
    java.io.File("greeting.txt").writeText("Hello, Multiplatform!")
}
```

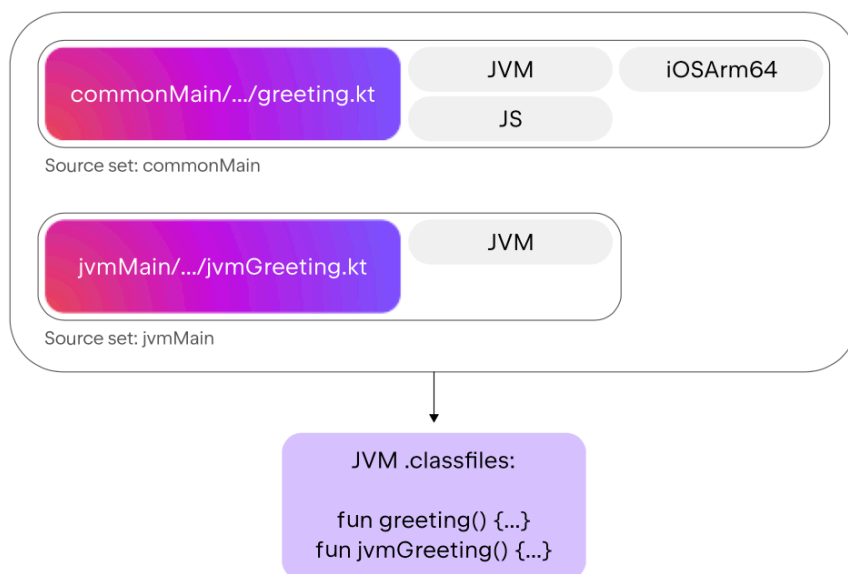
Compilación a un objetivo específico

La compilación a un objetivo específico funciona con múltiples conjuntos de fuentes. Cuando Kotlin compila un proyecto multiplataforma en un objetivo específico, recopila todos los conjuntos de fuentes etiquetados con este objetivo y produce binarios a partir de ellos.

Considere un ejemplo con los objetivos de jvm, iosArm64 y js. Kotlin crea el conjunto de fuentes `commonMain` para el código común y los correspondientes conjuntos de fuentes `jvmMain`, `iosArm64Main` y `jsMain` para objetivos específicos:



Durante la compilación en la JVM, Kotlin selecciona todos los conjuntos de fuentes etiquetadas con "JVM", a saber, `jvmMain` y `commonMain`. Luego los compila en los archivos de clase JVM:



Debido a que Kotlin compila `commonMain` y `jvmMain` juntos, los binarios resultantes contienen declaraciones tanto de `commonMain` como de `jvmMain`.

Cuando trabajes con proyectos multiplataforma, recuerda:

- Si quieres que Kotlin compile tu código en una plataforma específica, declara un destino correspondiente.
- Para elegir un directorio o archivo fuente para almacenar el código, primero decida entre qué objetivos desea compartir su código:
- Si el código se comparte entre todos los objetivos, debe declararse en `commonMain`.
- Si el código se utiliza para un solo objetivo, debe definirse en un conjunto de fuentes específica de la plataforma para ese objetivo (por ejemplo, `jvmMain` para la JVM).
- El código escrito en conjuntos de fuentes específicos de la plataforma puede acceder a las declaraciones desde el conjunto de fuentes comunes. Por ejemplo, el código de `jvmMain` puede usar código de `commonMain`. Sin embargo, lo contrario no es cierto: `commonMain` no puede usar el código de `jvmMain`.
- El código escrito en conjuntos de fuentes específicos de la plataforma puede utilizar las dependencias correspondientes de la plataforma. Por ejemplo, el código de `jvmMain` puede usar bibliotecas solo de Java, como Guava o Spring.

Conjuntos de fuentes intermedias

Los proyectos multiplataforma simples generalmente solo tienen código común y específico de la plataforma. El conjunto de fuentes `commonMain` representa el código común compartido entre todos los objetivos declarados. Los conjuntos de fuentes específicos de la plataforma, como `jvmMain`, representan un código específico de la plataforma compilado solo para el destino respectivo.

En la práctica, a menudo necesitas compartir código más granular.

Considere un ejemplo en el que necesita dirigirse a todos los dispositivos Apple modernos y Android:

```
kotlin {
    android()
    iosArm64()    // 64-bit iPhone devices
    macosArm64() // Modern Apple Silicon-based Macs
    watchosX64() // Modern 64-bit Apple Watch devices
    tvosArm64()  // Modern Apple TV devices
}
```

And you need a source set to add a function that generates a UUID for all Apple devices:

```
import platform.Foundation.NSUUID

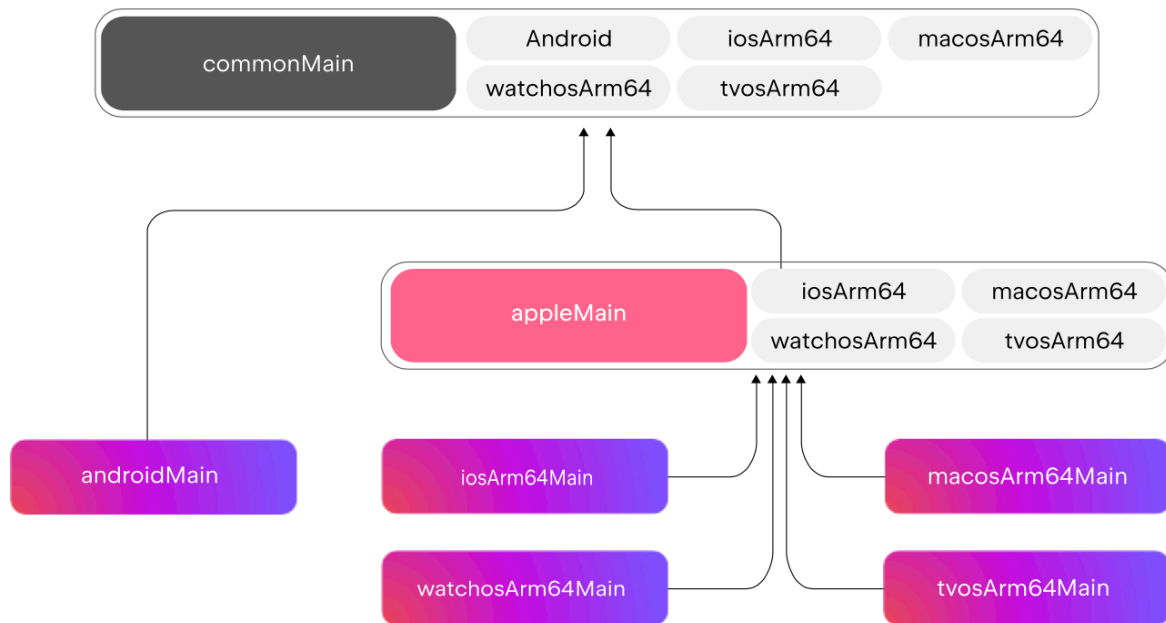
fun randomUuidString(): String {
    // You want to access Apple-specific APIs
    return NSUUID().UUIDString()
}
```

No se puede agregar esta función a `commonMain`. `commonMain` se compila en todos los objetivos declarados, incluido Android, pero `platform.Foundation.NSUUID` es una API específica de Apple que no está disponible en Android. Kotlin muestra un error si intenta hacer referencia a `NSUUID` en `commonMain`.

Puedes copiar y pegar este código en cada conjunto de fuentes específico de Apple: `iosArm64Main`, `macosArm64Main`, `watchosX64Main` y `tvosArm64Main`. Pero este enfoque no se recomienda porque duplicar código como este es propenso a errores.

Para resolver este problema, puede utilizar conjuntos de fuentes intermedias. Un conjunto de fuentes intermedias es un conjunto de fuentes de Kotlin que se compila con algunos, pero no con todos los objetivos del proyecto. También puede ver conjuntos de fuentes intermedias a los que se hace referencia como conjuntos de fuentes jerárquicas o simplemente jerarquías.

Kotlin crea algunos conjuntos de fuentes intermedias de forma predeterminada. En este caso específico, la estructura del proyecto resultante se verá así:



No se puede agregar esta función a `commonMain`. `commonMain` se compila en todos los objetivos declarados, incluido Android, pero `platform.Foundation.NSUUID` es una API específica de Apple que no está disponible en Android. Kotlin muestra un error si intenta hacer referencia a `NSUUID` en `commonMain`.

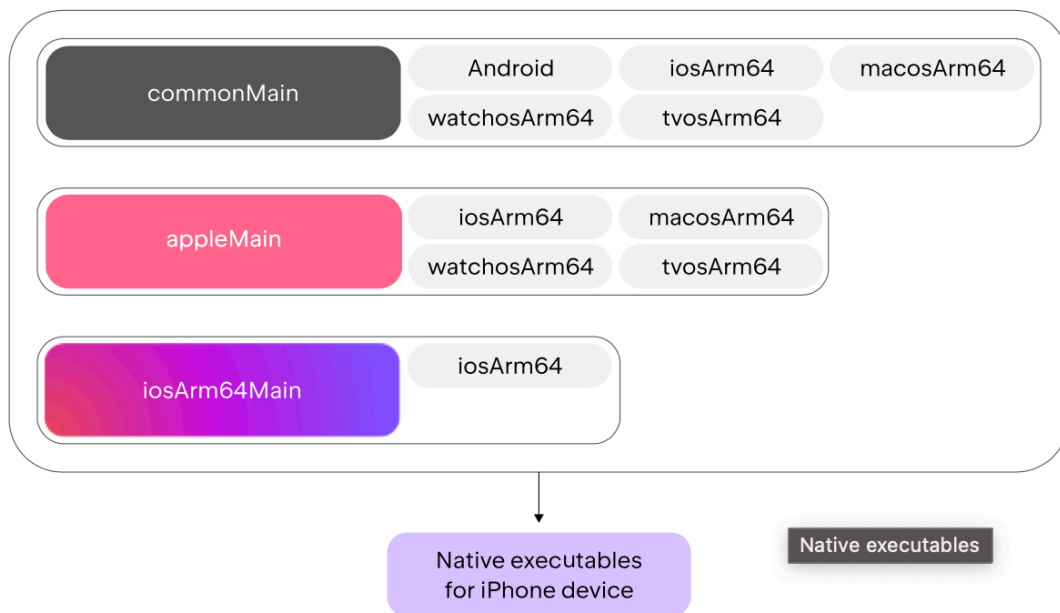
Puedes copiar y pegar este código en cada conjunto de fuentes específico de Apple: `iosArm64Main`, `macosArm64Main`, `watchosX64Main` y `tvosArm64Main`. Pero este enfoque no se recomienda porque duplicar código como este es propenso a errores.

Para resolver este problema, puede utilizar conjuntos de fuentes intermedias. Un conjunto de fuentes intermedias es un conjunto de fuentes de Kotlin que se compila con algunos, pero no con todos los objetivos del proyecto. También puede ver conjuntos de fuentes intermedias a los que se hace referencia como conjuntos de fuentes jerárquicas o simplemente jerarquías.

Kotlin crea algunos conjuntos de fuentes intermedias de forma predeterminada. En este caso específico, la estructura del proyecto resultante se verá así:

Consulte la estructura jerárquica del proyecto para encontrar todos los conjuntos de fuentes intermedias que Kotlin crea y configura de forma predeterminada y aprenda qué debe hacer si Kotlin no proporciona el conjunto de fuentes intermedias que necesita de forma predeterminada.

Durante la compilación a un objetivo específico, Kotlin obtiene todos los conjuntos de fuentes, incluidos los conjuntos de fuentes intermedias, etiquetados con este objetivo. Por lo tanto, todo el código escrito en los conjuntos de fuentes `commonMain`, `appleMain` e `iosArm64Main` se combina durante la compilación en el destino de la plataforma `iosArm64`:



📖 Está bien si algunos conjuntos de fuentes no tienen fuentes. Por ejemplo, en el desarrollo de iOS, generalmente no hay necesidad de proporcionar un código que sea específico para dispositivos iOS, pero no para simuladores de iOS. Por lo tanto, iosArm64Main rara vez se utiliza.

Objetivos de dispositivos y simuladores de Apple

Cuando utilizas Kotlin Multiplatform para desarrollar aplicaciones móviles de iOS, normalmente trabajas con el conjunto de fuentes `iosMain`. Si bien podrías pensar que es un conjunto de fuentes específicas de la plataforma para el objetivo de iOS, no hay un único objetivo de iOS. La mayoría de los proyectos móviles necesitan al menos dos objetivos:

- **El objetivo del dispositivo** se utiliza para generar binarios que se pueden ejecutar en dispositivos iOS. Actualmente solo hay un objetivo de dispositivo para iOS: `iosArm64`.
- **El objetivo del simulador** se utiliza para generar binarios para el simulador de iOS lanzado en su máquina. Si tienes un ordenador Mac de silicio Apple, elige `iosSimulatorArm64` como objetivo de simulador. Usa `iosX64` si tienes un ordenador Mac basado en Intel.

Si declara solo el destino del dispositivo `iosArm64`, no podrá ejecutar y depurar su aplicación y pruebas en su máquina local.

Los conjuntos de fuentes específicas de la plataforma como `iosArm64Main`, `iosSimulatorArm64Main` e `iosX64Main` suelen estar vacíos, ya que el código Kotlin para dispositivos y simuladores iOS es normalmente el mismo. Solo puedes usar el conjunto de fuentes intermedias de `iosMain` para compartir código entre todos ellos.

Lo mismo se aplica a otros objetivos de Apple que no son de Mac. Por ejemplo, si tiene el objetivo del dispositivo `tvosArm64` para Apple TV y los objetivos del simulador `tvosSimulatorArm64` y `tvosX64` para simuladores de Apple TV en dispositivos Apple Silicon e Intel, respectivamente, puede utilizar el conjunto de fuentes intermedias `tvosMain` para todos ellos.

Integración con pruebas

Los proyectos de la vida real también requieren pruebas junto con el código de producción principal. Esta es la razón por la que todos los conjuntos de fuentes creados de forma predeterminada tienen los prefijos principal y de prueba. `Main` contiene código de producción, mientras que `Test` contiene pruebas para este código. La conexión entre ellos se establece automáticamente, y las pruebas pueden utilizar la API proporcionada por el código principal sin configuración adicional.

Las contrapartes de prueba también son conjuntos de fuentes similares a los principales. Por ejemplo, `commonTest` es una contraparte de `commonMain` y se compila en todos los objetivos declarados, lo que le permite escribir pruebas comunes. Los conjuntos de fuentes de pruebas específicas de la plataforma, como `jvmTest`, se utilizan para escribir pruebas específicas de la plataforma, por ejemplo, pruebas específicas de JVM o pruebas que necesitan API de JVM.

Además de tener una configuración de código fuente para escribir su prueba común, también necesita un marco de pruebas multiplataforma. Kotlin proporciona una biblioteca predeterminada de `kotlin.test` que viene con la anotación `@kotlin.Test` y varios métodos de aserción como `assertEquals` y `assertTrue`.

Puedes escribir pruebas específicas de la plataforma como pruebas regulares para cada plataforma en sus respectivos conjuntos de fuentes. Al igual que con el código principal, puede tener dependencias específicas de la plataforma para cada conjunto de fuentes, como `JUnit` para JVM y `XCTest` para iOS. Para ejecutar pruebas para un objetivo en particular, utilice la tarea `<targetName>Test`.

Aprende a crear y ejecutar pruebas multiplataforma en el tutorial [Prueba tu aplicación multiplataforma](#).