KMP & Compose Multiplatform

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

Motivation

Imagine a scenario where you have code to share, either across products with similar features, or a single project that you want to deploy across platforms e.g., Android and iOS.

Kotlin Multiplatform (KMP) is a technology from JetBrains to enable sharing code across Android, iOS, desktop, web, and server while retaining the advantages of native development.

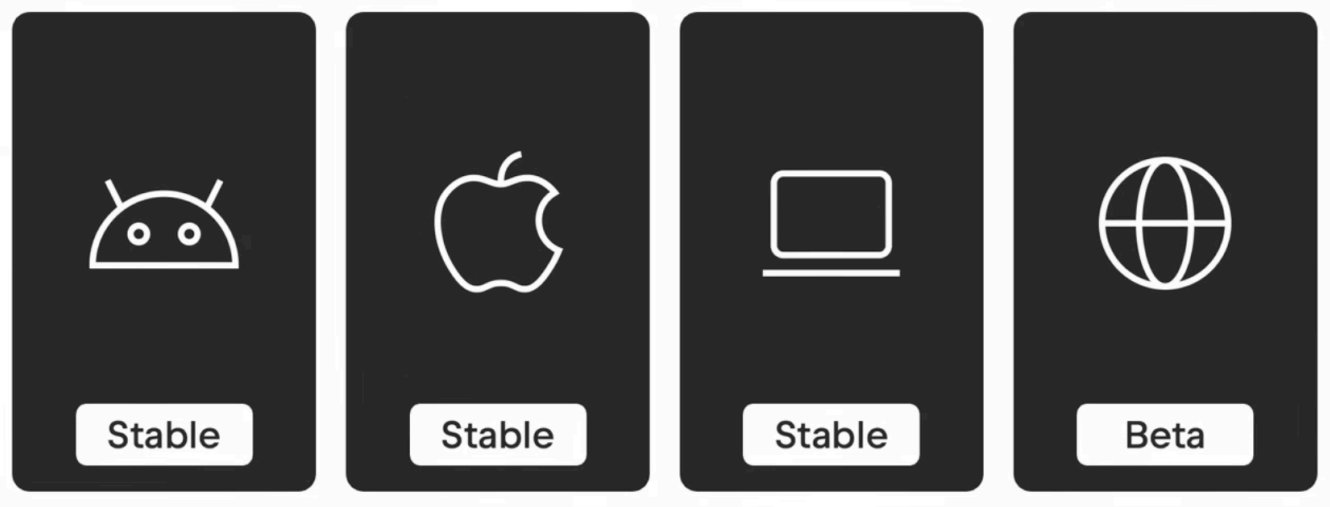


Figure 1: Kotlin Multiplatform lets you share code between Android, iOS and Desktop (web in beta).

Kotlin Multiplatform

Kotlin Multiplatform (KMP) lets you choose how much code to share, and how to share it when developing cross-platform e.g., Android, iOS and desktop.

Options include:

- A Kotlin project with a small amount of shared code e.g., an exported library.
- A Kotlin multiplatform project with shared logic, that uses native UI for each platform e.g., Swift UI for iOS and Compose for Android and Desktop.
- A Kotlin multiplatform project with a 100% shared UI.

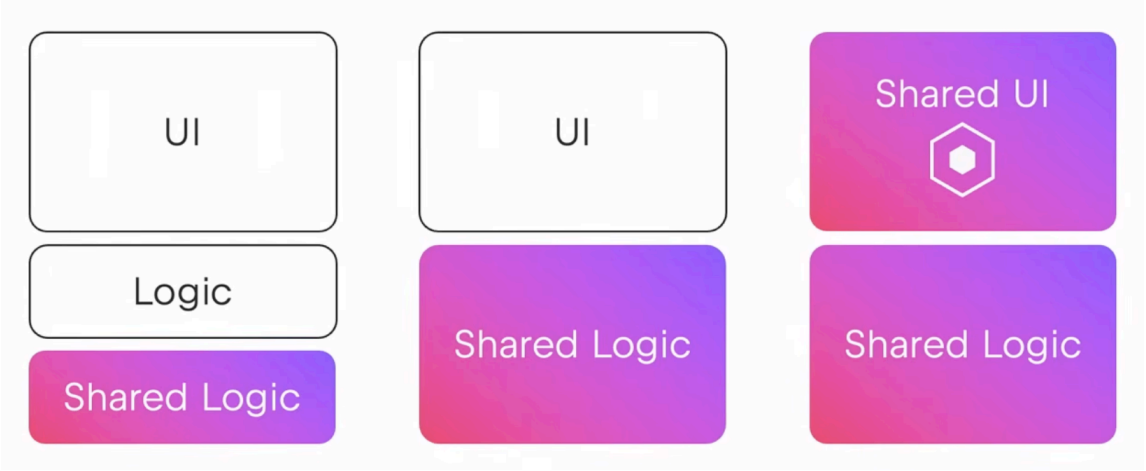
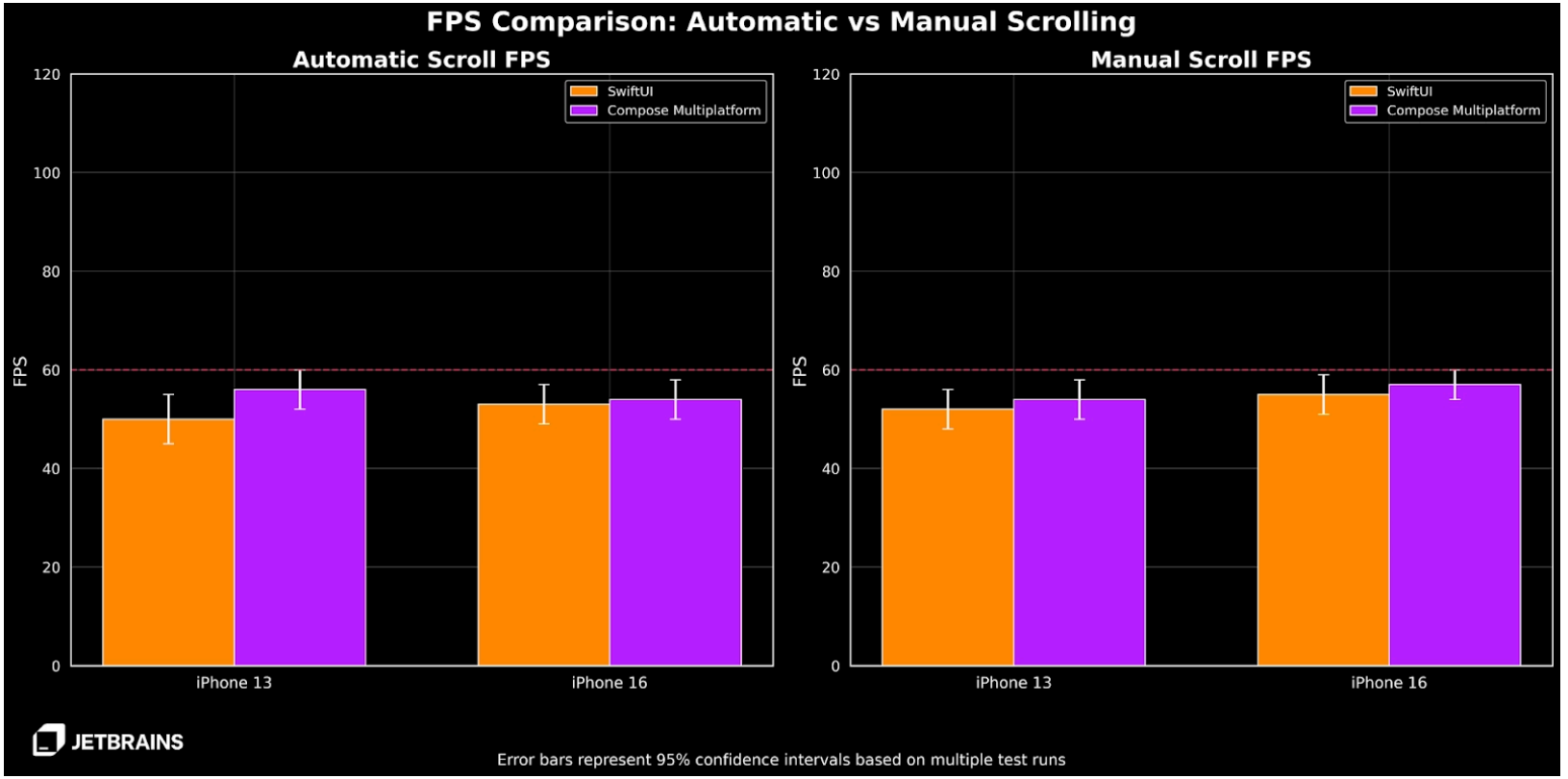


Figure 2: You can decide how much Kotlin code to share, and can strategically mix shared and native logic in the same codebase.

Kotlin Multiplatform

Kotlin Multiplatform uses the `Kotlin/Native` compiler to produce native binaries and access platform APIs directly e.g., Android and iOS.

- This helps achieve near-native performance on all platforms, and a native look-and-feel with supported UI toolkits. [1]



Compose Multiplatform

KMP is fantastic for business logic, but by itself doesn't support a shared UI. For that we need a UI toolkit that runs on each platform.

Compose Multiplatform is a UI framework that lets you build a shared UI in Kotlin, that works with all supported platforms.

- You build your user interface code in Kotlin + Compose.
- You can integrate with native UI components if needed!
- Compose libraries are integrated into the native executable.
- Your application runs at near-native performance and looks/feels native.

Compose Multiplatform

KMP + Compose Multiplatform together provide options for code sharing.

- **Option 1: Kotlin projects share some code**
  - Useful where you want to code share across existing projects e.g., you have an iOS app and want to create a new Android app.
  - Write new logic in Kotlin, and export this functionality as a library to be consumed by both projects.
- **Option 2: Kotlin with shared logic and native UI**
  - You have an existing Kotlin project, and wish to add a second target. Could also be useful if you are building “new” for multiple platforms but need native UI i.e., performance or feature concerns.
  - Build a KMP project that includes both targets, and integrate native UI calls for each platform.
- **Option 3: Kotlin KMP + Compose**
  - Everything is shared!
  - Build a single KMP project with Compose Multiplatform.

Compose Multiplatform

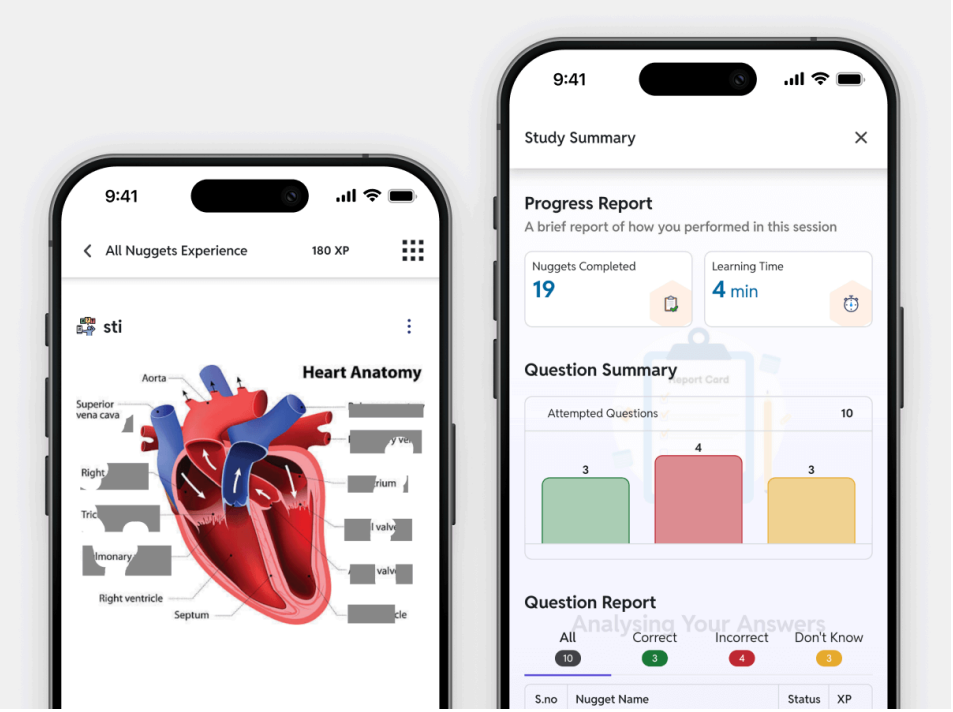


Figure 4: Physics Wallah (10M+ downloads) shares logic and UI code across iOS and Android, and exhibits near-native performance on both platforms.

Supported Platforms

Platform	KMP Stability	Compose Stability
Android	Stable	Stable
iOS	Stable	Stable
Desktop (JVM)	Stable	Stable
Server (JVM)	Stable	n/a
Web Kotlin/WASM	Beta	Beta
Web Kotlin/JS	Stable	Beta

Legend

- Beta: “It’s almost done, user feedback is especially important now.”
- Stable: “It’s done. We will be evolving it.”

> Android, iOS and Desktop are stable and usable targets for your project.

KMP & Compose Projects

Creating a KMP + Compose Project

You can create a project directly in the CLI.

```
$ mkdir project && cd project
$ kotlin init

Select a project template:

  ❯ Compose Multiplatform
      A KMP project with Android, iOS, and JVM desktop
      applications sharing UI with Compose Multiplatform
```

Tip

You can achieve the same result by creating your project in IntelliJ IDEA. Select File → New → Project → Kotlin, and choose the Compose Multiplatform template.

KMP & Compose Projects

```
├── android-app
│   ├── module.yaml
│   └── src
├── ios-app
│   ├── module.xcodeproj
│   ├── module.yaml
│   └── src
├── jvm-app
│   ├── module.yaml
│   └── src
├── kotlin
├── kotlin.bat
├── project.yaml
└── shared
    ├── module.yaml
    ├── src
    ├── src@android
    ├── src@ios
    ├── src@jvm
```

Let's break down the structure:

- `kotlin` & `kotlin.bat` are scripts.
- `project.yaml` is a top-level configuration file that describes the project contents.
- `android-app`, `ios-app` and `jvm-app` are modules that contain platform-specific code. Shared code is imported to build an executable. Very little code should reside here.
- `shared/` contains code that is shared across all platforms. Exported as a library for the other modules. This is where *most* code should reside.

KMP & Compose Projects

The top-level `project.yaml` file contains common configuration for the entire project. In this case, just a list of the modules to import.

`project.yaml`

```yaml
modules:
  - android-app
  - ios-app
  - jvm-app
  - shared
```

Each module corresponds to a top-level subdirectory. Note that each module (including `shared`) has it's own `module.yaml` configuration file.

KMP & Compose Projects

Each of the native modules contains code related to packaging and building for that platform. The rest of the application code is in `shared`.

```
jvm-app
├── module.yaml
└── src
    └── main.kt
```

module.yaml

```yaml
product: jvm/app

dependencies:
  - ../shared
  - $compose.desktop.currentOs

settings:
  compose: enabled
```

```
android-app
├── module.yaml
└── src
    ├── AndroidManifest.xml
    └── MainActivity.kt
```

module.yaml

```yaml
product: android/app

dependencies:
  - ../shared
  - androidx.activity:activity-compose:1.7.2

settings:
  compose: enabled
  junit: junit-4
```

KMP & Compose Projects

Running a Project

To run one of your projects, you can do one of the following:

Option 1: Click `Play` on the toolbar.

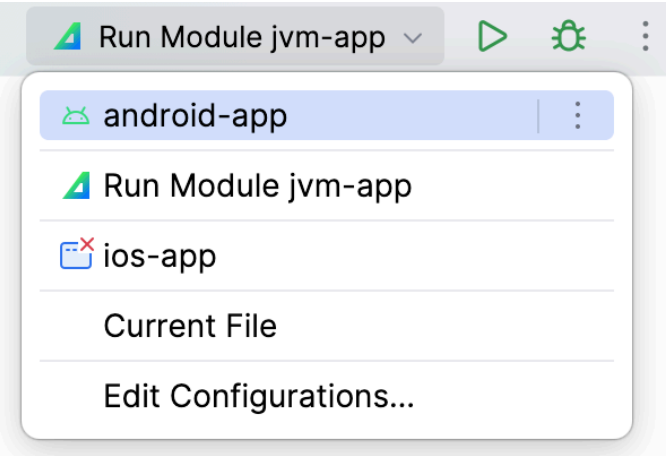


Figure 5: All configured modules will be shown, even those with build errors

KMP & Compose Projects

Option 2: Click `Play` from a module file.

Each module has it's own `module.yaml` file.

- Click the Play button beside the platform type to see options.

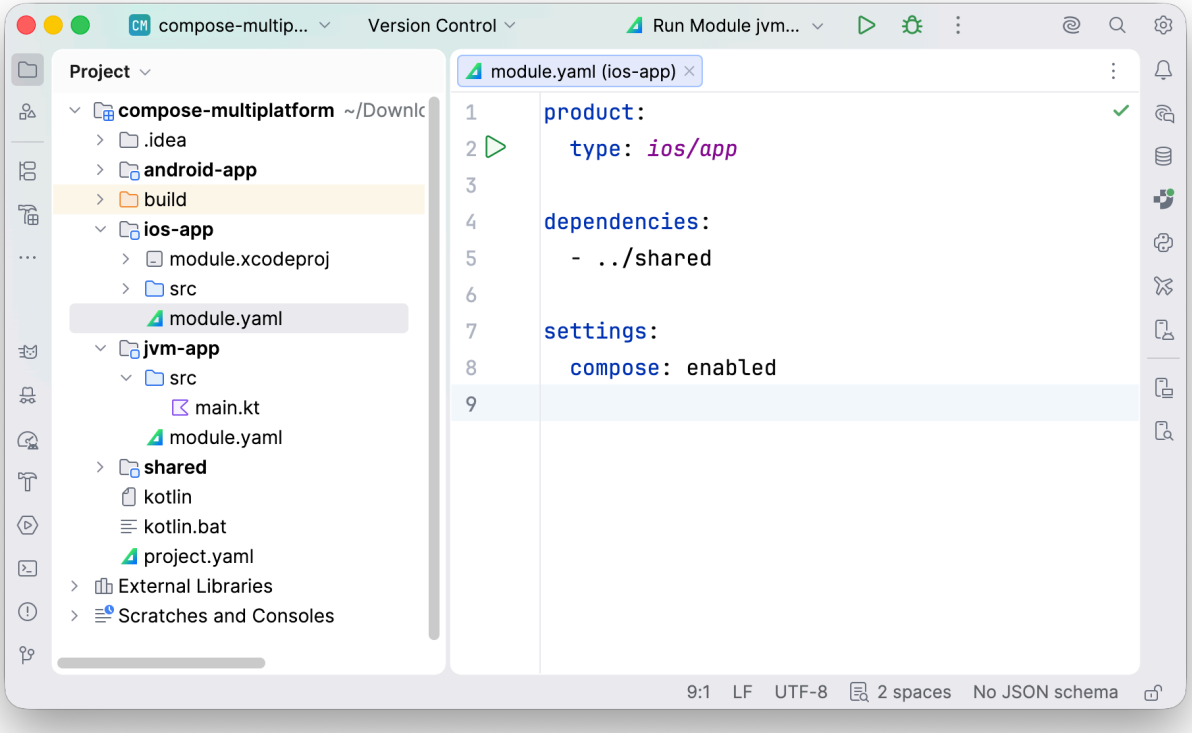
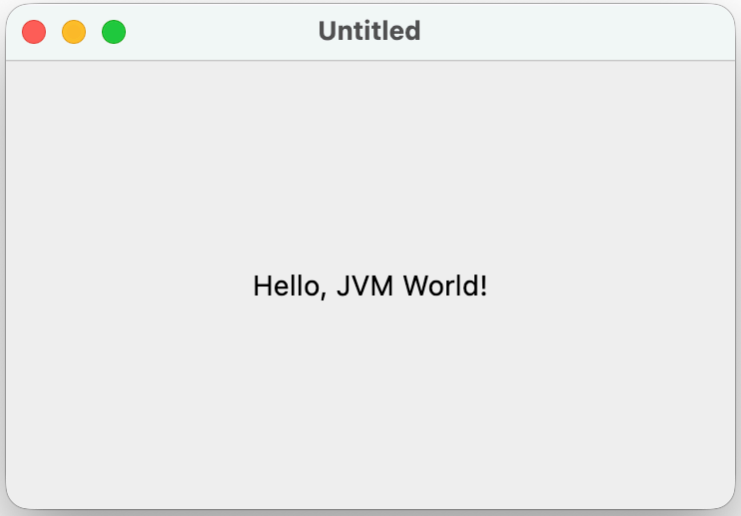


Figure 6: Modules may have different options, based on the platform.

Example: Hello World!



Let's look at the generated example[1].

JVM

We'll start with the `jvm-app` version.
We only have one file under jvm-app.

`/jvm-app/src/main.kt`

```
fun main() = application {
    Window(onCloseRequest = ::exitApplication) {
        Screen()
    }
}
```

This is a standard Compose Desktop application entry point. It's calling `Screen()` which is a composable that emits the UI.

[1] It's just `Hello World`, but it demonstrates the project structure nicely!

Example: Hello World!

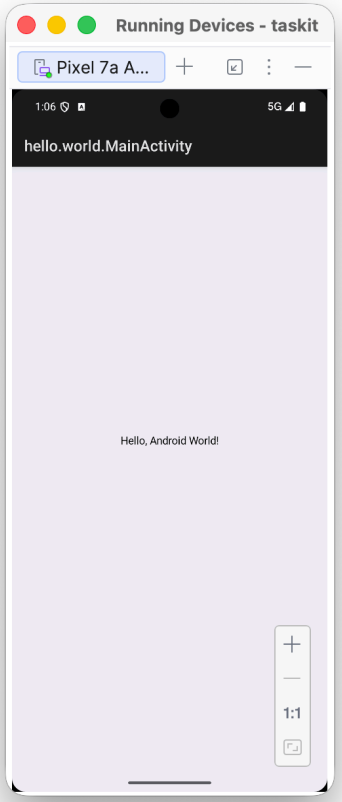
/shared/src/Screen.kt

```
@Composable
fun Screen() {
    MaterialTheme {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally,
        ) {
            BasicText("Hello, ${getWorld()}!") // calling a function
        }
    }
}
```

/shared/src/World.kt

```
expect fun getWorld(): String // "expect" suggests a native function
```

Example: Hello World!



Android

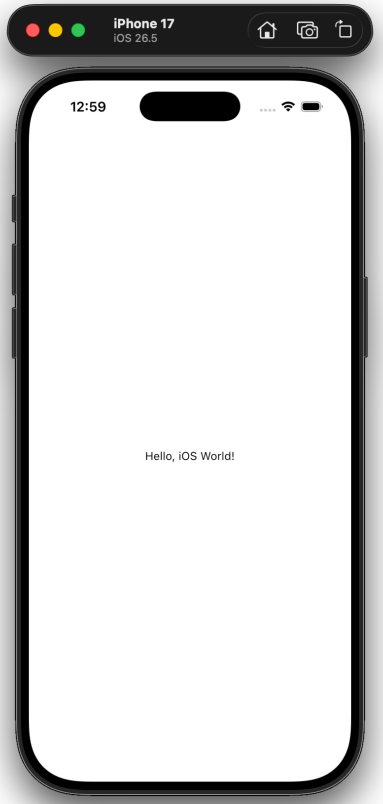
Let's see how the Android verson is setup:

```
/android/src/AndroidManifest.xml
/android/src/MainActivity.kt
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstance: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Screen()
        }
    }
}
```

This is a standard Android application entry point. It's calling the same `Screen()` function.

Example: Hello World!



iOS

Let's look at the iOS version.

```
/ios-app/module.xcodeproj
/ios-app/src/iosApp.swift
/ios-app/src/Info.plist

/ios-app/src/ViewController.kt

fun ViewController() =
  ComposeUIViewController { Screen() }
```

What's going on here?

- A `ViewController` on iOS is akin to a MainActivity on Android.
- Kotlin builds the the `ios-app` code shared code into an Swift library that can be imported into an iOS project.
- The `module.xcodeproj` is a skeleton project that we use to import and build our application. This is necessary because we need to the Apple toolchain to compile apps for iOS.

Shared vs. Native Code

Motivation

There are times when you will need to make native calls to the underlying platform. Imagine that you have a platform specific library for instance, that only works with Android or desktop.

This isn't that unusual. You often run into functionality that differs based on platform.

- **Hardware & System Info**: Device-specific identifiers, battery levels, or OS versions (e.g., reading `android.os.Build.MODEL` on Android and `UIDevice.currentDevice.systemName` on iOS).
- **File System Operations**: Local device storage since directory paths differ fundamentally (e.g., `Context.getFilesDir()` on Android vs. `NSDocumentDirectory` on iOS).
- **Cryptography & Security**: Hardware-backed keystores or secure enclaves that require platform-native crypto libraries.

Expect/Actual

How do you manage this seamlessly in your code?

- `expect` and `actual` function declarations let you declare a common API, and then fulfil that in a different way for each platform.

We saw this in our earlier `Hello World` example, where we had a shared `expect` declaration and one `actual` definition for each platform.

```
/shared/src/World.kt:
  expect fun getWorld(): String

/shared/src@jvm/World.kt:
  actual fun getWorld() = "JVM World"

/shared/src@ios/World.kt:
  actual fun getWorld() = "iOS World"

/shared/src@android/World.kt:
  actual fun getWorld() = "Android World"
```

Expect/Actual

Rules for expected and actual declarations

To define expected and actual declarations, follow these rules [2]:

1. In the common source set, declare a standard Kotlin construct. This can be a function, property, class, interface, enumeration, or annotation.

2. Mark this construct with the `expect` keyword. This is your **expected declaration**. These declarations can be used in the common code, but shouldn't include any implementation. Instead, the platform-specific code provides this implementation.

3. In each platform-specific source, declare the same construct in the same package and mark it with the `actual` keyword. This is your **actual declaration**, containing implementation using platform-specific libraries.

> **Tip**
>
> Avoid the need to do this entirely by using KMP cross-platform libraries.

Bibliography

[1] JetBrains Inc., “Getting Started with Kotlin Multiplatform.” [Online]. Available: https://kotlinlang.org/docs/multiplatform/get-started.html

[2] JetBrains Inc., “Kotlin Documentation.” [Online]. Available: https://kotlinlang.org/