Kotlin Multiplatform

Alexander Takács

Brief overview of the presentation topic

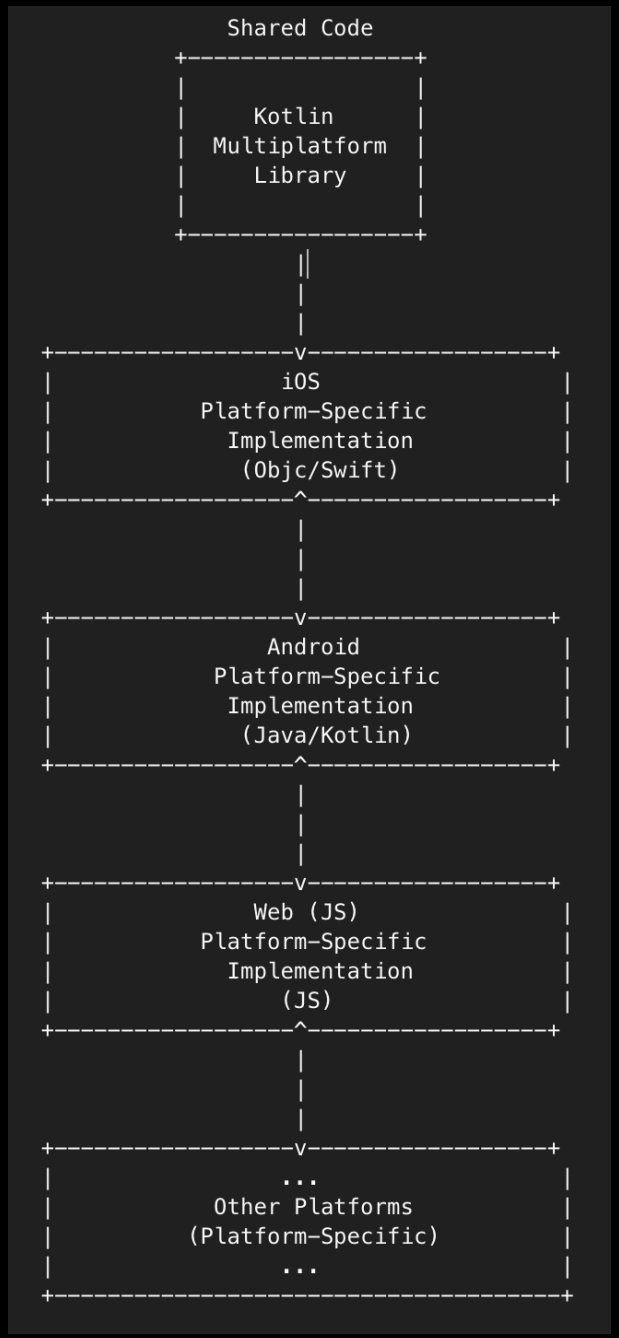
- definition of Kotlin Multiplatform
- comparison of Kotlin Multiplatform with some other popular cross-platform technologies
- pre-requisites for using Kotlin Multiplatform
- overview of Kotlin Multiplatform in the project
  - platform-specific code
  - common API
  - platform-specific API
  - compiling and running

Brief overview of the presentation topic

- dependency management and package management
- sharing resources
- sharing view models?
- third party native libraries?
- benefits and drawbacks

What is Kotlin Multiplatform?

- technology that enables developers to write code in Kotlin and share it across multiple platforms: Android, iOS, macOS, Windows, Linux, web ...
- share common business logic, data models, and other code
- ability to write platform-specific code
- providing platform-specific libraries and APIs
- provides a unified build system to compile and package the shared code for each platform

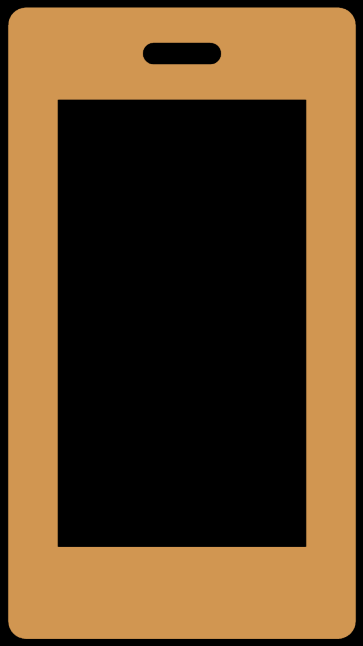

Shared Code
Kotlin
Multiplatform
Library
iOS
Platform-Specific
Implementation
(Objc/Swift)
Android
Platform-Specific
Implementation
(Java/Kotlin)
Web (JS)
Platform-Specific
Implementation
(JS)
...
Other Platforms
(Platform-Specific)
...

Comparison with some other cross-platform technologies

- focused on sharing business logic
- no UI support (at least for mobile development)
- flexibility in choosing the appropriate UI toolkit for each platform
- different compiler
- performance
- full native app as a result

Pre-requisites for using Kotlin Multiplatform

- knowledge of Kotlin
- familiarity with a mobile platform
- familiarity with a build system (Gradle, Maven, or Xcode)
- development environment
- familiarity with platform-specific APIs
- understanding of platform-specific constraints



Code preview

https://github.com/cngroupdk/MobileMeetups/tree/feature/kmm/pokedex

Dependency management and package management

1. Dependency Management
   - managed through Gradle
   - dependencies for each target platform in their build.gradle file

```
kotlin {
    ios {
        binaries {
            framework {
                baseName = "MyLib"
            }
        }
    }
    android {
        dependencies {
            implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
        }
    }
}
```

Dependency management and package management

2. Package Management
   - managed through Gradle
   - provides support for publishing and consuming packages through the Maven repository format
3. Native dependency management in target platform

Sharing resources (moko-resources)

- Strings
- Plurals
- Images
- Fonts
- Files
- Colors with light/dark mode support
- StringDesc

How to share view models?

- needs to setup expect parent object in common code
- needs to define common coroutine scope
- Android:
  - inherit from lifecycle.ViewModel
  - pass reference to viewModelScope
- iOS:
  - create custom coroutine scope
  - resolve coroutine scope cancellation on deinit
  - map coroutine flows into SwiftUI state properties

Sharing view models

- Pros:
  - code sharing
  - testability
  - a benefits of having a cross-platform developer team

- Cons:
  - synchronization when the UI is platform specific
  - can slow down the UI development (UI development team need to wait for the shared library)
  - a disadvantage when the team is split into platform-specific developers

Third party native libraries?

- Android:
  - need to be implemented in android platform specific code within the shared module, otherwise the same as with native development
- iOS:
  - create an interface for the iOS native library in Kotlin code
  - write an implementation of the interface using Objective-C in Xcode
  - use the Kotlin Native interop mechanism to link the Objective-C implementation to your Kotlin code

KMP benefits

- code sharing: reduce code duplication, improve code maintainability, and increase developer productivity
- improved performance: compiled natively for each platform
- platform-specific functionality: allows write platform-specific code when needed
- strong typing: Kotlin is a statically typed language
- community support

KMP drawbacks

- learning curve: relatively new technology, and it may take some time for developers to learn the platform-specific APIs and libraries
- build and deploy complexity: especially when targeting multiple platforms
- debugging challenges: especially when dealing with platform-specific code
- versioning: ensure that changes made to the shared code do not affect other platforms

Thank you

Any questions?