MOBILE APPLICATION DEVELOPMENT

ANDROID (2017)

LECTURE 24: NDK (PART 3) AND IOS

MULTI-PLATFORM PROJECTS

- Writing native code and limiting it to only one platform that wraps it is not very useful (generally) - a primary benefit of native code is its ability to be portable.
- Using native code across platforms means using it in multiple projects. This raises a number of issues that are not present in 'normal' projects:
    - How can the code be structured efficiently into repositories / projects?
    - Who owns each component of the codebase? How do they agree on changes?
    - How does versioning for the projects work? How are native updates propagated?
- There is no correct answer on how to correctly structure such projects!

ONE SOLUTION TO MULTI-PLATFORM PROJECT ORGANIZATION

- A relatively good structure for multi-platform projects is to make the components of those projects as self-contained and modular as possible.
- Example:
  - The native code itself is a self-contained library in a language like C.
  - The native code has wrappers on each target platform, which are each separate and self-contained projects that reference the native code project.
  - The applications which consume the wrapped native code import the wrapper projects as dependencies.
- This allows components to generally be worked on and updated in isolation.

IOS

- iOS is Apple's mobile platform, and the primary competitor to Android worldwide. (Unless you consider different 'flavors' of Android to be competing...)
- iOS development has a large number of similarities to Android development, particularly since Kotlin's introduction.
  - Expressive, modern languages built with attention to nullability and safety.
  - Massive, well-designed APIs for building apps.
  - Similar architecture, such as collection adapters, View trees, and listeners.
- You can use your knowledge from this course to understand iOS code quickly.

SWIFT VS KOTLIN

- Swift and Kotlin are similar in many ways - for a few examples, compare:
  - Support for mutable and non-mutable types (Swift uses `let` and `var` while Kotlin uses `val` and `var` to denote immutable and mutable values, respectively).
  - Support for nullable and non-nullable types - the same notation is used in both languages (Type vs Type?).
  - First-class functions, meaning functions can be passed to and returned from other functions - closures and lambdas are also supported.
  - Extensive support for algebraic data types via enum classes.

SWIFT VS KOTLIN

- Swift and Kotlin also have a number of differences - for a few examples, compare:
  - Swift is a compiled language, whereas Kotlin has support for being either compiled or interpreted (interpreted in the case of the JVM).
  - Kotlin types are `final` by default - Swift requires manually declaring this.
  - Swift has support for traditional `static` functions and properties within classes.
  - Swift is much more cleanly-compatible with C than Kotlin is, and can natively understand a variety of C classes.
  - Swift is not available for Windows by default, whereas Kotlin is - via the JVM.

ANDROID VS IOS - VIEW HIERARCHY

- Android and iOS have a number of differences in the way their View classes are organized, even though the general principles are similar.
  - Where Android uses Activities to manage content Views and Fragments to subdivide UI, iOS uses ViewControllers to manage Views and NavigationControllers (among others) to manage ViewControllers.
  - There is not a direct equivalent to the measurement process from Android on iOS - iOS Views can directly contain other Views and are expected to dictate measurements or constraints to their child Views.
- Despite these differences, both systems use a tree structure to organize UI and a number of concepts, such as layering Views, are identical on both systems.

ANDROID VS IOS - DEVELOPMENT

- Both Android and iOS have first-party IDEs made by the company that makes the OS, along with sets of best practices and guidelines for developers.
- In general, Android's guidelines and system restrictions are more permissive than iOS, but also less clearly defined and predictable.
- iOS developers have the ability to test all currently-supported devices with their app through their IDE - Android developers cannot reasonably test all devices.
- The barrier to entry in the Android 'app store' is much lower than the barrier to entry in the iOS App Store.
- Apple regularly exercises more control over their store than Google does (good / bad).

GUIDELINES FOR BEING A MULTI-PLATFORM DEVELOPER

- Be flexible in how you approach solving problems.
- Do not learn specific libraries and languages obsessively - learn techniques.
- Don't be a single 'type' of developer.
- Use every platform you develop for as a 'power user'.
- Read platform design guidelines, and understand when to break them.
- Be an early adopter.
- Be familiar with old versions of your platforms as well as the new versions.