Ser423 Mobile Systems

Unit 1. Introduction to iOS and Android Apps
1.a Outcomes and References

1.a.1 What should you get from this section?

• **Outcomes**
  
  - To know and be able to explain the differences between desktop programs and iOS an Android apps. Their execution model, their components, and how they are built.
  
  - To be able to describe the differences between building and developing iOS and Android apps. **Cross-Compiling** versus **Just In-Time Compiling** with **Byte Code Interpretation**.

• **References**

  • **Android** reading in: *Busy Coders Guide to Android* --> Section on: **Key Android Concepts**.
  
  • **Android** Reading: *Android App Components*
  
  • **Android** Reading: *Intents and Filters*
  
  • **iOS** Reference: *App Programming Guide*
  
  • **iOS** Reference: *App Build Process*
1.b Mobile App Components by Design

1.b.1 Model, View, and Control - Design Pattern

By Totallybradical - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=56162327
1.b.2 What is MVC?

- **Model-View-Control (MVC)** is a software design pattern, which is especially applicable to mobile apps.

  - Nearly all of the components (classes, databases, resources) of a mobile app (Android and iOS) can be categorized as part of the Model, View, or Control aspects. Data sources, delegates, and notifications complicate the diagram, above, but MVC is prevalent in mobile app design.

  - **Model**: Characterizes the application’s behavior in terms of the problem domain, and is antipodean of the user interface. It constitutes the Data and underlying rules for its representation.

  - **View**: Consists of what the user of a system sees. In a mobile app, the view is the presentation layer consisting of the buttons, lists, images, and other objects that are displayed to the user.

  - **Control**: These components mediate between the Model and the View. For example, when the user clicks a button or selects a list item, the controller receives notice of that action and uses or modifies the underlying model according to the request and may cause the view to change based on the application domain logic.
1.c Android Components and Build

1.c.1 Android Components -- Primary Classes

- **Activity** - Represents the controller for a single screen (View) of an App’s user interface. Its implemented in one of the subclasses of `Activity` (and/or Fragment). Similar to a window, web page, or a dialog in a desktop application. Activities are short-lived and are the building blocks of an App.

- **Service** - Runs in the background to perform long-running work for remote or local processes. Execution is independent of Activities, and may be used for longer-term functionality, such as scheduled tasks, checking a remote server, or to play back music or other audio.

- **Content Provider** - Manages a shared set of app information (model) that may be stored in files, a SQLite database, in the Web, or provided by any data-service. Can be used for private data, or for inter-app sharing.

- **Broadcast Receivers** - responds to system-wide announcements or **Intents**.

- **Intents** - A messaging object for requesting an action from another app component. A data bearing request for an action within or outside the app.
1.c.2 Non-Code Components of an Android App

• **Resources** - **images**, **xml layouts** for views and menus, and specific values: **string** definitions, **colors**, **styles** and **dimensions**. These are defined to provide an efficient runtime profile both in terms of space and time.
  - These are all nested within the **res** folder of the **Android Studio** project.
  - All views are stored in **XML** layout files: Essentially a **serialized** format. The designer directly produces the layouts, and when the app is run it de-serializes its view (UI) from the layouts as needed.

• **AndroidManifest.xml** - The manifest file collects everything the Android system must know about the application. It includes:
  - Definitions of all app components (Activities, Content Providers, Services) that can be started by the App or another communicating App.
  - ** Declares the Permissions** needed by the App. Access to the network, to other apps, such as **Contacts**, access to on-board resources, such as location services, accelerometer, and camera.
  - Declares the **minimum Android API level** required for the App. Defines any **hardware** and **software** features used by the App. Defines **External API** features that are used by the App, which must be linked to at runtime. Such as Android services like Ad or mapping.
1.c.3 Android App Build Process

![Diagram of Android App Build Process]

Legend:
- Your Source
- External Inputs
- Tools
- Generated Source
- Binary Outputs

Mobile Systems © T. Lindquist 2017
1.c.4 Android Build and Platform Tools

- **Build** - Command line tools that are provided as part of the **SDK**. Some are used by the **build process**, others are available to the **Developer** and/or **IDE**.
  
  - **aapt** - **Android App Packaging Tool**. Processes the Resource files, and generates the **R.java** (Resources) class which is used to access resources at runtime by developers and system code. Coalesces the non-Java resources into proper format for the app bundle (jar archive).
  
  - **apkbuilder** - creates the bundle (compressed archive) that is the app.
  
  - **zipalign** - Processes the bundle (.apk file) to align all un-compressed data relative to the beginning of the file. This provides access to images or other raw files by aligning them on 4-byte boundaries.
  
  - **aidl** - **Android Interface Definition Language**. To do interprocess communication you define an AIDL interface. This tools processes it.

- **Platform**
  
  - **adb** - **Android debug bridge**. To start a device or emulator shell:
    
    `adb -e shell`
  
  - **Android Monitor**. See device or emulator’s log, memory, CPU, GPU, and network usage.
1.d  iOS App Components and Build Process

1.d.1  iOS Key Classes and Objects

- **UIApplication object** - Is the basis for App scheduling, managing foreground/background, App creation and destruction. It manages the event loop and other app behaviors, such as push notifications.

- **App Delegate object** - Is the focus of an App’s code, handling initialization, state transitions (pause, resume), high-level app events, and **Model** setup.

- **Documents and Data Model** - The underlying data model for an App. May be files, and/or a database. **Sqlite3**, and **Core Data** are directly supported.
  
  - Apps can use document objects, which are a custom subclasses of **UIDocument** which provides specific services for managing content, including using **iCloud**.
  
  - See: [Document-based App Programming Guide](#)

- **Views** and **View Controllers** - **UIViewController** is the base class for view controllers, providing default functionality for loading, and rotating Views. UI components (buttons, text fields, pickers, lists, navigation bar) are part of the **View. Data Sources** for composite view objects (such as pickers, and lists/table view) are programmed through specific view controller classes.
1.d.2 iOS Build Process Steps

- **Swift** and **Objective-C** are compiled to object code, not to bytecode, as done for Android. Thus, they both directly run on the mobile device.
  - You cannot take a compiled and linked Swift or Objective-C program from one machine to another type of machine and execute it without recompiling. Unless, its complied using a cross-compiler targeting the new machine. This is an advantage of Java’s bytecode.
  - See: [Wikipedia on Cross Compiling](https://en.wikipedia.org/wiki/Cross-compilation)

- Xcode Build Steps to generate iOS App. See Report Navigator Build
  - **Cross-Compile** dependent code if necessary -- to ARM object code
  - **Cross-Compile** your App’s Swift or Obj-C code to ARM object code
  - Copy necessary libraries
  - **Link;** targeting ARM/iOS with linker that runs on OS X
  - Compile storyboard files, and asset catalogs; process **Info.plist**
  - Link storyboards using **Interface Builder** tool **ibtool**
  - Copy **Swift standard libraries** into the app
  - Use **Touch** to update mod time. (Signing and others occur with deploy)
1.e Why Bytecode for Java/Android?

1.e.1 Compare the Output of Compiling for iOS with Android

• Android Runs primarily on ARM-Based processors, but not exclusively.

• For example, suppose source code input is something like the following:
  \[ \text{Position} = \text{Initial} + \text{Rate} \times 60.0; \]

• The compiler’s Parser may produce a syntax tree of this code, such as:

```
  =
  \text{Id}_1
  +
  \text{Id}_2
  \times
  \text{Id}_3
  \times
  60.0
```

• The compiler output for a Java program with this statement will be part of a stack-based bytecode in a `.class` file. For this statement it may resemble:

```
dload 60.0  
(dpush the constant 60.0 (dload is for double (32 bits) where iload is int

dload_0
(push double Rate which is in local storage and indexed from a base)

dmultiply
(pops two factors (2 words each), double multiply, and push the result)

dload_2
(same as for Rate, Initial is an index relative to a base, here 3)

dadd
(remove and add the top two float elements; push the result)

dstore_4
(ditto above, no source-level variable identifiers are used in bytecode)
```
1.e.2  Compilation Comparison and Execution -- Android

- **Android Studio** uses “any” java compiler (such as `jdk javac`) and produces the same output as if the program were to be run on a desktop (bytecode).

- The **bytecode** format is a stack-based intermediate. Intermediate means: between java source code and object code (binary machine code).

- Bytecode is used by Java runtime environments (the jre, or the java command) for desktop systems (MacOS, Windows, Linux) executes this stack-based bytecode.

- Java runtimes often use what is called just-in-time compiling (JIT).

- The Android build process converts the stack-based bytecode to a register-based bytecode format (Dalvik, or `.dex` file extension).

- **Dalvik** Runtime. Prior to Android version 4.4 (KitKat) a Dalvik runtime executed this register-based bytecode on the device using JIT approach.

- **Android Runtime** (ART) has since KitKat, replaced the Dalvik Runtime. The Dalvik (.dex) bytecode is still produced in the build process by Android Studio, but when its installed onto a device, its converted to machine code, using the `dex2oat` utility running on the Android device.
1.e.3 Comparing Dalvik with Android Runtime

by MjolnirPants
1.e.4 iOS Compiler Output

- Apple has used ARM-based processors in its devices running iOS. ARM stands for Advanced RISC Machines.

- For iOS a more traditional approach is taken: The same statement would be compiled into object code for iOS (not MacOS/x86) to run on the ARM processor architecture. The object code generated for the statement may be an assembled version (binary machine code) something similar to the assembly language instructions below:

  movf   Id3, R2
  mulf   #60.0, R2
  movf   Id2, R1
  addf   R2, R1
  movf   R1, Id1

- The Swift compiler that is executed by Xcode, is a cross-compiler since its running on a MacOS/X86 machine and is generating object code to run on iOS/Arm. That is, although compiled on MacOS, this code will only run on MacOS under the Simulator, and not natively.
1.e.5 Languages and Their Compilers

• Languages commonly compiled directly into Object Code.

  - These include languages such as: C, C++, Objective-C, Pascal, Ada, Cobol, Fortran, Swift, Go. The GNU Java compiler has the option to produce object code. The compiler creates machine language, which is operating system and processor specific. The linker packages the resulting code so that it can be directly executed on the machine.

• Another class of languages (compiled to bytecode) stop at the intermediate code phase. Most common examples include Java, C#, and Python.

  - Java uses bytecode. C# uses Common Intermediate Language (CIL).

  - The Java command is an interpreter for bytecode, and for C#, the Common Language Runtime (CLR) provides execution support by directly interpreting the intermediate form.

• Byte-code compiled languages commonly use Just-In-Time (JIT) compiling. JIT converts intermediate code into machine code at execution time. Often, only parts of the program are JIT compiled to native code. The runtime keeps execution statistics to determine when/what to JIT compile.