CS 6018: Application Systems Design
AKA Android Programming

Fall 2024

Lecture 1

Intro, High level stuff, hello world

Course overview

- This course covers recent “best practice” design/implementation of apps on Android platform
- The problems we'll solve are not unique to Android, and the solutions we'll discuss appear in other platforms (SwiftUI, React, etc)
- Generally we'll “hack” at a problem to get a solution, analyze what we come up with to understand it's downsides and how to avoid/eliminate them
- Our naive solutions might be OK for small, single-person apps, but don't scale to bigger apps or teams

Main “problems”

- Managing “state” — data that changes over time, modified by user actions or other events
- Event driven programming — “Life does not start and stop at your convenience”
- Persistence — Where/how should data be stored? How do you manage updates?
- Asynchrony — How do we manage work we can do “in the background”?
- Reusability — Write once, use many times
- UI/UX — We're writing software aimed at nontechnical users

Course Structure

- Lectures Tuesday, Wednesday
- Labs Thursday — work on a small app in class
  - These are graded, individual work
  - You'll practice using tools that will help on the big project
- Course project: drawing app
  - MSPaint-like app for the phone
  - Will showcase lots of Android libs/techniques/capabilities
  - Teams of 3, 4 “phases”
  - Some phases add new features, other refactor to improve/simplify code

Getting help

TAs

- Aparna Gudivada
- Alex Yang

LLM Policy

- First time I've written one of these, we'll see how it works
- At this point you all “know how to program” and we want you to get “real world” experience
- You can use Copilot (LLM powered autocomplete) in this course **as long as commits that contain copiloted code describe what code it helped with, and you include your “reflections” on the experience of using it**
- Let me know how this policy is working and we can adjust if necessary

Android

- Platform mostly run by Google (Alphabet?), but lots of it is open source
- Majority of worldwide mobile market, but not the US anymore (as of fall 2022)
- Development originally used Java (Sort of, see this lawsuit), now Kotlin encouraged
- On version 34, lots of changes and new features added over the last 15 years!

Kotlin

- Language developed by JetBrains (the IntelliJ people)
- Compiles to JVM bytecode, just like Java
- Mostly a “nicer” Java which lets you write less, ore expressive code
- I'll highlight some differences, but we won't spend a lot of time talking about Kotlin in a lot of depth

Android Studio

- This is the IDE people (everyone?) use for writing Android apps.
  It's developed by (or a slight modification of) Jetbrains, so it should be familiar if you used IntelliJ or PyCharm or CLion, etc.
- There's a bunch of extra stuff compared to IntelliJ
  - Graphical UI designer tool — drag + drop editor for Android layouts
  - Device emulator — run your code on a simulated phone
  - Other testing tools, etc
- Get it from here or run `brew install --cask android-studio`

Programming a UI

- If you had time and $$ to write a UI system for developers to use, what would it look like?
- Maybe you'd use an OOP approach, and you'd have a base class for “UI Elements” that Buttons, text fields, images, etc extend
- You create objects, and have methods that take callbacks to respond to events (clicks) or to change their state (`setText`, etc)
- What are the pros/cons of this design?

Android UI

- How does Android do it?
- We'll start by discussing “view based UIs” and will see a different approach later in the semester
- This is basically the OOP idea we discussed, but with some polish
- View based apps rely on 3 families of classes

Activities

- An activity corresponds to a “screen”
- An activity is defined in 2 files: a .xml file that describes the UI elements on that screen, and a kotlin (or Java) file that handles the behavior of that screen
- You can use the UI designer to drag/drop components in your activity layout and/or edit the xml file by hand. I often do a mix of these
- The corresponding Kotlin file is where any code you write to determine the functionality of the activity goes
- In the old days, Activities were the only place to put code like this, but things have changed...

Views

- a View is a UI element like a text box, button, image displayer, etc
- Some views are “containers” or “layouts” which can contain children views, such as “lists” or “grids” of other views
- Views should generally be “dumb” and not have much code associated with them

Fragments

- A Fragment is a piece of a UI which is reusable and could theoretically appear in multiple places in an app
- Like an activity, a Fragment has an XML file and a kotlin file
- An activity can contain fragments
- fragments can contain fragments
- fragments and activities can contain Views

Building a “modern” “view based” app

- One activity, many fragments
- For a few reasons we'll discuss later, creating a single Activity and having a Fragment for each “Screen” in the app is the current recommendation
- The “screen” fragments can/should contain fragments like “fragment showing a user avatar + username” or “fragment showing a list of all users” which might appear on many different screens
- The smaller fragments are made up of multiple fragments and views
- For some simple examples, we'll just put views in an activity, but remember that's just for simple examples

Details

- Views have a bunch of attributes which can (mostly) be modified at runtime with setter methods
- We can also specify them at build time by specifying them in the XML file (which can be nicely edited using the GUI editor)
- At runtime the XML file is “inflated” to create objects

Layouts

- If we want to have many Views in an activity/fragment, we need a way to organize them and control how they are positioned
- These are called Layout Views in Android. There are several of these which organize their children in various ways
- My advice for laying out stuff is combine simple layouts (horizontal/vertical layouts, maybe grids) and use nesting to build more complex designe
- Apparently deep nesting was very slow on Android for some reason (I honestly can't imagine why), so they made a “super layout” for doing complex layouts in “one shot”

Common Layouts

- FrameLayout — Not really a layout. Children are drawn on top of each other in the order they're added
- LinearLayout — Lay children in order, either horizontally or vertically
- TableLayout — Lays elements in rows and columns
- ConstraintLayout — Do complex layout with many children with 1 level of hierarchy
  - With constraint layouts you specify constraints (relative placement of child views) either in the GUI designer (nice if it works), or by editing the XML (if it doesn't)

Useful attributes

- Some of these can be controlled easily via the GUI editor, some I more often add to XML myself
- width and height — common values are match_parent (fill all available space) or wrap_content (as small as possible)
- padding (inside) and margin(outside) control spacing around elements. You can adjust these individually for top/bottom/left/right or all at once
- gravity — centering/alignment
- Many attributes that deal with sizes work with several different units. Display pixels(dp) seems to be the recommended units to use

Stuff we'll see later

- “responsive views” — easier on Android since you can't resize windows... just supporting phones/tables in portrait/landscape modes
- Interaction — clicking buttons, typing text, other taps/drags/etc
- Navigating between screens
- Much more