More Jetpack Compose

Layout/Sizing: Another Hard Problem

- Given a UI tree (a tree of elements to display) how big should each element be drawn, and where should they be placed?
- Some elements have a simple given size (ie, an image), but others depend on the size/positions of other elements (a `Row` with children)
- Today we'll talk discuss how Compose solves this problem in an efficient/understandable way

Compose drawing algorithm

1. Composition: Starting at the root composable function, traverse the "call tree" using the current values of any state (`if(myState){ Text() } else { Button() }` for example). Here, Text will be composed, but the Button won't be if myState is currently true
2. Measurement and Layout
   - Measure
   - Place
3. Drawing

Measurement

- How big should a column be? It depends... What's in it?
- How much space can it occupy? If there's stuff to the left/right of it, it can't fill the whole screen
- Basically, for each element, we call a method `measure(SizeConstraints)-> Measurement`
- The parent passes constraints to the children which affect how they're measured
- Then it computes its size based on its children, and decides where to place them
- For the most part this is a top-down process although there are a couple of ways to skirt that

Modifiers

- By default, the constraints that get passed to measure are very permissive, basically 0-width and 0-height of the device
- The parent will pass its constraints down to its children when measuring them
- Modifiers can adjust the constraints before they're passed to children. For example, the padding modifier will be subtracted from the max width/height constraints of the child
- Modifiers like "`fillMaxWidth()`" will set the min width constraint of a child to be the maxWidth of the parent, etc

App layout: Scaffold class

- We've seen how to layout individual parts of the app with Compose, including using Modifiers to size/place components
- For the overall layout of a “Screen” it might be useful to use the provided Scaffold composable
- It has “slots” for top and bottom “app bars”, a main “content” section and “floating action buttons”
- A FAB is a button that sits on top of other elements, often used for stuff like “compose a new email”

App bars

- From Google:
  - Top app bar: “Provides access to key tasks and information. Generally, hosts a title, core action items, and certain navigation items.”
  - Bottom app bar: “Typically includes core navigation items. May also provide access to other key actions, such as through a contained floating action button.”

Adaptive Layouts

- Android recommendations are to use explicit control flow (if statements/switch/case) to determine an appropriate layout for the current available size
- Since tablets/foldable devices might not give you access to the full screen all the time, they suggest NOT querying the hardware capabilities, which could be an overestimate of what's available
- `androidx.compose.material3.windowsizeclass.calculateWindowSizeClass` returns a “size class” for the width and height. The classes can be “Compact,” “Medium,” or “Expanded”
- That method is a “Composable” so it will be “recomposed” (called again) when the window size changes (ie, screen is rotated, etc)

Animation

- This is a HUGE topic, so we'll just scratch the surface
- There are typically many different ways to do stuff, and the performance of them can vary greatly if they do work in different parts of the drawing process (most expensive to modify the “composition”, less expensive to modify “layout”, cheapest to modify “drawing”)
- The purpose of animation is to communicate what is happening to help the user navigate the app

Simple tools: AnimatedVisibility

- This is a composable that takes a boolen for whether the thing it wraps should be displayed
- It has a bunch of options for what animation to use when we show/hide an element

Simple Tool: animateAsState

- Remember, that when `State` objects are passed to composables, the composables get re-run when the state changes
- One way to do animation is to just have state variables keep changing
- For a bunch of “visual” properties there are a family of methods such as `animateFloatAsState` or `animateColorAsState`, etc
- Declare these outside of a compsable and use them to inside for properties (position, or size, for example) and you have animation

Neat tool: animation editor

- We've seen compose previews that basically replace the GUI layout designer that we used for View-based apps
- It also has some great tools for animation
- Most animation functions take a “label” parameter which doesn't show up in your app... it gets used by the tool
- If you hover over the preview, there's a “start animation preview” button that will play the animation
- It also lets you pause, single step, and look at the “animation curves” that are running. It's really handy!

Testing

- Composable functions can be tested pretty easily with this Composable container: `@get:Rule val composeTestRule = createComposeRule()` which you declare in an instrumented test
- This has a `setContent()` method which you can call a composable function in
- From there, it has a collection of “matchers,” “actions,” and “assertions” similar to Espresso for you to write tests with
- If you split your UI into smaller Composables, you can essentially “unit test” the various pieces in isolation
- Check the cheat sheet