Jetpack Compose

Overview

- Way back in the day, UIs were all code (`myPanel = new JPanel(); myPanel.append(new Button);) //etc`
- Maybe ~10ish years ago, lots of UI libraries decided to split GUI programming into 2 pieces:
  - Layout in something XML-like (Android XML, C# XAML, JavaFX has something like that, Qt has an XML layout)
  - Code that inflates/accesses that layout
- There's pros + cons of this
- In the last few years, many UI libs have gone back to all code, but with some upgrades

Main idea

- In these new libraries, instead of classes for each UI element, they're exposed to you as functions
- Creating a nested GUI layout is a nested function call
- The tricky part is since you no longer have direct access to a Button object or a TextView object, you can't attach event handlers
- These libraries all have various “binding” mechanisms to handle these sorts of things
- Examples of these libraries are SwiftUI (from apple), and Jetpack Compose (also, Flutter is at least superficially designed this way)

Jetpack Compose

- Create a new “View” like thing by annotating a function as `@Composable`
- When its parameters change, the layout implied by the function will be updated
- There are special `State` wrappers that let you explicitly specify variables that will be modified and should trigger updates to the UI
- There are converters for stuff like `LiveData` to make it work with JC UIs
- It's possible to mix the View based designs we've been working with an JC stuff without too much trouble

A @Composable function in detail

```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier, enabled: Boolean
    = true,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    elevation: ButtonElevation? = ButtonDefaults.elevation(), shape: Shape =
    MaterialTheme.shapes.small,
    border: BorderStroke? = null,
    colors: ButtonColors = ButtonDefaults.buttonColors(), contentPadding: PaddingValues =
    ButtonDefaults.ContentPadding, content: @Composable RowScope.() -> Unit
): Unit
```

Button analysis

- onClick is passed as a parameter to the button function (it's () -> Unit meaning no parameters, specifically no reference to the button that was clicked! No return value)
- The last parameter is content: @Composable RowScope.() -> Unit
  meaning the stuff to display inside of the button (maybe text, or an image)
  is just some other composable function!
- Most of the elements work like this, and their "content" is usually passed using Kotlin's "trailing lambda" syntax: SomeComposeable(...) { //this is a lambda containing its contents }
- Often we take advantage of Kotlin's "named parameter" syntax to accept default values for most args, and just specify the ones to customize

Modifiers

- Most compose functions take a `Modifier` parameter
- Modifiers specify stuff like size, color, event handlers, font, etc
- Typically we write them using a “chain” notation like:

```
Modifier.padding(padding)
    .clickable(onClick = onClick)
    .fillMaxWidth()
```

- Modifiers are normal values so we can store them in vars, etc to reuse them

State

- The only way to “redraw” a composable is to call the function again with new arguments. We can't get a reference to a TextView and call setText anymore...
- To store mutable data, we need to use a wrapper class that Compose can use to figure out necessary updates
- Basically, if we modify a state, we'll redraw anything composables that us it.
- This is pretty similar to modifying the value in LiveData and triggering observers to run update code

Common State apporach: remember

```
var someData by remember { mutableStateOf("") }
```

- This creates a MutableState<string>. We can pass someData to Composables that have String parameters, and they'll be “recomposed” when it changes
- For working LiveData or Flow data there are `observeAsState()` and `collectAsState()` which use observe or collect to detect updates

Simple made difficult (weird at least): EditText in Compose

```
var name by remember { mutableStateOf("") }
OutlinedTextField(
        value = name,
        onValueChange = { name = it }, label = {
        Text("Name") }
)
```

- Pretty weird... When the user types something, onValueChange is called with the next text
- We update the State field, triggering recomposition
- The text field gets redrawn and the text in it is what we just stored in State
- This is actually sort of a typical compose pattern

Guidance: Pass state down, bubble events up

- We can easily pass data from parent to child (ie, we can pass stuff arguments to composable functions we call)
- It's tricky to get that data back up... we don't get to `return` values to pass info back up
- When a child view needs to modify state stored at a higher level, we typically pass it something like an `onChange` callback which modifies the state
- This feels like sort of a hacky way to deal with the fact that information (state) can only flow top down

Example: clearing text on button submit

- We saw how we need State to have a functional editable text field
- Imagine a submit button went along with it...
- The state must be defined in the composable containing the TextField and the Button
- The button's onClick can read the value out of the state and then set it to "" to clear the text field
- The state needs to be in scope when we write the callback we want to give to the button so we can access the state from within it

How it works: “Composer” and “Gap Buffer”

- For more details see this great blog post
- The Compose compiler basically adds an extra parameter of type Composer to each @Composable function and passes it to calls to nested functions
- The tree of “views” is flattened into a “gap buffer” which is basically an array
- When things are redrawn, the data structure can quickly determine what needs to be updated, and can do so efficiently

Example: Weather App from yesterday

- The UI is all in our MainActivity. There's not even a corresponding layout XML file!
- LazyList is a MUCH easier tool to work with compared to RecyclerView!
- We can factor our common UI elements into functions (WeatherDataDisplay function in the example)

Wrapup

- The ecosystem/guidance from Google change rapidly. Be prepared to keep learning, even if you only work on one platform in your career
- You'll have to evaluate new technology as it comes out and decide if it's useful in your application
- I used SwiftUI in a project, and it was pretty great for simple stuff, but when I strayed from writing a “normal” app, it was difficult
- From what I hear, JC is easier to integrate with “legacy” UIs, but I suspect you'll have a tough time if you try to do wild stuff