“Jetpack” Navigation

Recap

- View-based World
- Activities / Activity lifecycle
- Binding
- MVVM architecture
- RecyclerView / Custom Views
- Fragments / Fragments Lifecycle
- Navigation
- Project discussion

Jetpack?

- Jetpack is a set of “modern” libraries for Android
- Ironically, some of them were built for view based UI designs, so are sort of obsolete
- ViewModel + LiveData classes + libraries are part of Jetpack
- Today we'll look at the “Navigation” component from Jetpack

Navigation graph

- A big idea is that you specify all the various paths between parts of your app as a graph, which is stored in an XML file
- To create one, right click “new” then choose “android resource file” and select “navigation”
- Just like a layout, once you open that XML file you can switch between code/design/split views of that file
- Usually the “nodes” of the graph are Fragments which correspond to screens

Single activity navigation

- To have Jetpack help us with navigation, we need a “container” view that will display whatever fragment we navigate to as appropriate
- To do this, we'll have our Activity just show a `FragmentContainerView` which will show a `NavHostFragment`
- You'll need to set these XML attributes: `app:defaultNavHost="true"`

  `app:navGraph="@navigation/your_xml_file"`
- This fragment will use the Navigation graph to determine which fragment to show first, and will respond to navigation requests and update the fragment for us
- No more manual FragmentTransaction stuff!

Navigating

- There's a lot you can do, see this doc for more info
- The most basic thing is to just call `findNavController().navigate(R.id.theFragmentId)`
- This does the fragment transaction correctly and manages the back stack for you
- You can pass arguments between fragments, but we'll mostly stick to using viewmodels to share data in a more centralized way

Testing

Unit testing: nothing special

- The JUnit library works just fine with Kotlin code in your android projects
- Any code that isn't UI related (involving Views/event listener stuff) should be straightforward to add test for (in the `java.your.package.test` folder)
- Even stuff like ViewModels which you might not think can be easily tested can be!

Junit

- The default Android Studio project actually comes with 2 provided unit tests
- The first is in the “test” directory of the “Java” folder which actually contains Kotlin code
- These are just plain unit tests, that by default run with JUnit 5
- You've used these a bunch in CS6012

“Instrumented Tests”

- Junit tests work fine for Kotlin code that's not “Android specific”
- These tests go in the `androidTest` folder
- Unfortunately google doesn't officially support Junit 5, so these are stuck with JU4
- Code that does data processing or things like that will work fine with vanilla JUnit, and should be added to the “test” folder
- Some code, however, does require Android stuff in order to run, and requires some setup to run
- Some surprising things fall into this category...

Example: Color

- The color class we've been using is in the android.graphics package, so it's an “Android” class even though it's probably a pretty simple class
- By default for normal unit tests, all Android classes are “mocked” or “stubbed” by the unit testing framework so calling **ANY** methods on them will cause an exception to be thrown
- Unfortunately, this means you need to use instrumented tests to test code that uses Colors (and many other simple classes)

Another example: LiveData

- LiveData also, unfortunately requires Android components to run, so must be run as an instrumented test
- And also there's some other irritating hoops to jump through
- We'll look at some tests I added for the viewmodel that we used to draw colored circles
- My test setup worked in this scenario, but to handle all the dark corners of running code in “coroutines” and other details, a detailed tutorial has been provided by google

Instrumented Testing

- Instrumented tests give you access to other Android objects including `Context` objects, etc
- This means you can test parts of your app that are more tightly coupled to Android, and you still don't need to run a full on app in the phone to do so
- But to test the full app, we'll need something more

UI Testing: Espresso

- Espresso is a library for doing automated testing on your full app's UI
- You can perform actions on views in the app (which you can, for example, select by ID)
- You can inspect views as well, and perform assertions like “assert the button is disabled” or “assert the text view contains this string”
- You can write espresso tests that perform a sequence of user actions and assert that the app behaves as expected
- There's lots more documentation here

Espresso example

Email Splitter

The Cheat sheet is really good