CS Android Programming: LiveData and ViewModel

The Rules. Recall the rules for flipped classrooms: do the work alone or with a partner that is unique for the semester. You can find the github link in canvas.

Only one student needs to submit this code, but both students may do so if they wish. This assignment should be submitted through github (sorry, partners can't share the repository). Fill in the provided README and be sure to include both student's names and EIDs.

The Purpose. The purpose of this assignment is to get you comfortable with LiveData and ViewModels by having you program some of the idioms you will come to use with these abstractions. For this FC, the ViewModel will be very simple, but we will hook everything up and get to see how LiveData allows the ViewModel and the View (which is implemented in a Fragment) to be related, but decoupled. If the View goes away, the ViewModel does not mind, it will carry on functioning. And Android requires no explicit code to make the ViewModel cope with the View disappearing.

The Dynamics. Watch the video to get a feel for what the dynamics are, but you have a producer fragment and a consumer fragment. They are both attached to an activity which creates the ViewModel and shares it with both fragments. The fragments communicate via the shared ViewModel. The producer creates a random 4 digit number as a string and two seconds later, it updates the view model, which causes both the producer and consumer to display the new value. Unless the consumer does not exist (i.e., its fragment layout is occupied by the EmptyFragment).

Because of the way navigation works, the best thing to do when we remove the consumer is navigate to the EmptyFragment.

You can hit the produce button, then kill the consumer (which really means navigating to the empty fragment). With no consumer, you can hit the produce button, then create a consumer. The consumer always sees the most up to date version of the live data. The video makes this clear.

The Instructions. Look for the `// XXX` comments in the code for places you have to modify. Heed the hints in the comments. If you do not see a `// XXX` , you do not need to change the code and you should not change the code.

– ~~**AndroidManifest.xml** All good.~~

– ~~**drawable/border.xml**~~ You d~~on't need to change this fil~~e, but you will use it. We haven't seen a shape drawable in XML before, but here it is. You can read the referred-to stackoverflow discussion where I found it.

– **activity_main.xml** Two fragments go here, equally spaced and equally sized, with a layout margin of 8dp (which is twice the value used in the video). The top is the producer and the bottom the consumer.

  Android says you should use a FragmentContainerView, but in 2024 that causes a crash, so we will stick with fragments for now. Either way, the layout object is a frame. It is a space where fragments can be navigated to using the nav graph.

  There are two technical requirements for your fragment objects. They need these properties.
  `android:name="androidx.navigation.fragment.NavHostFragment"`
  `app:navGraph="@navigation/mobile`$_{n}avigation$”

  The first line tells the Android runtime that this object is a NavHostFragment, which means its the kind of thing that can host multiple different fragments. In the case of the producer, the `<fragment>` object only ever hosts a single fragment, but we navigate to that fragment to pass it an argument (its name).

  The `app:navGraph` line tells the NavHostFragment what graph it is following. For this course we just need a single graph. We can put any fragment we want into the nav graph, they don't need to be connected.

– ~~**empty_layout.xml** Just exactly perfect~~.

– ~~**fragment_main.xml**~~ All good. It is a simple LinearLayout. The default title should never be visible in your app.
  Check out `android:background="@drawable/border"`.

– **navigation/mobile_navigation.xml** What is here is correct, but you need to add two global actions, which I call `createConsumer` and `createEmpty`. You can create a global action in the visual editor by clicking on the background and then choosing this arrow, which is the create action button.

  [height=0.25in]fc20-nav-action

  You can also write the action in the xml directly. The action to create the consumer fragment requires a string argument.

– **MainActivity.kt** The view binding stuff is standard, and the on click listeners should make sense.

  You need to implement the three navigate functions, which navigate to the producer, consumer and empty fragment. You will need a nav controller, and since this is MainActivity you need to pass it an ID from the layout where there is a fragment object.

  You will need Directions objects for the producer and consumer.

– **ProduceFragment.kt** Most of this is written. You have to initialize the view model and the navigation arguments. We have seen examples of this in the demo code.

You have to observe some live data from the viewModel and use it to update the text view.

Check out how we generate a random number between 0 and 9999 (inclusive on both ends). So cool, Sir Kotlin!

- **ConsumeFragment.kt** Just like the producer case, you need to get the view model and the navigation arguments.

  You need to write most of `onViewCreated`. ProduceFragment might have similar code. The title text comes from the navigation arguments. I provide the button text in a companion object. You need an on click listener and you should observe the data in the ViewModel.

- ~~**EmptyFragment.kt** No changes needed. Pretty simple, right?~~

- **ViewModel.kt** There is usually a bit more going on here, but the heart of the matter is a mutable string called data. Then we have two one-liners. For the return from the coroutine (the code inside the `launch` block), you need to communicate the string from the coroutine to the main thread. Check out the comments in the code for more details.

  And for `observeData` notice that the return type is `LiveData<String>`, but we have a `MutableLiveData<String>`. This situation is very common because the ViewModel can make changes to the underlying LiveData objects, but clients can't. It turns out, you don't have to do any work to turn a `MutableLiveData` into a `LiveData`, the Kotlin compiler will do the cast for you. What a pleasant language.