CSCI 448 – Lab 03C
Friday, February 03, 2023
LAB IS DUE BY **Tuesday, February 14, 2023, 11:59 PM**!!

The QA was thankful to have all the logging in place. They have found that the bug is reproduceable. They've opened a new ticket and the CTO has assigned it to you to resolve.

Step 0 – Inspect the Ticket

Reading the ticket information, it says:

> *When a user is playing the game, upon rotating the device their score is reset and they return to the first question.*
>
> *Furthermore, sometimes after a variable amount of time has passed and the user has left the app, upon returning to the app their score is reset as well. This is not always able to be duplicated but seems like the item above.*

You take it upon yourself to verify the ticket is accurate. Hoping for a quick "works for me" resolution, you open the app on your device and perform the following steps while following the logs:

- Open the app
- Answer first question correctly
- Move to second question
- Answer second question correctly
- Move to third question
- Rotate the device

It seems the ticket is correct. Inspecting the logs, the reason quickly becomes apparent: the configuration causes the activity to be recreated. Every time an activity is created, a new instance of the View Model is created as well.

Step 1 – Save State Across Configuration Changes

It's time to make our View Model a `ViewModel`. (Wait, what?) The View Model component of our MVVM architecture will be implemented using an Android Architecture Component (AC) – the Jetpack `ViewModel`.

Part 1.I – Setup the Jetpack ViewModel

Have your `QuestionViewModel` extend the abstract `ViewModel` class.

~~`class QuestionViewModel(private val mQuestions: List<Question>) {`~~
`class QuestionViewModel(private val mQuestions: List<Question>) : ViewModel() {`

Perfect, now to associate a single instance with the corresponding Activity.

Part 1.II – Build the Factory

Make a new Kotlin class alongside the `QuestionViewModel` called `QuestionViewModelFactory`. The factory needs to extend the abstract `ViewModelProvider.NewInstanceFactory()` class.

In order to encapsulate what the factory is producing, add a public function called `getViewModelClass()` that returns the `QuestionViewModel java class` instance.

```
fun getViewModelClass() = QuestionViewModel::class.java
```

Now it's time to override the factory's create method. Begin by logging what class the factory is attempting to create.

```
override fun <T : ViewModel> create(modelClass: Class<T>): T {
  Log.d(LOG_TAG, "Creating $modelClass")
}
```

It's possible the factory is asked to create something that it doesn't produce, so we want to first check that the `modelClass` can be assigned from the ViewModel class the factory produces. If it can, the factory will make an instance. If it can't, then the factory will throw an exception.

```
if( modelClass.isAssignableFrom(getViewModelClass()) )
  return ... // will fill this in next
throw IllegalArgumentException("Unknown ViewModel")
```

Kotlin is built on top of Java therefore we can leverage the reflective capabilities of Java. We'll ask the `modelClass` for a specific form of the constructor and then invoke the method with the appropriate arguments.

~~return ...~~
```
return modelClass
    .getConstructor(List::class.java)
    .newInstance(QuestionRepo.questions)
```

Now the Factory is the entity in charge of creating the ViewModel. It hides that the ViewModel requires data from the repository. In the future, the Factory will be tasked with the more complex task of retrieving the data from a database. Thus, the model layer is abstracted away from the View Model.

Part 1.III – Produce ViewModels (Actually, Just One – The Same One)

The current structure of `MainActivity` likely makes the problem immediately clear – every time inside of `onCreate()` a new View Model was being constructed:

```
setContent {
  ...
  QuestionScreen( QuestionViewModel() )
}
```

We'll need only one instance to exist on the activity. Add a `private lateinit` variable to the activity of type `QuestionViewModel`:

```
private lateinit var mViewModel: QuestionViewModel
```

We'll assign it inside of `onCreate()` before getting to the `setContent` composable.

Create an immutable object corresponding to the `QuestionViewModelFactory`.

```
val factory = ...
```

Next, we ask the `ViewModelProvider` to give us the ViewModel associated with our activity. The activity corresponds to the owner. If there's no ViewModel associated with the owner, then the factory will be used to create a ViewModel of the corresponding type.

```
mViewModel = ViewModelProvider(this, factory)[factory.getViewModelClass()]
```

If this is the first request to the provider, then the factory makes a new instance. The provider then associates this instance with the owner. If this is a subsequent request to the provider, then the provider returns the already created instance. We'll now send this instance down to the `QuestionScreen`.

Note: we cannot use the provider and factory in the Preview composable since we do not have an activity context to give the provider. For now, manually create a ViewModel to send to the `QuestionScreen` composable. We will better solve this problem later when we discuss interface abstraction and dependency injection.

Build, deploy, run, answer, rotate, follow the logs. Now when the activity is recreated, a new ViewModel instance is not created!

Step 2 – Save State Across Process Death

Having resolved the first part of the ticket, you decide to further investigate the second part. As the developer, you have some additional tools at your disposal the QA Team is unaware of.

Part 2.I – Force the Issue

Begin by overriding a method on `QuestionViewModel` – `onCleared()`. Log that this method is being called.

Now, on the device be sure the Developer Options are enabled (go to Settings > About > click on the build number seven times). Search through the Developer Options and turn on "Don't Keep Activities." This will force an activity to be discarded whenever you navigate away from the activity. This simulates the situation where the OS kills a process to regain the associated resources. Killing the process removes all objects associated with the activity.

Open the app, answer a few questions correctly, then go to the home screen. Look at the log: the activity was destroyed and the ViewModel was cleared. Return to your app and the logs show that a new ViewModel was created!

The QA Team was on to something here.

Part 2.II – Refactor the ViewModel

There's no avoiding the fact that since the activity is completely destroyed and removed from the provider's store list, that a new ViewModel instance will be created. The two pieces of information that we need to know are what was our previous score and what question were we previously on. Currently, both of these are being initialized to zero when the `QuestionViewModel` is being constructed.

We'll refactor our QuestionViewModel to accept these pieces of information inside the constructor (which currently only is expecting a list of questions). The current index will be set as a data member, but the initial score will only be a parameter that is used to initialize the state.

~~class QuestionViewModel(private val mQuestions: List<Question>) : ViewModel() {~~
~~private var mCurrentQuestionIndex = 0~~
~~private val mCurrentScoreState = mutableStateOf(0)~~

```
class QuestionViewModel(private val mQuestions: List<Question>,
                        private var mCurrentQuestionIndex: Int = 0,
                        initialScore: Int = 0) : ViewModel() {
  private val mCurrentScoreState = mutableStateOf(initialScore)
```

We're storing and tracking all the same information, but now the ViewModel can be created at various initial states.

We will also want to expose a public immutable value corresponding to the current question index we are currently on.

```
val currentQuestionIndex: Int
  get() = mCurrentQuestionIndex
```

Now that the ViewModel has changed how it's constructed, the Factory needs to be informed to change its production process.

Part 2.III – Refactor the Factory

The `QuestionViewModelFactory` oversees producing `QuestionViewModel`s. Since the ViewModel will require two integers for the initial index and score, the Factory will also need this information. Add two private immutable data members to the Factory constructor corresponding to the initial index and the initial score. Initialize both to zero by default.

The Factory uses the create() method to build a ViewModel. The current build process is using the constructor that accepts only a list. The Factory now needs to request the constructor that accepts a list and two integers.

~~.getConstructor(List::class.java)~~

```
.getConstructor(List::class.java, Int::class.java, Int::class.java)
```

Then when making the new instance, provide the Factory's data members respectively.

~~.newInstance(QuestionRepo.questions)~~

```
.newInstance(QuestionRepo.questions, initialIndex, initialScore)
```

The Factory now has the ability to produce ViewModels in different configurations. However, the additional parameters are never being set when the Factory is created.

Part 2.IV – Leverage the Bundle

Since the values we need correspond to the previous state the application was in, we'll first handle writing out the existing state. Then when we are restarting the application, we'll load the previous state to initialize our current.

Part 2.IV.A – Put Data on the Bundle

`MainActivity` is the primary interface between our application and the operating system. The `ActivityManager` is responsible for telling our activity what to do. The conversation the `ActivityManager` has with our activity goes something like the following:

> "Hi, `MainActivity`? This is the `ActivityManager`. I need to put something on top of you, let's pause." *`onPause()` is called*
>
> "Yea, it's me again. We're going to place you in the background for the moment. I need you to stop, mmmkay?" *`onStop()` is called*
>
> "It turns out we're going to need you to pack your things. But first, if there's anything your replacement may need, then can you save them to this `Bundle`? *`onSaveInstanceState()` is called*
>
> "Already, here's the door. Ciao." *`onDestroy()` is called*

The activity is given notice that it's going to be destroyed and given an opportunity to save any state information associated with this given instance. The `ActivityManager` will hold onto the resultant `Bundle` for some period of time, in case a similar new activity is created in the near future.

On the existent `companion object`, add two private constant strings that will correspond to the keys for the `"index"` and `"score"`.

Override the `onSaveInstanceState(Bundle)` method and the final statement should be passing the `Bundle` object up to the `super` implementation. Before we send the `Bundle` to the parent implementation, we'll need to manipulate it with our implementation's specific tasks.

First, log that the method is called. Then put the integer corresponding to the current index onto the Bundle. Do the same for current score. Finally, send the `Bundle` upwards.

Part 2.IV.B – Retrieve Data from the Bundle

Inspecting the method signature for `onCreate()`, it is now apparent that the `ActivityManager` may give us an existent `Bundle` when it calls the `onCreate()` method for our activity. If the `ActivityManager` has a `Bundle` associated with a prior instance of our activity, then it will provide it. Otherwise, the `ActivityManager` will send `onCreate()` `null` (nothing).

We'll need to check if the `Bundle` object we're given exists or not and if it does exist then check if the values we need exist within or not.

We'll need to store the initial index and score to use. If they are present on the `Bundle`, then we'll use the stored values. Otherwise, we'll start at zero. We could check if the object is not null or not, but we'll leverage the safe call and elvis operators to assign our values.

```
val initialIndex = savedInstanceState?.getInt(KEY_INDEX, 0) ?: 0
```

The safe call operator only accesses the integer if the `Bundle` is not null. If it is null, then zero is returned (after the `?:` operator). When getting the value for a given key, if the key is not found then zero is returned by default. Otherwise, the associated value for the key is returned.

Add a similar line for the score.

After setting the initial index and score, log their values.

To complete the state flow, provide these two values to the Factory when it is created to use as the initial state within the ViewModel.

Build, deploy, run, answer a few questions correctly, go to the home screen, and then finally come back into the app. Your state has been saved and restored! Follow the log sequence to verify this is correctly occurring. The key points to detect within the flow:

- `MainActivity::onCreate()`
- `Initial index and score == 0`
- `Factory creating`
- `ViewModel initialized`
- `MainActivity::onSaveInstanceState()`
- `ViewModel::onCleared()`
- `MainActivity::onDestroy()`
- `MainActivity::onCreate()`
- `Initial index and score != 0`

Step 3 – Deploy Your App & Submit

You've pushed your fixes to the build, mark the ticket as resolved, and notify the QA Team. After testing the latest version, they have accepted your changes are mark the ticket as complete and closed! The CTO excitedly informs you it is time to publish Quizler out to the world.

When Lab03 is fully complete, you will submit a video of your working app to Canvas. Demonstrate the following actions inside the app:

- With the device in portrait mode
- Answer the first question correctly (score == 1 & correct toast)
- Move to the second question
- Answer the second question incorrectly (incorrect toast)
- Answer the second question correctly (score == 2 & correct toast))
- Move to the third question
- Continually move to the next question until you reach the third question again
- Continually move to the previous question until you reach the third question again
- Rotate the device to landscape mode (score== 2 && question == 3)
- Rotate back to portrait mode (score== 2 && question == 3)
- Go to the home screen
- Open the app again (score== 2 && question == 3)

Then stop the recording. Save it as webm format, name the video `<username>_L03.webm`, and upload this file to Canvas Lab03.

Working through the submission process you've noticed the landscape layout could be nicer – some of the buttons are cut off. You start to get an idea of how you can really impress the CTO and starting sketching some mockups on your own...

vvv !!! IMPORTANT !!! vvv

Step 4 – Reset the Device Settings

Be absolutely sure to go back into the Developer Options and <u>turn OFF the "Don't Keep Activities" setting</u>. Otherwise, there will be very frustrating side effects on your device moving forward.

^^^ !!! IMPORTANT !!! ^^^

LAB IS DUE BY **Tuesday, February 14, 2023, 11:59 PM**!!