CSCI 448 – Lab 06A
Monday, February 27, 2023
LAB IS DUE BY **Tuesday, March 07, 2023, 11:59 PM**!!

After the success of Quizler, Little Green Games was so impressed with your contributions that they have made you a full-time offer to be a Junior Android Developer as part of their games division. You eagerly accept and are given a new project to work on.

You now know what game the other developers had been playing previously – Samodelkin, the latest tabletop game gaining instant popularity. Some of the other developers had started making a companion app but were not able to finish it. They were able to get most of the MVVM components in place but stopped before getting the navigation in place. Over the next series of sprints, the following stories have been requested to be put in place:

1. Implement navigation to view, add, and delete characters from each player's codex.
2. Persist a player's codex to persistent local storage.
3. Load new characters from the remote character repository.
4. Transfer a user's characters to another user.

You pull the project code and set out on hooking up the existing screens.

Step 0 – Load The Project

Begin by opening the provided Android Studio project. Ensure that your Android Studio is up to date with the following:

- Android Studio Electric Eel
- Under Tools > SDK Manager > SDK Tools
    - Android SDK Build-Tools 34.0.0 rc1
    - Android SDK Platform-Tools 34.0.0
- Under Settings > Build, Execution, Deployment > Build Tools > Gradle
    - Gradle JDK needs to support Java8, YMMV but JDK v11 tends to work (you may not need to change this setting)

Take a code walk to look at the structure of each MVVM component and the functionality it provides or supports.

- The Model: SamodelkinCharacter and Repository
- The View: List screen, Detail screen, New Character screen, and specifications
- The View Model: ViewModel and Factory

You should now be able to build and deploy the app – the character list screen will be displayed when the app is opened.

Step 1 – Put The NavHost In Place

Part 1.I – Screen Specifications

The SamodelkinNavHost composable has already been started to work with the IScreenSpec interface. Begin by completing each screen's specific specification:

- ListScreenSpec
  - Displays SamodelkinListScreen for the list of characters the ViewModel is holding.
  - When a specific item is clicked, navigates to the DetailScreen of that item.
- DetailScreenSpec
  - Displays SamodelkinDetailScreen for the character that corresponds to the UUID provided as the route argument.
  - Note on how the ViewModel loads a character:
    - The public currentCharacterState member stores the single character object that the ViewModel is loading. This member is a StateFlow object and the View needs to collectAsState() from the Flow to a State object to be observed.
    - Once the View is observing the character state, request the ViewModel to load a character by UUID. This method will update the character state with the corresponding result.
- NewCharacterScreenSpec
  - This will be completed in Step 3.

Part 1.II – The Scaffold

With the NavHost & NavGraph now complete, MainActivity needs to make use of the navigation framework. Currently, the activity is always displaying the list with no action provided for an item selection.

In its place we will use a Scaffold since we will soon be adding a top menu bar to use on each screen. Before adding in the Scaffold, create immutable variables to hold on to the NavController and the current Context.

Now, replace the static call to SamodelkinListScreen with a call to the Scaffold composable. For now, there will be no arguments. In the trailing lambda corresponding to the content, be sure to name the paddingValues argument. Inside the trailing lambda, call through to your SamodelkinNavHost composable passing through the NavController, the SamodelkinViewModel, the Context, and specifying the padding Modifier to apply.

At this point build, deploy, and run the app. You should be able to navigate between the list and corresponding detail if the screen specifications were set up correctly.

Step 2 – Add The TopBar

Time to add some decoration to the screen. The next step is to add a top header bar that displays the title of our app throughout the experience. This will appear on every screen, so will become part of our specification.

Part 2.I – Specify the Title

Begin by adding a member field to the `IScreenSpec` interface called `title` and returns an integer. On each concrete screen specification, override this value to be our app name (stored in `R.string.app_name`). Note, it is possible that each screen could display a different title along the top menu bar but our application will only print `Samodelkin` consistently for the user.

Part 2.II – Create the `TopAppBar`

On `IScreenSpec` again, create a private composable function called `TopAppBarContent` that accepts a `SamodelkinViewModel`, a `NavHostController`, a nullable `NavBackStackEntry`, and the `Context`. Inside this method, we will call the `TopAppBar` composable and specify two arguments for the composable call:

Part 2.II.A – `navigationIcon`

The `navigationIcon` argument allows us to display the hamburger menu, the up arrow, or nothing. Since we will not have a side drawer layout the hamburger menu is not an option. Instead, when on the detail screen we'll display the up arrow and when on the list screen we'll display nothing. More accurately, if we're on any child destination of our NavGraph we'll display the up arrow to go back to its parent. When we are at a child destination, the previous back stack entry will not be null since our parent exists.

Set the `navigationIcon` as follows:

```
navigationIcon = if (navController.previousBackStackEntry != null) {
  {
    IconButton(onClick = { navController.navigateUp() }) {
      Icon(
        imageVector = Icons.Filled.ArrowBack,
        contentDescription = stringResource(R.string.menu_back_desc)
      )
    }
  }
} else {
  { }
}
```

Pay very careful attention to where and how curly braces are used to denote code blocks. This is due to the dual roles of `if-else` in Kotlin. Here we are using an `if` expression (not an `if` statement). With the `if` expression, the value of the `if-else` branch is assigned. The values in each branch are themselves a lambda expression.

Part 2.II.B – `title`

After setting the navigationIcon, set the title to be a Text composable resolving the string resource for our title member field created in Part 2.I.

Part 2.III – Expose the `TopBar`

This will be done in two pieces.

Part 2.III.A – Have `IScreenSpec` Expose the `TopBar`

Create a static public composable function on IScreenSpec in the companion object called TopBar that accepts all the parameters required for the TopAppBarContent method previously created. We will need to call the TopAppBarContent method, but it needs to be done in a specific way.

We can't call it directly since we would never have an actual instance of our IScreenSpec interface. Therefore, we need to call it on an instance of a concrete child that implements the full interface. Note how the allScreens member is created for this app – it's slightly different than in Quizler. Here, allScreens is a map that uses the screen's route as the key an the object instance as the value.

Therefore, in our TopBar method well first get the route that corresponds to our current destination (if one exists):

```
val route = navBackStackEntry?.destination?.route ?: ""
```

Then, we'll get the matching the object from the map and call its TopAppBarContent method:

```
allScreens[route]?.TopAppBarContent( ... )
```

Part 2.III.B – Cleanly Encapsulate the `TopBar`

We want to clean up the IScreenSpec usage and provide an entry point to abstract the existence and usage of the IScreenSpec. In the presentation.navigation package create a new composable function called SamodelkinTopBar. This will behave in a similar manner to how SamodelkinNavHost abstracts the IScreenSpec for the NavHost creation.

Have SamodelkinTopBar accept parameters for the NavHostController, SamodelkinViewModel, and Context.

Inside the function, first get a reference to the current back stack entry state:

```
val navBackStackEntryState = navController.currentBackStackEntryAsState()
```

Now, call through to IScreenSpec.TopBar and provide all the matching arguments.

Part 2.IV – Add the `TopBar`

The final step is to specify the topBar argument in the Scaffold of MainActivity. Invoke the SamodelkinTopBar composable with the necessary arguments.

Build, deploy, run, and see the top bar in action! Beyond the "Samodelkin" title that appears, when on the detail page the user can now use the up arrow to return to the list.

If you'd like, on the TopAppBar composable itself, you can specify the colors argument using TopAppBarDefaults.topAppBarColors() in a manner like the button colors.

Step 3 – Add New Characters

Time to get our third screen in place while also expanding our top bar.

Part 3.I – Finish NewCharacterScreenSpec

NewCharacterScreenSpec will need to call through to the NewCharacterScreen composable. The New Character screen is given an initial character and then the user can choose to either continue creating new random characters or save their current character. These latter two actions are provided to the screen via lambda functions.

In the Content method of NewCharacterScreenSpec, first remember a mutable state for the character initialized to a random character returned by the character generator.

```
val characterState = remember {
  mutableStateOf( CharacterGenerator.generateRandomCharacter(context) )
}
```

Now we can call the NewCharacterScreen composable.

The onGenerateRandomCharacter function needs to update the characterState value to a new random character using the CharacterGenerator.

The onSaveCharacter function will first add this character to the ViewModel and then navigate back to the list. This second step is accomplished by telling the NavController to pop to a specific parent.

```
navController
  .popBackStack(route = ListScreenSpec.buildRoute(), inclusive = false)
```

Part 3.II – Add TopAppBarActions

Our NewCharacterScreen is ready to use, now a way to navigate to it must be provided. To do so, we'll add menu buttons to our top bar.

Part 3.II.A – Abstract the Menu Actions

On the IScreenSpec, add a public abstract composable method called TopAppBarActions that accepts as parameters a SamodelkinViewModel, a NavHostController, a nullable NavBackStackEntry, and a Context.

Part 3.II.B – Place the Menu Actions

When the `TopAppBar` composable is created, in addition to setting the `navigationIcon` and `title` also specify the `actions` argument. Invoke the `TopAppBarActions` method providing the matching arguments.

Part 3.II.C – Implement the Menu Actions

Begin in the `DetailScreenSpec` and `NewCharacterScreenSpec` by implementing the abstract `TopAppBarActions` method and providing an empty function body. For now, these screens will have no menu actions.

The `ListScreenSpec` will include a button to navigate to the `NewCharacterScreen` to add a character to the list.

The `actions` argument of `TopAppBar` is already set up as a `Row` composable, so we only need to specify the contents of the row. Override the `TopAppBarActions` method on `ListScreenSpec` and use an `IconButton` (like the `navigationIcon` in Part 2.II.A). For this button, the `onClick` function should navigate to the `NewCharacterScreenSpec route`. The `Icon` will use the `Icons.Filled.AddCircle imageVector` and the `contentDescription` will be the `R.string.menu_add_character_desc` string.

Build, deploy, run, and begin adding random characters to your codex!

Step 4 – Delete Existing Characters

The `DetailScreenSpec` will provide a mechanism to delete a character and remove it from the list. Now we will provide an implementation for `DetailScreenSpec::TopAppBarActions`.

Begin by observing the `characterState` of the associated route argument as is done in `DetailScreenSpec::Content`.

Then add the `IconButton`. The `Icon` used will be the `Icons.Filled.Delete imageVector` with a `contentDescription` of `R.string.menu_delete_character_desc`.

The `onClick` for the `IconButton` will perform two steps. First it will delete the corresponding character from the ViewModel. Then it will pop back to the `ListScreen` destination.

Build, deploy, run, and delete any characters that are not fit for adventure at this time.

Step XC – Polish the App

There two possibilities for extensions and extra credit at this point. Both are items that we will eventually discuss, but you are encouraged to look ahead and do some research into how they are implemented. View the example video on the resources page for expected UX.

Part XC.I – Confirm Deletion

A part of good UX is adding a confirmation to prevent accidental deletion of data. When the user presses the delete character menu item, first present a dialog. If the user presses cancel, do nothing. If the user presses OK, proceed with the deletion.

Part XC.II – Notify of Adding and Deleting

Once a character is added or deleted to the list, display a `Snackbar` with a message of what action was performed on the character. The `Snackbar` is like a `Toast`. However, the `Toast` will persist beyond our app as part of the System UI. The `Snackbar` is a component of our `Scaffold` that remains within the context of our app.

Step 5 – Deploy Your App & Submit

When Lab06 is fully complete, you will submit a video of your working app to Canvas. Demonstrate the following actions inside the app:

- Press the add new character menu button
- Press the up arrow
- Press the add new character menu button
- Press the generate random character button twice
- Press the save character to codex button
- Scroll to the newly added character
- Select the newly added character
- Press the up arrow
- Select the newly added character
- Press the delete character menu button

Then stop the recording. Save it as webm format, name the video `<username>_L06.webm`, and upload this file to Canvas Lab06.

The app isn't yet ready for deployment yet until the backend team gets the database in place. You start looking to request your own LGG Intern to assist with the rest of the stories.

LAB IS DUE BY **Tuesday, March 07, 2023, 11:59 PM**!!