CSCI 448 – Lab 07A
Wednesday, March 01, 2023
LAB IS DUE BY **Friday, March 10, 2023, 11:59 PM**!!

While the Little Green Games HR Department begins putting together the posting to find the next set of interns, you get started on the next story so Samodelkin can be pushed out to the LGG Store. Users will want to be able to store their characters across campaigns, so the database needs to be incorporated.

> ***Comment from Dr. Paone:*** *The addition of the Room database will involve refactoring the entire Model-View-View Model stack. Therefore, the app will not be able to be fully tested until Step 3 is fully complete. You may wish to make a backup of your code in the event you need to roll back changes.*

Step 0 – Add the Dependencies

The Room database requires multiple items to be included at different levels.

Part 0.I – `build.gradle (Project: Samodelkin)`

In the top level Gradle file, specify the version of the kapt plugin we'll be using.

```
id 'org.jetbrains.kotlin.kapt' version '1.6.0' apply false
```

Part 0.II – `build.gradle (Module :app)`

In the module Gradle file, first add the `kapt` plugin to the top `plugins` block.

```
plugins {
  id 'kotlin-kapt'
}
```

Then in the `dependencies` block, add the room dependencies and the kapt compiler.

```
dependencies {
  implementation 'androidx.room:room-runtime:2.5.0'
  implementation 'androidx.room:room-ktx:2.5.0'
  kapt 'androidx.room:room-compiler:2.5.0'
}
```

Step 1 – Model: Create the Room Database Structure

> ***Comment from Dr. Paone**: Be sure to reference the slides for the parallel example of building up all the various components.*

Part 1.I – Mark `SamodelkinCharacter` as an `@Entity`

The `SamodelkinCharacter` class needs to be marked as an entity so it can be converted into a SQL table. Perform the following steps:

- Annotate `SamodelkinCharacter` as an `Entity` with the `tableName` of `character`.
- Annotate the `avatarAssetPath` field with a `ColumnInfo name` of `avatar`.
- Annotate the `id` field as the `PrimaryKey`.

Part 1.II – Create `SamodelkinTypeConverters`

Since the `UUID` field of our table is not a primitive SQL type, SQLite does not know how to convert the value into a string. We'll need to create our own custom type converter to work with the database.

First, create a new package named `data.database`. In this package, create a new class named `SamodelkinTypeConverters`.

A type converter follows a very specific format – any function to convert from a type to a string is named `from`*`Class`*`(obj: `*`Class`*`): String` where *`Class`* should be replaced with the class type name. Likewise any function to convert from a string to a type is named `to`*`Class`*`(str: String):` *`Class`* where again *`Class`* should be replaced with the class type name. Additionally, these functions are annotated as a `TypeConverter`. Create the necessary functions on your newly created class.

```
@TypeConverter
fun fromUUID(uuid: UUID?) = uuid?.toString()

@TypeConverter
fun toUUID(uuid: String?) = UUID.fromString(uuid)
```

Part 1.III – Create `SamodelkinDatabase`

Inside of the `database` package, create an `abstract class` named `SamodelkinDatabase` and extends `RoomDatabase`.

Add the `@Database` annotation using the `SamodelkinCharacter` class entity and marking the database as version `1`.

Additionally, add the @TypeConverters annotation to use your custom type converter.

```
@Database(...)
@TypeConverters(SamodelkinTypeConverter::class)
abstract class SamodelkinDatabase : RoomDatabase() {
  ...
}
```

Then create the singleton instance using a database name of "`samodelkin-database`".

Part 1.IV – Create **`SamodelkinDao`**

Inside of the `database` package, create an `interface` named `SamodelkinDao` with the `@Dao` annotation.

Part 1.IV.A – Create RUD

Add a create function called `addCharacter()` that accepts a `SamodelkinCharacter` parameter. This is a `suspend` function and uses the `@Insert` annotation.

Part 1.IV.B – C Read UD

Add a read function called `getCharacters()` that returns a `Flow` of a `List` of `SamodelkinCharacter`. The `@Query` annotation should select all characters.

Add a second read function called `getCharacterById()` that accepts a `UUID` parameter. This is a `suspend` function and returns a nullable `SamodelkinCharacter`. This `@Query` annotation adds the `WHERE id=(:id)` clause to the SQL statement.

Part 1.IV.C – CRU Delete

Add a delete function called `deleteCharacter()` that accepts a `SamodelkinCharacter` parameter. This is a `suspend` function and uses the `@Delete` annotation.

Part 1.IV.D – Add the DAO to the Database

Return to `SamodelkinDatabase` and add an abstract immutable member field corresponding to the `SamodelkinDao`.

Part 1.V – Refactor **`SamodelkinRepo`**

Part 1.V.A – Refactor the Singleton

First, change the parameters the constructor accepts. It will need to store a private immutable object corresponding to the `SamodelkinDao` and store a reference to a `CoroutineScope` that defaults to `GlobalScope`.

class Samodelkin
~~private constructor(context: Context?) {~~
private constructor(private val samodelkinDao: SamodelkinDao,
**   private val coroutineScope: CoroutineScope = GlobalScope) {**

> ***Comment from Dr. Paone:*** *Android Studio will give you a warning regarding the use of `GlobalScope`. This is an important warning to read, do so before accepting the `@OptIn` fix.. Since the Global Scope is not bound to the lifecycle of any specific object, its coroutine can continue to run in the background. This is a potentially dangerous and inefficient operation if not used properly and with care. The lifespan of the coroutines we will be launching will be*

short tasks to the device and therefore are a safe operation to perform in the Global Scope. We would not want to perform a remote request to an external database via Global Scope since that request could hang/lag/etc. However, since we made this object a constructor parameter, we can provide a different coroutine scope to use at some point in the future if our repository will be working with different data sources.

Now that the `SamodelkinRepo` constructor changed, we need to update how the `getInstance()` method generates the singleton. First, do not allow the `getInstance()` parameter to be nullable. Then, if the current instance is null first get the reference to the database. Then provide the DAO repository constructor.

The last modification to our object creation is to remove the generation of a random list inside of the `init` block.

Part 1.V.B – Expose the DAO Façade

The repository acts as a passthrough to the DAO. Add the following methods on the repository:

- **`fun getCharacters()`** – returns `DAO::getCharacters()` result
- **`suspend fun getCharacter(UUID)`** – returns `DAO::getCharacterByid()` result
- **`fun addCharacter(SamodelkinCharacter)`** – calls `DAO::addCharacter()` inside of the `coroutineScope`
- **`fun deleteCharacter(SamodelkinCharacter)`** – calls `DAO::deleteCharacter()` inside of the `coroutineScope`

Step 2 – ViewModel: Observe the Model Repository

Before making any changes to the `SamodelkinViewModel` class, we're going to create a backup. Copy the file and rename the copy (and its contained class) to be `PreviewSamodelkinViewModel`. We will come back to this new class in Step 4 and we don't want to completely lose prior functionality we had in place.

Part 2.I – Refactor `SamodelkinViewModel`

The first order of business is to change the constructor parameter from the list of characters to be an instance of the repository. We do want to mark the repository object as a private immutable member field.

Since our list is now hidden behind the repository, we will need to rework how we manipulate the list.

Part 2.I.A – Refactor the Add and Delete Methods

Instead of `addCharacter()` directly adding the character to the list, we'll need to ask the repository to add the character.

Instead of `deleteCharacter()` searching for the character to delete and then deleting it, we'll need to ask the repository to delete the character.

Yay abstraction! Each function is two lines: (1) logging the call (2) invoking the repository.

Part 2.I.B – Refactor the Read Methods

There are two types of reading we need to handle: (1) reading all characters (2) reading a single character.

Part 2.I.B.i – Read All Characters

Begin by changing mCharacters from being a mutable state list (which made the type SnapshotStateList) to be a MutableStateFlow that is initially an empty list.

```
private val mCharacters = characters.toMutableStateList()
private val mCharacters: MutableStateFlow<List<SamodelkinCharacter>> =
     MutableStateFlow(emptyList())
```

We continually use a Flow object to receive any updates the list may have occur.

Similarly update the public characters member to reflect that it is a StateFlow.

```
~~val characters: List<SamodelkinCharacter>~~
~~  get() = mCharacters.toList()~~
val characterListState: StateFlow<List<SamodelkinCharacter>>
  get() = mCharacters.asStateFlow()
```

Now add an `init` block. We need to listen for changes to the list from the repository and send along the notification to the View to update the state being displayed. This is where our two levels of Flow will come in to play. In the init block, launch a viewModelScope coroutine.

```
init {
  viewModelScope.launch {

  }
}
```

Inside the coroutine, we need to collect the updates from the repository's getCharacters() flow.

```
samodelkinRepo.getCharacters().collect { characterList -> }
```

The value we collect will be the new list object. We'll use this list object to update our internal StateFlow.

```
samodelkinRepo.getCharacters().collect { characterList ->
  mCharacters.update { characterList }
}
```

What happens with all this? When our View Model is created, we launch a coroutine tied to the lifecycle of the View Model object. On the coroutine, we open a cold flow to the repository that queries all the characters. When the repository receives a list of characters from the DAO, it emits the new list value on the flow. Our coroutine then collects the new list value and updates the flow on the View Model. Any Views that are listening to our flow are then notified of the new value and the UI recomposes. The repository flow is cold, but as long as our View Model stays alive then the coroutine will stay alive. Our View Model will remain a consumer of the repository flow and that flow stream will remain open. If the

DAO reports any changes to the list from the database, then the updates will propagate up from the Model database to the View composable.

Part 2.I.B.ii – Read A Single Character

Reading a single character will follow a similar two-tiered flow structure. However, this time the two flows will correspond to different object types and we'll need to manipulate the flow midstream.

Add a private immutable member field called `mCurrentCharacterIdState` that is a `MutableStateFlow` of type `UUID` and is initialized to a random UUID.

```
private val mCurrentCharacterIdState: MutableStateFlow<UUID> =
     MutableStateFlow(UUID.randomUUID())
```

Inside of the existing `loadCharacterByUUID` method, remove the current contents (except the Log call). The only task should be to update the ID state with the provided `uuid` value.

```
mCurrentCharacterIdState.update { uuid }
```

This posts a new value to the ID flow stream and this value now needs to be collected. In the `init` block, launch a second `viewModelScope` coroutine.

```
init {
  viewModelScope.launch {
    samodelkinRepo.getCharacters().collect { characterList ->
      mCharacters.update { characterList }
    }
  }
  viewModelScope.launch {

  }
}
```

This second coroutine is where we'll add an intermediate flow operator before we do the collection. We will want to map the new UUID we receive to the corresponding character from the database. We'll then collect this character and post it to the existing character state flow.

```
mCurrentCharacterIdState
  .map { uuid -> samodelkinRepo.getCharacter(uuid) }
  .collect { character -> mCurrentCharacterState.update { character } }
```

But wait, `SamodelkinRepo::getCharacter()` is a blocking function. That's fine. The map transformation is expecting a blocking call. The entire process is already occurring on a separate coroutine.

> ***Comment from Dr. Paone:*** *We're now starting to see the power of the Observer design pattern and the asynchronous listening flows provide. This is how we should have had Quizler structured with our index and question state. In Quizler, whenever we modified the current question index we manually updated the question state. In Samodelkin we have a similar relationship where changing one value should trigger the change of a second value. We are now able to do that. Our View Model is internally listening for changes to the current ID state.*

When the View sends an event to change the ID state, we hear that change and transform the state change into a different type and post that change back to the View.

Take additional note of the functional programming style that the map-collect interface provides for our flows.

Part 2.II – Patch `SamodelkinViewModelFactory`

We just changed how the View Model gets constructed. The Factory is in charge of constructing View Models. Therefore, the Factory needs to be updated accordingly.

SamodelkinViewModel's constructor now accepts a SamodelkinRepo object. Get the corresponding constructor and provide it with the corresponding instance.

It is now apparent why we use the Factory to create View Models. The Factory internally is responsible for requesting the Repository instance. The Repository internally is responsible for requesting the Database instance. The Database internally is responsible for creating itself. The Factory kicks off this complex construction process and encapsulates all the chained requests behind a single interface function call.

Step 3 – View: Listen to the View Model

Luckily most of the refactoring to the View Model involved internal modifications to how the functions were implemented and resolved. However, we did change how we were exposing the UI State and will need to observe the new state objects. Our View does not depend upon on many state variables so there is not too much work to be done in this layer.

Part 3.I – Patch `ListScreenSpec`

Previously, the View Model exposed the list of characters directly. Now the View Model more appropriately wraps the list of characters inside of a `StateFlow` object.

```
val characterListState =
     samodelkinViewModel.characterListState.collectAsState()
```

Then pass the value of this state object to the `SamodelkinListScreen` composable.

Part 3.II – Temporarily Fix `SamodelkinListScreen`

Since we changed our Repository, this broke how we were providing the character list to the `SamodelkinListScreen` inside the composable preview. For the time being, comment out of the composable preview. We will solve this problem in Step 4.

Part 3.III – Temporarily Fix `MainActivity`

Since we changed our View Model, this broke how we were providing the view model object to the `MainActivityContent` inside the composable preview. The problem is that the database cannot be created outside of the full Android runtime environment so we cannot create the view model to use inside the composable preview. For the time being, comment out of the composable preview. We will solve this problem in Step 4.

Part 3.IV – Run the Database

With all the refactoring complete and patching in place, we can now run the application. Build, deploy, and run the app on a device. When the app initially starts, the database is empty so no list will appear. After going through the new character screen and saving a character to the codex, the character should then appear in the list. The details for that character can then be brought up and the character deleted.

Time for the patented self high five: put your right hand above your head with the palm facing to the left. With your left hand, slap your right hand. There should be a feeling of accomplishment to get the database backend persisting the data across sessions.

> ***Comment from Dr. Paone:*** *As you are working on the database portion, if you make a change to your database schema (the tables/columns/etc) then you'll have a database clash and your app won't run. In production, you would update the version of your database via the annotation and provide a migration plan. However, in our development purposes the simpler solution is to uninstall the app from your device and then reinstall the app. Upon running the application for the first time, the new database will be created. If the database is not working properly, add Log calls throughout the MVVM to trace the data being sent down and up the stack. The logcat will be your friend.*

At this point, our app is fully functioning, and we could be done. However, it is only functioning for the front-end user. The back-end developer still has several problems to work through, and they all relate to testing. Time to go solve the problems from Parts 3.II and 3.III.

Step 4 – Program to an Interface

Any attempt to test our app efficiently will now fail with the Room database in place. The required View Model cannot be created without a full Android runtime environment and file system. Both are needed to create the Room database instance. This means we currently cannot:

- Test our View composable previews through Android Studio.
- Unit test our Model, View, and View Model functionality.

These problems exist because we are violating one of our design principles:

Program to an interface, not an implementation.

Throughout our View, many of our composable functions depend directly on the concrete `SamodelkinViewModel`. The functions are expecting a parameter of this specific type to be provided. Therefore, our View is dependent on having this specific implementation available. With all our layers of abstraction, we don't truly care how the connected event handlers are implemented. We only care that the event handler is present. Our fix to this problem will be an early form of Dependency Injection (DI) where we are using our interface design principle to manually provide different implementations based on the current context of the environment.

Part 4.I – Create `ISamodelkinViewModel`

Part 4.I.A – Create the Interface

In the `viewmodel` package, create a new `interface` named `ISamodelkinViewModel`. Add public members for all the existing public members on our `SamodelkinViewModel` class:

- `characterListState: StateFlow<List<SamodelkinCharacter>>`
- `currentCharacterState: StateFlow<SamodelkinCharacter?>`
- `loadCharacterByUUID(UUID)`
- `addCharacter(SamodelkinCharacter)`
- `deleteCharacter(SamodelkinCharacter)`

Part 4.I.B – Depend Upon the Interface

We now need to update all the references where the concrete `SamodelkinViewModel` is used as a dependency and replace them with a reference to the abstract `ISamodelkinViewModel`. Refactor these dependencies in order:

- `IScreenSpec::TopBar, Content, TopAppBarContent, TopAppBarActions`
- `DetailScreenSpec::Content, TopAppBarActions`
- `ListScreenSpec::Content, TopAppBarActions`
- `NewCharacterScreenSpec::Content, TopAppBarActions`
- `SamodelkinNavHost`
- `SamodelkinTopBar`
- `MainActivity::MainActivityContent`

We're now depending only on an interface and thankfully due to our abstraction the only dependencies are present in the navigation layer. All the Views properly depend upon state to be provided down and any events are sent up.

Part 4.II – `SamodelkinViewModel` implements `ISamodelkinViewModel`

Update `SamodelkinViewModel` to implement `ISamodelkinViewModel` in addition to extending the `ViewModel` abstract class. `override` all the corresponding public members of the interface.

If you read the documentation that accompanies the `ViewModel()` constructor, it says you should never manually construct a `ViewModel` outside of a `ViewModelProvider.Factory`. We were doing that previously in our composable previews! Eek. Now, the only place the concrete `SamodelkinViewModel` is created is during the `MainActivity` creation via the factory. Phew.

Part 4.III – `PreviewSamodelkinViewModel` implements `ISamodelkinViewModel`

The previous View Model snapshot we saved and called `PreviewSamodelkinViewModel` – it's time to use and fix it. Have this class implement the `ISamodelkinViewModel` interface instead of extending the `ViewModel` abstract class. There's some patching to be done to conform to our new interface, but the existing functionality to manipulate an internal `MutableList` is already in place. Luckily, we did not lose the prior working implementation.

First, change the constructor to accept a `Context` instead of a list of characters.

Next, set `mCharacters` to be an empty mutable list.

```
private val mCharacters: MutableList<SamodelkinCharacter> =
      mutableListOf()
```

Then remove the public `characters` member field and replace it with a private `mCharacterListState` that is a mutable state flow of a list of characters initialized to the current character list.

```
private val mCharacterListState = MutableStateFlow(mCharacters.toList())
```

Now override the public member `characterListState` with the getter return the private value as a state flow.

```
override val characterListState: StateFlow<List<SamodelkinCharacter>>
  get() = mCharacterListState.asStateFlow()
```

Question Time!

You may be asking "Why do we keep making private mutable lists with a private mutable state of an immutable list and expose a public immutable state for an immutable list? Why don't we just make a mutable state of a mutable list?"

We can answer the last part of the first question directly – we don't want to expose a mutable form of our state because then any client that is connected to our View Model could directly change the state. This could have undesired consequences for all the other clients that are also connected to our View Model. Therefore, we want to encapsulate and control the state changes internally to the View Model.

To answer the other questions, read the following article titled Two mutables don't make a right (https://dev.to/zachklipp/two-mutables-dont-make-a-right-2kgp). Then complete the Canvas Quiz titled "Lab07A Mutability" using the access code `TeenageMutableNinjaTurtles`.

Returning to `PreviewSamodelkinViewModel`, override the `currentCharacterState` that is in place.

Override `loadCharacterByUUID()` that is in place.

Override `addCharacter()`. After we add the character to the list, update the `mCharacterListState` with the now modified list.

Override `deleteCharacter()`. After we remove the character from the list, update the `mCharacterListState` with the now modified list.

Finally, add an `init` block to add ten random characters to the list. *Hint: call `addCharacter()` ten times.*

Now `PreviewSamodelkinViewModel` provides mocked database functionality using a temporary list to store the corresponding model data. We can use this concrete instance in any necessary tests.

Part 4.IV– Fix The Composable Previews

The places that we were improperly depending on our `ViewModel` was in our composable previews. This is where we were making the `ViewModel` outside of the Factory.

Part 4.IV.A – Patch `SamodelkinListScreen`

Uncomment the `SamodelkinListScreen` preview. The steps we need to put in place are:

1. Get a reference to the current local context
2. Create an instance of `PreviewSamodelkinViewModel`
3. Collect the `characterListState` flow as state
4. Provide the state value to the list screen composable

With a rebuild, the composable preview will come back to life!

Part 4.IV.B – Patch `MainActivity`

The `MainActivity` preview will look very similar. This time, once the `PreviewSamodelkinViewModel` instance is created we will pass it directly to `MainActivityContent`.

With that, nothing noticeable has changed on the front end for the user. However, rest well knowing the back end is now properly implemented as an interface dependency.

Step 5 – Fix A Warning

One more quick fix. You may be wondering about the schema warning that keeps popping up every time you build.

```
Schema export directory is not provided to the annotation
processor so we cannot export the schema. You can either
provide room.schemaLocation annotation processor argument OR set
exportSchema to false.
```

We could set the export value to false and that silences the warning, but that doesn't provide much benefit. We would much prefer to write out the database schema to file. This allows us to put the database schema in our version control system and replicate the database structure that the app is using. Additionally, as we migrate database versions each version schema will be written out separately so we can patch external instances to match as well.

In the build.gradle (Module :app) file, add the following option to the android block.

```
android {
  ...
  defaultConfig {
    ...
    javaCompileOptions {
      annotationProcessorOptions {
        arguments += [
          "room.schemaLocation": "$projectDir/schemas".toString()
        ]
      }
    }
  }
}
```

Now when building the warning will be gone and if you navigate to the `app/schemas` folder of your project, you will find your database schema saved as a JSON file.

Step XC – Good UX

To provide a better user experience, when the list is empty, and no characters exist don't just display an empty screen. Instead, for extra credit, if the list is empty display a message stating no characters exist and have a button that brings the user to the new character screen. View the example video on the resources page for expected UX.

Step 6 – Deploy Your App & Submit

When Lab07 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 save character to codex button
- Select the newly added character
- Press the add new character menu button
- Press the save character to codex button
- Press the recents button
- Swipe to close your app
- Reopen your app
- Select the last added character
- Press the delete character menu button

Then stop the recording. Save it as webm format, name the video `<username>_L07.webm`, and upload this file to Canvas Lab07.

Excited to have a functioning usable app, you begin packaging up the app for deployment.

LAB IS DUE BY **Friday, March 10, 2023, 11:59 PM**!!