Project 1 : Lemonade app

Step 1: Configured the ImageView

1. updated setOnClickListener() the app's state. The method to do this is clickLemonImage().
2. setOnLongClickListener() responds to events where the user long presses on an image (e.g. the user taps on the image and doesn't immediately release their finger). For long press events, a widget, called a snackbar, appears at the bottom of the screen letting the user know how many times they've squeezed the lemon. This is done with the showSnackbar() method.

Step 2: Implemented clickLemonImage()

After completing the previous step, the clickLemonImage() method is now called each time the user taps the image. This method is responsible for moving the app from the current state to the next and updating any variables as needed. There are four possible states: SELECT, SQUEEZE, DRINK, and, RESTART; the current state is represented by the lemonadeState variable. This method needs to do something different for each state.

1. SELECT: Transition to the SQUEEZE state, set the lemonSize (the number of squeezes needed) by calling the pick() method, and setting the squeezeCount (the number of times the user has squeezed the lemon) to 0.
2. SQUEEZE: Increment the squeezeCount by 1 and decrement the lemonSize by 1. Remember that a lemon will require a variable number of squeezes before the app can transition its state. Transition to the DRINK state only if the new lemonSize is equal to 0. Otherwise, the app should remain in the SQUEEZE state.
3. DRINK: Transition to the RESTART state and set the lemonSize to -1.
4. RESTART: Transition back to the SELECT state.

Step 3: Implemented setViewElements()

The setViewElements() method is responsible for updating the UI based on the app's state. The text and image should be updated with the values shown below to match the lemonadeState.

SELECT:

- Text: Click to select a lemon!
- Image: R.drawable.lemon_tree

SQUEEZE:

- Text: Click to juice the lemon!
- Image: R.drawable.lemon_squeeze

DRINK:

- Text: Click to drink your lemonade!
- Image: R.drawable.lemon_drink

RESTART:

- Text: Click to start again!
- Image: R.drawable.lemon_restart

All codes are uploaded in the github :

Project 2 : Dogglers app

Implement the layout

Both the vertical and horizontal layouts are identical, so you only need to implement a single layout file for both. The grid layout displays all the same information, but the dog's name, age, and hobbies are stacked vertically, so you'll need a separate layout for this case. Both layouts require four different views to display information about each dog.

1. An `ImageView` with the dog's picture
2. A `TextView` with the dog's name
3. A `TextView` with the dog's age
4. A `TextView` with the dog's hobbies

You'll also notice some styling on each card to show a border and a shadow. This is handled by `MaterialCardView`, which is already added to the layout files in the starter project. Within each `MaterialCardView` is a `ConstraintLayout` where you'll need to add the remaining views.

Implement the adapter

Once I've defined your layouts, your next task is to implement the RecyclerView adapter. Open up DogCardAdapter.kt in the **adapter** package. You'll see there are lots of TODO comments that help explain what you need to implement.

Project 3 : Lunch Tray app

Define the ViewModel

As seen in the screenshots on the previous page, the app asks for three things from the user: an entree, a side, and an accompaniment. The order summary screen then shows a subtotal and calculates sales tax based on the selected items, which are used to calculate the order total.

In the **model** package, open up OrderViewModel.kt and you'll see that a few variables are already defined. The menuItems property simply allows you to access the DataSource from the ViewModel

First, there are also some variables for previousEntreePrice, previousSidePrice, and previousAccompanimentPrice. Because the subtotal is updated as the user makes their choice (rather than being added up at the end), these variables are used to keep track of the user's previous selection if they change their selection before moving to the next screen. You'll use these to ensure the subtotal accounts for the difference between prices of the previous and currently selected items.

There are also private variables, _entree, _side, and _accompaniment, for storing the currently selected choice. These are of type MutableLiveData<MenuItem?>. Each one is accompanied by a public backing property, entree, side, and accompaniment, of immutable type LiveData<MenuItem?>. These are accessed by the fragments' layouts to show the selected item on screen. The MenuItem contained in the LiveData object is also nullable since it's possible for the user to not select an entree, side, and/or accompaniment.

```
// Entree for the order
    private val _entree = MutableLiveData<MenuItem?>()
    val entree: LiveData<MenuItem?> = _entree

    // Side for the order
    private val _side = MutableLiveData<MenuItem?>()
    val side: LiveData<MenuItem?> = _side

    // Accompaniment for the order.
```

```
    private val _accompaniment = MutableLiveData<MenuItem?>()
    val accompaniment: LiveData<MenuItem?> = _accompaniment
```

There are also `LiveData` variables for the subtotal, total, and tax, which use number formatting so that they're displayed as currency.

```
// Subtotal for the order
    private val _subtotal = MutableLiveData(0.0)
    val subtotal: LiveData<String> = Transformations.map(_subtotal) {
        NumberFormat.getCurrencyInstance().format(it)
    }

    // Total cost of the order
    private val _total = MutableLiveData(0.0)
    val total: LiveData<String> = Transformations.map(_total) {
        NumberFormat.getCurrencyInstance().format(it)
    }

    // Tax for the order
    private val _tax = MutableLiveData(0.0)
    val tax: LiveData<String> = Transformations.map(_tax) {
        NumberFormat.getCurrencyInstance().format(it)
    }
```

Finally, the tax rate is a hardcoded value of 0.08 (8%).

```
private val taxRate = 0.08
```

There are six methods in `OrderViewModel` that you'll need to implement.

setEntree(), setSide(), and setAccompaniment()

All of these methods should work the same way for the entree, side, and accompaniment respectively. As an example, the `setEntree()` should do the following:

1. If the `_entree` is not `null` (i.e. the user already selected an entree, but changed their choice), set the `previousEntreePrice` to the current `_entree`'s price.
2. If the `_subtotal` is not `null`, subtract the `previousEntreePrice` from the subtotal.
3. Update the value of `_entree` to the entree passed into the function (access the `MenuItem` using `menuItems`).
4. Call `updateSubtotal()`, passing in the newly selected entree's price.

The logic for setSide() and setAccompaniment() is identical to the implementation for setEntree().

updateSubtotal()

updateSubtotal() is called with an argument for the new price that should be added to the subtotal. This method should do three things:

1. If _subtotal is not null, add the itemPrice to the _subtotal.
2. Otherwise, if _subtotal is null, set the _subtotal to the itemPrice.
3. After _subtotal has been set (or updated), call calculateTaxAndTotal() so that these values are updated to reflect the new subtotal.

calculateTaxAndTotal()

calculateTaxAndTotal() should update the variables for the tax and total based on the subtotal. Implement the method as follows:

1. Set the _tax equal to the tax rate times the subtotal.
2. Set the _total equal to the subtotal plus the tax.

resetOrder()

resetOrder() will be called when the user submits or cancels an order. You want to make sure your app doesn't have any data left over when the user starts a new order.

Implement resetOrder() by setting all the variables that you modified in OrderViewModel back to their original value (either 0.0 or null).

Create data binding variables

Implement data binding in the layout files. Open up the layout files, and add data binding variables of type OrderViewModel and/or the corresponding fragment class.

You'll need to implement all the TODO comments to set the text and click listeners in four layout files:

1. fragment_entree_menu.xml
2. fragment_side_menu.xml
3. fragment_accompaniment_menu.xml
4. fragment_checkout.xml

Each specific task is noted in a TODO comment in the layout files, but the steps are summarized below.

1. In fragment_entree_menu.xml, in the <data> tag, add a binding variable for the EntreeMenuFragment. For each of the radio buttons, you'll need to set the entree in the ViewModel when it's selected. The subtotal text view's text should be updated accordingly. You'll also need to set the onClick attribute for the cancel_button and next_button to cancel the order or navigate to the next screen respectively.
2. Do the same thing in fragment_side_menu.xml, adding a binding variable for the SideMenuFragment, except to set the side in the view model when each radio button is selected. The subtotal text will also need to be updated, and you'll also need to set the onClick attribute for the cancel and next buttons.
3. Do the same thing once more, but in fragment_accompaniment_menu.xml, this time with a binding variable for AccompanimentMenuFragment, setting the accompaniment when each radio button is selected. Again, you'll also need to set attributes for the subtotal text, cancel button and next button.
4. In fragment_checkout.xml, you'll need to add a <data> tag so that you can define binding variables. Within the <data> tag, add two binding variables, one for the OrderViewModel, and another for the CheckoutFragment. In the text views, you'll need to set the names and prices of the selected entree, side dish, and accompaniment from the OrderViewModel. You'll also need to set the subtotal, tax, and total from the OrderViewModel. Then, set the onClickAttributes for when the order is submitted, and when the order is canceled, using the appropriate functions from CheckoutFragment.

Initialize the data binding variables in the fragments

Initialize the data binding variables in the corresponding fragment files inside the method, onViewCreated().

1. EntreeMenuFragment
2. SideMenuFragment
3. AccompanimentMenuFragment
4. CheckoutFragment

Create the navigation graph

As you learned in Unit 3, a navigation graph is hosted in a FragmentContainerView, contained in an activity. Open activity_main.xml and replace the TODO with the following code to declare a FragmentContainerView.

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/mobile_navigation"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

The navigation graph, `mobile_navigation.xml` is found in the **res.navigation** package.

This is the navigation graph for the app. However, the file is currently empty. Your task is to add destinations to the navigation graph and model the following navigation between screens.

1. Navigation from `StartOrderFragment` to `EntreeMenuFragment`
2. Navigation from `EntreeMenuFragment` to `SideMenuFragment`
3. Navigation from `SideMenuFragment` to `AccompanimentMenuFragment`
4. Navigation from `AccompanimentMenuFragment` to `CheckoutFragment`
5. Navigation from `CheckoutFragment` to `StartOrderFragment`
6. Navigation from `EntreeMenuFragment` to `StartOrderFragment`
7. Navigation from `SideMenuFragment` to `StartOrderFragment`
8. Navigation from `AccompanimentMenuFragment` to `StartOrderFragment`
9. The **Start Destination** should be `StartOrderFragment`

Once you've set up the navigation graph, you'll need to perform navigation in the fragment classes. Implement the remaining `TODO` comments in the fragments, as well as `MainActivity.kt`.

1. For the `goToNextScreen()` method in `EntreeMenuFragment`, `SideMenuFragment`, and `AccompanimentMenuFragment`, navigate to the next screen in the app.
2. For the `cancelOrder()` method in `EntreeMenuFragment`, `SideMenuFragment`, `AccompanimentMenuFragment`, and `CheckoutFragment`, first call `resetOrder()` on the `sharedViewModel`, and then navigate to the `StartOrderFragment`.
3. In `StartOrderFragment`, implement the `setOnClickListener()` to navigate to the `EntreeMenuFragment`.
4. In `CheckoutFragment`, implement the `submitOrder()` method. Call `resetOrder()` on the

sharedViewModel, and then navigate to the StartOrderFragment.

5. Finally, in MainActivity.kt, set the navController to the navController from the NavHostFragment.

Project 4 : Amphibians app

Implement the API service

The app displays a list of amphibian data from the network. The amphibian data comes from a JSON object returned by the API. Take a look at the Amphibian.kt file in the **network** package. This class models a single amphibian object, a list of which will be returned from the API. Each amphibian has three properties: a name, type, and description.

```
data class Amphibian(
        val name: String,
        val type: String,
        val description: String
    )
```

The project already has the Retrofit and Moshi dependencies. In the **network** package, you'll find the AmphibianApiService.kt file. The file contains several TODO comments. Perform the following five tasks to implement the amphibians app.:

1. Create a variable to store the API's base URL.
2. Build the Moshi object with Kotlin adapter factory that Retrofit will be using to parse JSON.
3. Build aRetrofit instance using the Moshi converter.
4. Implement the AmphibianApiService interface with a suspend function for each API method (for this app, there's only one method, to GET the list of amphibians).
5. Create an AmphibianApi object to expose a lazy-initialized Retrofit service that uses the AmphibianApiService interface.

Implement the ViewModel

that need to be displayed. You'll do this in the AmphibianViewModel.kt class found in the **ui** package.

You'll notice that above the class declaration is an enum called AmphibianApiStatus.

```
enum class AmphibianApiStatus {LOADING, ERROR, DONE}
```

The three possible values, LOADING, ERROR and DONE, are used to show the status of the request to the user.

In the AmphibianViewModel.kt class itself, you'll need to implement some LiveData variables, a function to interact with the API, and a function to handle setting the amphibian on the detail screen.

1. Add a _status a private MutableLiveData variable that can hold an AmphibianApiStatus enum and backing property status for the status.
2. Add an amphibians variable and private backing property _amphibians for the list of amphibians, of type List<Amphibian>.
3. Add an _amphibian variable of type MutableLiveData<Amphibian> and backing property amphibian for the selected amphibian object, of type LiveData<Amphibian>. This will be used to store the selected amphibian shown on the detail screen.
4. Define a function called getAmphibianList(). Launch a coroutine using ViewModelScope, inside the coroutine, perform a GET request to download the amphibian data by calling the getAmphibians() method of the Retrofit service. You'll need to use try and catch to appropriately handle errors. Before making the request, set the value of the _status to AmphibianApiStatus.LOADING . A successful request should set _amphibians to the list of amphibians from the server and set the _status to AmphibianApiStatus.DONE. In the event of an error, _amphibians should be set to an empty list and the _status set to AmphibianApiStatus.ERROR.
5. Implement the onAmphibianClicked() method to set the _amphibian property you defined to the amphibian argument passed into the function. This method is already called when an amphibian is selected, so that it will be displayed on the detail screen

Update the UI from the ViewModel

After implementing the ViewModel, all that's left to do is edit the fragment classes and layout files to use the data bindings.

1. The ViewModel is already referenced in AmphibianListFragment. In the onCreateView() method, after the layout is inflated, simply call the getAmphibianList() method from the ViewModel.
2. In the fragment_amphibian_list.xml, the <data> tags for the data binding variables have already been added to the layout files. You just need to implement the TODOs for the UI to update based on the view model. Set the appropriate bindings for the listData and apiStatus.

3. In fragment_amphibian_detail.x

Project 5 : Forage app

Define the Forageable entity

The project already has a Forageable class that defines the app's data (**model.Forageable.kt**). This class has several properties: id, name, address, inSeason, and notes.

```
data class Forageable(
    val id: Long = 0,
    val name: String,
    val address: String,
    val inSeason: Boolean,
    val notes: String?
)
```

However, in order to use this class to store persistent data, you'll need to convert it to a Room entity.

1. Annotate the class using @Entity with the table name "forageable_database".
2. Make the id property the primary key. The primary key should be auto-generated.
3. Set the column name for the inSeason property to "in_season".

Implement the DAO

ForageableDao (**data.ForageableDao.kt**), as you might guess, is where you define methods for reading and writing from the database that you will access from the view model. As the DAO is only an interface that you define, you won't actually have to write any code to implement these methods. Instead, you should use Room annotations, specifying the SQL query where needed.

Within the ForageableDao interface, you'll need to add five methods.

1. A getForageables() method that returns a Flow<List<Forageable>> for all rows in the database.
2. A getForageable(id: Long) method that returns a Flow<Forageable> that matches the specified id.

3. An insert(forageable: Forageable) method that inserts a new Forageable into the database.
4. An update(forageable: Forageable) method that takes an existing Forageable as a parameter and updates the row accordingly.
5. A delete(forageable: Forageable) method that takes a Forageable as a parameter and deletes it from the database.

Implement the view model

The ForageableViewModel (**ui.viewmodel.ForageableViewModel.kt**) is partially implemented, but you'll need to add functionality that accesses the DAO methods so it can actually read and write data. Perform the following steps to implement ForageableViewModel.

1. An instance of ForageableDao should be passed as a parameter in the class constructor.
2. Create a variable of type LiveData<List<Forageable>> that gets the entire list of Forageable entities using the DAO, and converts the result to LiveData.
3. Create a method that takes an id (of type Long) as a parameter and returns a LiveData<Forageable> from calling the getForageable() method on the DAO, and converting the result to LiveData.
4. In the addForageable() method, launch a coroutine using the viewModelScope and use the DAO to insert the Forageable instance into the database.
5. In the updateForageable() method, use the DAO to update the Forageable entity.
6. In the deleteForageable() method, use the DAO to update the Forageable entity.
7. Create a ViewModelFactory that can create an instance of ForageableViewModel with a ForageableDao constructor parameter.

Implement the Database class

The ForageDatabase (data.ForageDatabase.kt) class is what actually exposes your entities and DAO to Room. Implement the ForageDatabase class as described.

1. Entities: Forageable
2. Version: 1
3. exportSchema: false
4. Inside the ForageDatabase class, include an abstract function to return a ForageableDao
5. Inside the ForageDatabase class, define a companion object with a private variable called INSTANCE and a getDatabase() function that returns the ForageDatabase singleton.

6. In the BaseApplication class, create a database property that returns a ForageDatabase instance using lazy initialization.

Forageables list

The forageables list screen just requires two things: a reference to the view model, and access to the full list of forageables. Perform the following tasks in **ui.ForageableListFragment.kt**.

1. The class already has a viewModel property. However, this is not using the factory you defined in the previous step. You'll need to first refactor this declaration to use the ForageableViewModelFactory.

```
private val viewModel: ForageableViewModel by activityViewModels {
    ForageableViewModelFactory(
      (activity?.application as BaseApplication).database.foragableDao()
    )
}
```

2. Then in onViewCreated(), observe the allForageables property from the viewModel and call submitList() on the adapter where appropriate to populate the list.

Forageable details screen

You'll do almost the same thing for the detail list in **ui/ForageableDetailFragment.kt**.

1. Convert the viewModel property to correctly initialize the ForageableViewModelFactory.
2. In onViewCreated(), call getForageable() on the view model, passing in the id, to get the Forageable entity. Observe the livedata and set the result to the forageable property, and then call bindForageable() to update the UI.

Add and edit forageables screen

Finally, you'll need to do a similar thing in **ui.AddForageableFragment.kt**. Note that this screen is also responsible for updating and deleting entities. However, these methods from the view model are already called in the correct place. You'll only need to make two changes in this file.

1. Again, refactor the viewModel property to use ForageableViewModelFactory.
2. In onViewCreated(), in the if statement block before setting the delete button's visibility, call getForageable() on the view model, passing in the id, and setting the result to the forageable

property.

That's all you need to do in the fragments. You can now run your app and should be able to see all the persistence functionality in action.

Project 6 : Water Me! app

All the functionality for the Water Me! app is already implemented, except for the part to schedule and a notification. The code for displaying a notification is in `WaterReminderWorker.kt` (in the **worker** package). This happens in the `doWork()` method of a custom `Worker` class. Because notifications may be a new topic, this code is already implemented.

```
override fun doWork(): Result {
        val intent = Intent(applicationContext, MainActivity::class.java).apply {
            flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
        }

        val pendingIntent: PendingIntent = PendingIntent
            .getActivity(applicationContext, 0, intent, 0)

        val plantName = inputData.getString(nameKey)

        val builder = NotificationCompat.Builder(applicationContext, BaseApplication.CHANNEL_ID)
            .setSmallIcon(R.drawable.ic_android_black_24dp)
            .setContentTitle("Water me!")
            .setContentText("It's time to water your $plantName")
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setContentIntent(pendingIntent)
            .setAutoCancel(true)

        with(NotificationManagerCompat.from(applicationContext)) {
            notify(notificationId, builder.build())
        }

        return Result.success()
    }
```

Your task is to create a `OneTimeWorkRequest` that will call this method with the correct parameters

from PlantViewModel.

Create work requests.

To schedule the notification, you'll need to implement the scheduleReminder() method in PlantViewModel.kt.

1. Create a variable called data using Data.Builder. The data should consist of a single string value where WaterReminder.Worker.nameKey is the key and the plantName that was passed into scheduleReminder() is the value.
2. Create a one-time work request using WaterReminderWorker, using the delay and unit passed into the scheduleReminder() function, and setting the input data to the data variable you created.
3. Call the workManager's enqueueUniqueWork() method, passing in the plant name, using REPLACE as the ExistingWorkPolicy, and the work request.

Your app should now be working as expected. Since each reminder will take a long time to appear, we recommend running the included tests to verify that the notification works as expected.