Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 17:

Navigation continued

Objectives

- Navigation continued
  - Shared Layout & Navigation Bars
  - Nested Navigation
- New language feature: Sealed Classes
- Kahoot Quiz #2

Shared Layout

- We'd like to give a shared look-and-feel to our app.
- Typically, this means that we would use the same Scaffold layout, with the same TopBar, BottomBar, etc.
- But, if we just manually replicate that code, that isn't very elegant.
- Plus, when we have out animated transitions, the top bar and bottom bar also animate, even though they probably "shouldn't".
  - This is a design choice though
- So, we'd like to have a shared layout that can be used across our different routes.
- Let's define one…

MainLayout: "Wrapper" Composable

- We'd like to define a new Composable (e.g., called MainLayout) that will contain all the main layouting-related composables (e.g., Scaffold, Top Bar, App Bar, etc.)
- MainLayout should have at least one parameter – a lambda that will accept composable content
  - content: @Composable () -> Unit
  - This will allow you to pass appropriate composables to display within the MainLayout.
  - Note: Later, we may wish to add parameters such as "screenTitle"
- Within the Scaffold's content, we want to call the lambda to embed the content that was passed in

```
Column(modifier = Modifier.padding(it)) {
    content()
}
```

- Now, each of the other screens (MainScreen, AboutScreen, ContactScreen), can call MainLayout.

Try It!

- Create a new folder called layout and define a new composable in that folder called MainLayout.
- Create an appropriate MainLayout.kt file containing a MainLayout composable. In the composable,
    - Define an appropriate lambda parameter for passing in content
    - Use a Scaffold composable and any other shared layouting (e.g., app bars)
    - Use a Column inside the Scaffold to apply the required padding parameter
- Now, use MainLayout in your Main, About and Contact Screens
    - Remove any Scaffold or obsolete layouting used in them

- This layout will provide the same Scaffold to all screens that use it.
    - Note the use of content() to simply embed whatever was passed to this layout (cool!)

```
@Composable
fun MainLayout(
    content: @Composable () -> Unit
) {

    val navController = LocalNavController.current
    Scaffold(
        topBar = { TopAppBar(title = { Text("My App") }) },
        bottomBar = { BottomAppBar { Text("Copyright (c) 2023 CoolEntertainment, Inc.") } },
        floatingActionButton = { FloatingActionButton(onClick = {navController.navigate("MainScreenRoute")}) {
             Text("Home") } },
    ) {
        Column(modifier = Modifier.padding(it)) {
            content()
        }
    }
}
```

```
@Composable
fun MainScreen() {
    val navController = LocalNavController.current
    MainLayout() {
      ...
    }
}

@Composable
fun ContactScreen() {
    val navController = LocalNavController.current
    MainLayout() {
      ...
    }
}
```

Shared Navigation

- Now that we have defined a shared layout that has access to the navController, we can also define shared Navigation behaviours
- App Bars
  - https://developer.android.com/jetpack/compose/components/app-bars
- TopAppBar
  - Across the top of the screen.
  - Provides access to key tasks and information. Generally hosts a title, core action items, and certain navigation items.
  - Different types: TopAppBar, CenterAlignedTopAppBar, LargeTopAppBar, MediumTopAppBar
- BottomAppBar
  - Across the bottom of the screen.
  - Typically includes core navigation items. May also provide access to other key actions, such as through a contained floating action button.

TopAppBar Parameters

- title: The text that appears across the app bar.
- navigationIcon: The primary icon for navigation. Appears on the left of the app bar.
- actions: Icons that provide the user access to key actions. They appear on the right of the app bar.
- scrollBehavior: Determines how the top app bar responds to scrolling of the scaffold's inner content.
  - enterAlwaysScrollBehavior: When the user pulls up the scaffold's inner content, the top app bar collapses. The app bar expands when the user then pulls down the inner content.
  - exitUntilCollapsedScrollBehavior: Similar to enterAlwaysScrollBehavior, though the app bar additionally expands when the user reaches the end of the scaffold's inner content.
  - pinnedScrollBehavior: The app bar remains in place and does not react to scrolling.
- colors: Determines how the app bar appears.

BottomAppBar Parameters

- actions: A series of icons that appear on the left side of the bar. These are commonly either key actions for the given screen, or navigation items.
- floatingActionButton: The floating action button that appears on the right side of the bar.

- Note: You can also use BottomAppBar without passing a value for actions and floatingActionButton.
    - In that case, you can create a custom bottom app bar by filling BottomAppBar with content as you would other containers.

Try It!

- Define appropriate top and bottom app bars in their own files
  - These can be stored in the layout folder, for example
- In the Top bar, provide a back button
- In the Bottom bar, provide buttons to all 3 screens (Home, About, Contact)
- Add a screenTitle parameter to MainLayout and pass that down to the top bar so that each screen will show an appropriately tailored title

```
@Composable
fun MainLayout(
    screenTitle: String,
    content: @Composable () -> Unit
) {

    val navController = LocalNavController.current
    Scaffold(
        topBar = { SharedTopBar(screenTitle) },
        bottomBar = { SharedBottomBar() },
    ) {
        Column(modifier = Modifier.padding(it)) {
            content()
        }
    }
}
```

```
@Composable
fun SharedTopBar(screenTitle: String){
    val navController = LocalNavController.current
    CenterAlignedTopAppBar(title = { Text(screenTitle) },
        navigationIcon = {
            IconButton(onClick = { navController.navigateUp()}) {
                Icon(
                    imageVector = Icons.Filled.ArrowBack,
                    contentDescription = "Go Back"
                )
            }
        },
        actions = {
            IconButton(onClick = { /* do something */ }) {
                Icon(
                    imageVector = Icons.Filled.Menu,
                    contentDescription = "Menu"
                )
            }
        },
        )
}
```

```
@Composable
fun SharedBottomBar() {
    val navController = LocalNavController.current
    BottomAppBar(
        actions = {
            IconButton(onClick = { navController.navigate("MainScreenRoute")}) {
                Icon(
                    imageVector = Icons.Filled.Home,
                    contentDescription = "Go Home"
                )
            }
            IconButton(onClick = {
                    navController.navigate("AboutScreenRoute/Franz")}) {
                Icon(Icons.Filled.Info, contentDescription = "Go to About Us")
            }
            IconButton(onClick = {
                    navController.navigate("ContactScreenRoute/Julie/Paris")}) {
                Icon(Icons.Filled.Phone, contentDescription = "Go to Contact Us")
            }
        })
}
```

Nested Navigation

- So far, we've been able to define multiple routes, but none of the routes are related to each other
- Often, in a more complex app, we may have parts of our app that contain related functionality.
  - For instance, "login" and "signup" operations are related to each other, but not to a product page.
  - In a web app, these would naturally fall under a common URL path
- In our Android app, we'd like to define routes with some hierarchy
  - This makes the navigation in our code easier to understand and maintain
- Links
  - https://nameisjayant.medium.com/nested-navigation-in-jetpack-compose-597ecdc6eebb

Nested Routing: navigation()

- Jetpack Compose provides an extension function called `navigation()`
- This is used within your NavHost to define a nested set of routes.
- A navigation() contains two parameters (route and startDestination) and then defines its subroutes via composables

```
composable("Main") { MainScreen() }
navigation(route="Register", startDestination = "Register/Login") {
    composable("Register/Login") {LoginScreen()}
    composable("Register/Signup") {SignupScreen()}
}
composable("About") { AboutScreen()
```

Accessing Routes using Strings -> Poor Design

- In our Router, we defined the unique route names as strings, and then had to use those strings in our other classes.
  - This may lead to code that is more difficult to maintain.
  - For example, if we want to change the name of the route, we may have to make changes in several places.
- What we would like to do instead is create the names in the Router and then access those names.
- Kotlin offers a new type of class called a Sealed Class that lets us do this easily.

Sealed Classes

- Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type.
- Often we need to represent a limited set of possibilities;
  - A web request either succeeds or fails,
  - A User can only be a Pro-User or a standard user
  - There is a specific set of unique routes
- We could use an enum, but they are limited.
  - E.g., can only allow a single instance of each value
  - Can't encode more information on each type.
- You could use an abstract class, but this loses the restricted set of types advantage of enums.
- Kotlin provides **sealed classes** to allow the best of both worlds: The freedom of representation of abstract classes with the restricted set of types of enums.

Sealed Classes

- To declare a sealed class or interface, put the **sealed** modifier before its name
- No other subclasses may appear outside the module and package within which the sealed class is defined.
  - For example, third-party clients can't extend your sealed class in their code.
- Like abstract classes, sealed classes allow you to represent hierarchies.
- Sealed classes cannot have public constructors (private by default)
- The child classes can be any type of class: a data class, an object, a regular class or even another sealed class. Unlike abstract classes, you have to define these hierarchies in the same file or as nested classes.

- https://www.freecodecamp.org/news/how-to-handle-ui-events-in-jetpack-compose/
- https://developer.android.com/jetpack/compose/navigation
- https://medium.com/androiddevelopers/sealed-with-a-class-a906f28ab7b5
- https://proandroiddev.com/understanding-kotlin-sealed-classes-65c0adad7015
- https://cazimirroman.medium.com/sealed-classes-vs-data-classes-669446e8ed3b

Sealed Class rules

- Sealed classes are abstract and can have abstract members.
- Sealed classes cannot be instantiated directly.
- Sealed classes cannot have public constructors (The constructors are private by default).
- Sealed classes can have subclasses, but they must either be in the same file or nested inside of the sealed class declaration.
- Sealed classes subclass can have subclasses outside of the sealed class file.

Sealed Class Example

- The below is an example of a sealed class (Result) with two possible subclasses (Success and Error), each of which behaves slightly differently

```
sealed class Result<out T : Any> {
    data class Success<out T : Any>(val data: T) : Result<T>()
    data class Error(val exception: Exception) : Result<Nothing>()
}
```

- Trying to extend the sealed class outside the file it was defined in yields a compile error
- When using a sealed class, we can check a result's type similarly to an enum:

```
when(result) {
    is Result.Success -> { }
    is Result.Error -> { }
}
```

Hierarchical

- We can extend subclasses when defining our sealed class.

```
sealed class Result<out T : Any> {
  data class Success<out T : Any>(val data: T) : Result<T>()
  sealed class Error(val exception: Exception) : Result<Nothing>() {
     class RecoverableError(exception: Exception) : Error(exception)
     class NonRecoverableError(exception: Exception) :
                                          Error(exception)
  }
  object InProgress : Result<Nothing>()
}
```

- These can be used as follows, for example:

```
val action = when (result) {
    is Result.Success -> { }
    is Result.Error.Recoverable -> { }
    is Result.Error.Nonrecoverable -> {}
    Result.InProgress ->
}
```

Pro/Con Sealed Classes

- Pros:
    - One of the main benefits of using sealed classes is that they can help to ensure that your code is **correct and easy to understand**. Because sealed classes only allow a fixed set of subclasses, you can be sure that any object of a sealed class will be one of the defined subclasses. This makes it easier to reason about your code and can help to reduce the risk of bugs.
    - Another advantage of sealed classes is that they can make it easier to write code that **handles different states in a consistent way**. For example, you can use a when expression to handle each of the different states, and the Kotlin compiler will ensure that you have covered all of the possible states. This can make it easier to write correct code and can save you time and effort.
- Cons:
    - However, sealed classes also have some drawbacks. One of the main drawbacks is that they can be **more verbose** than other options, especially if you have a large number of states. Defining a sealed class and all of its subclasses can take more code than using other options, such as data classes or enums.

Comparing sealed vs data vs enum classes

```
// Using a sealed class
sealed class AppState {
    object Loading : AppState()
    data class Success(val data: Data) : AppState()
    data class Error(val message: String, val stackTrace: String) : AppState()
}

// Using a data class
data class AppState(val state: String, val data: Data? = null, val message: String? = null)

// Using an enum
enum class AppState {
    LOADING,
    SUCCESS,
    ERROR
}
```

Sealed classes are best for situations where you need to define a fixed set of possible states and want to ensure that your code is correct and easy to understand.

Data classes are better for situations where you just need to hold simple data and don’t need any complex logic.

Pro/Con Data Classes

- Pros
  - One of the main advantages of data classes is that they are **concise and easy to use**. Data classes are especially useful when you just need to hold a simple data structure and don’t need any complex logic. They allow you to define the data you need in a single line of code and provide automatically generated methods for accessing and modifying that data.
  - Another advantage of data classes is that they can be **more flexible than sealed classes**. Because data classes don’t have any restrictions on where they can be subclassed, you can define them in one file and use them in another without any problems. This can be useful if you need to share data between different parts of your codebase.
- Cons:
  - However, data classes also have some drawbacks. One of the main drawbacks is that they **don’t provide the same level of type safety** as sealed classes. Because data classes can be subclassed anywhere, it’s possible for other code to define additional subclasses that you aren’t aware of. This can make it harder to reason about your code and can increase the risk of bugs.

Optional Reading: More on Enums

- https://blog.logrocket.com/kotlin-enum-classes-complete-guide/
- https://kotlinlang.org/docs/enum-classes.html

Routes as Sealed Class

- Now, let's use a sealed class to define our routes.
- We can put this in our Router.kt file or in a separate file
- Passing parameters to a Route when using a sealed class turns out to be quite complicated syntactically. We're going to use a little workaround (defining a helper function).
  - This approach has a slight redundancy since the base route name must be repeated twice each time, but at least they are both in one place and easy to use elsewhere in the code. ⏪
  - Note that for arguments, we use string interpolation (with a preceding '/') to create the actual route string to use

```
sealed class Routes(val route:String)  {
    object Main : Routes("MainScreenRoute")
    object About : Routes("AboutScreenRoute/{name}") {
        fun go(name: String) = "AboutScreenRoute/$name"
    }
    object Contact: Routes("ContactScreenRoute/{name}/{location}") {
        fun go(name: String, location: String) = "ContactScreenRoute/$name/$location"
    }
}
```

- https://medium.com/google-developer-experts/navigating-in-jetpack-compose-78c78d365c6a
- https://hey-agrawal.medium.com/navigation-in-jetpack-compose-android-4894de51d104

Routes as Sealed Class

- To use these sealed classes in defining the routes, just replace with the name of the class and .route
    - The treatment of the arguments remains unchanged.

```
composable(Routes.Main.route) { MainScreen() }
composable(Routes.About.route,
        enterTransition = { fadeIn() + expandIn() },
        exitTransition = { ExitTransition.None }) {
        AboutScreen(
            it.arguments?.getString("name") ?: ""
        )
}
```

- To navigate to these routes, use the name of the sealed class and .route or .go(params) as appropriate

```
Button(onClick = { navController.navigate(Routes.About.go("Jane")) }) {
        Text("About Us")
}
Button(onClick = { navController.navigate(Routes.Main.route) }) {
        Text("Go Home")
}
Button(onClick = { navController.navigate(Routes.Contact.go("Akira","Japan")) }) {
        Text("Contact Us")
}
```