Application Development II

420-5A6-AB

Instructor: Talib Hussain

Day 14-15:

Navigation

Objectives

- Navigation
  - NavHost/NavController for defining and using routes
  - Provider Pattern: CompositionLocalProvider
  - "Back" with popStackBack and navigateUp
  - Navigate with arguments
  - Navigation Animations

Multi-Screen Navigation

- So far, we've had just a single "screen" in our apps.

- In a typical app, we will often want to define multiple different screens to serve different purposes
  - Display a Profile
  - Product Ordering Screen
  - Checkout Screen
  - Information Pages
  - Etc.

- While we can use conditional rendering to make a given screen very dynamic, defining different screens is a common approach

- We want to be able to programmatically move from one screen to another

- We want the user to be able to use the system navigation (e.g., "back" button) to move between screens too.

- https://developer.android.com/jetpack/compose/navigation

- https://medium.com/google-developer-experts/navigating-in-jetpack-compose-78c78d365c6a

Initial Configuration

- First, you need to add the navigation dependency to your app/build.gradle.kts file.

- Insert the following line in the dependencies section
  - implementation("androidx.navigation:navigation-compose:2.7.2")
    - Latest release version information is available at:
      https://developer.android.com/jetpack/androidx/releases/navigation
  - Note: There is a different navigation package that is not for Compose. Be careful not to let the IDE add that dependency for you (looks very similar but does not have the word "compose" in it)

- Dependency 'androidx.navigation:navigation-compose:2.7.2' requires libraries and applications that depend on it to compile against version 34 or later of the Android APIs.
  - Your app may currently be compiled against android-33.

→In build.gradle.kts, change compileSdk to 34
  and targetSdk in build.gradle.kts to 34

→"Sync Now" and rebuild

→You may need to install a new emulator device that uses API level 34

Compose Navigation

- In Compose, the 3 main parts of Navigation are the NavController, NavGraph, and NavHost.
- The NavController is always associated with a single NavHost composable.
- The NavHost acts as a container and is responsible for displaying the current destination of the graph.
    - As you navigate between composables, the content of the NavHost is automatically recomposed.
    - It also links the NavController with a NavGraph (navigation graph)
- The NavGraph that maps out the composable destinations to navigate between.
    - It is essentially a collection of fetchable destinations.

```
@Composable
public fun NavHost(
    navController: NavHostController,
    startDestination: String,
    modifier: Modifier = Modifier,
    contentAlignment: Alignment = Alignme
    route: String? = null,
    enterTransition: (AnimatedContentTran
        { fadeIn(animationSpec = tween( d
    exitTransition: (AnimatedContentTrans
        { fadeOut(animationSpec = tween(
    popEnterTransition: (AnimatedContentT
        enterTransition,
    popExitTransition: (AnimatedContentTr
        exitTransition,
    builder: NavGraphBuilder.() -> Unit
```

Router: NavHost

- Compose provides a specific composable called NavHost that defines all the routes for your app

- A NavHost accepts two main parameters:
  - A NavController
  - Initial route to use

- A NavController is created using a special state function:
  - val navController = rememberNavController()

- A NavHost also defines one or more routes using the composable() function
  - We'll do this in the "content" part

```
NavHost(
    navController,
    startDestination,
    modifier,
) {

    content

}
```

Routes

```
composable( route ) {
    content
}
```

- To enable navigation to a specific screen, we can define a "route" for that screen
  - Similar to navigation in React.
- A route is a string corresponding to the name of a route. This can be any unique string.
  - E.g., "MainScreenRoute"
- Navigation Compose provides the NavGraphBuilder.composable extension function to easily add individual composable destinations to the navigation graph and define the necessary navigation information.
- For a given route, we identify the composable to use when navigating to that route

  composable("MainScreenRoute") {MainScreen()}
  - This calls MainScreen() composable for a route indicated by the string "MainScreenRoute"

Router

- We can define our NavHost directly in our MainActivity.kt, or we can create a separate Router composable to encapsulate the routing logic.

```
@Composable
fun Router() {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = "MainScreenRoute") {
        composable("MainScreenRoute") { MainScreen() }
        composable("AboutScreenRoute") { AboutScreen() }
    }
}
```

Navigating to a Route

`navController.navigate( route )`

- To navigate to a route, your need to call navController.navigate() passing in the route name.

  Button(onClick = { navController.navigate("AboutScreenRoute") }) {

       Text("About Us")

  }

- But, now we have a slight issue… how do we access navController lower in the hierarchy (since it is defined near the top level)

- Several approaches possible:
  - Pass navController as parameters down the tree of composables
    - Messy… This is like the issue of "prop drilling" in React…
  - Pass lambda "callback" functions down the tree
    - Actually suggested in official and other docs, and can be useful in testing since a navController wouldn't be required to test a specific composable… but still somewhat messy
    - https://medium.com/google-developer-experts/navigating-in-jetpack-compose-78c78d365c6a
    - https://developer.android.com/jetpack/compose/navigation
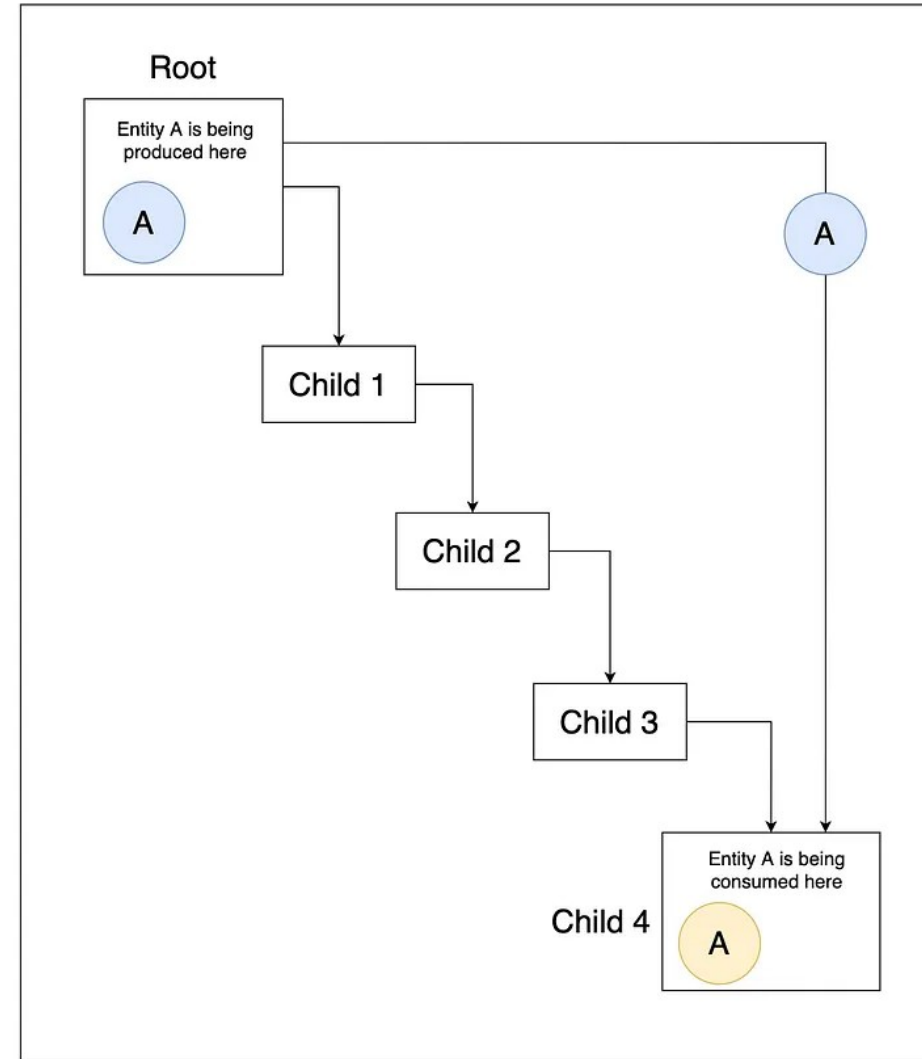 - Use the Provider pattern!

Issue with Parameters

- Passing a parameter down the component tree to the composable that needs is not always ideal
  - Each composable in the path needs to receive and pass on the parameter.
  - When the descendant is deep in the tree compared to where the state is defined, this results in unnecessary dependencies and requires a lot of extra unnecessary syntax along the way.
  - Also results in increased likelihood of introducing errors, introduces unnecessary coupling and the code becomes harder to understand and maintain
- Analogous to "prop drilling" from React.
- In example to the right, the root node produces an entity A, which is being consumed at the 4th level child, Child 4. A is not a singleton, and since its instance cannot be created from anywhere apart from root node, the created instance has to be passed down the tree up to Child 4. In this case, instance of A is being unnecessarily held by Children 1 through 3, which is useless for these nodes.
  - https://betterprogramming.pub/provider-pattern-in-jetpack-compose-bb4f4e27185e
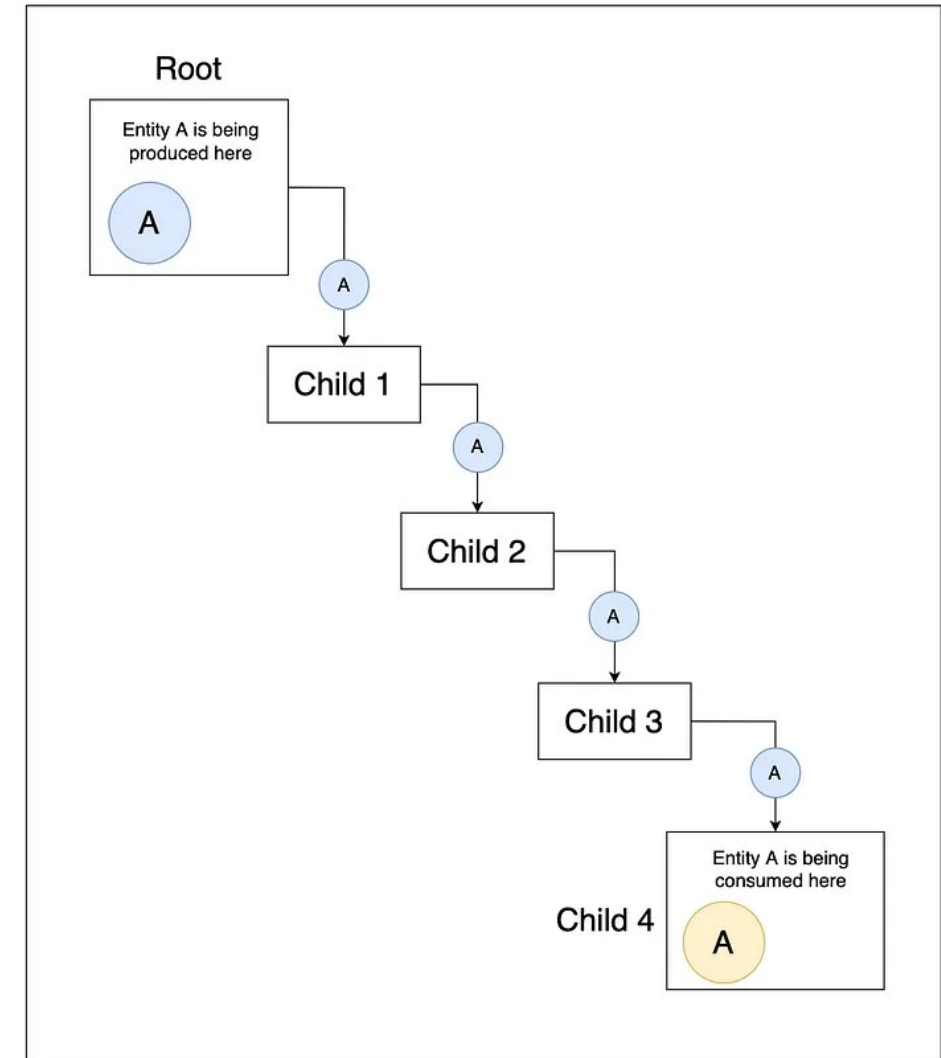
Provider Pattern

- Continuing example: Ideally, we'd like to somehow access A directly from Child 4

- The Provider Pattern gives this to us.

- In Compose, this is done using CompositionLocalProvider

- CompositionLocalProvider can provide a reference to some object from the higher level in the tree, and any child can access the provided value directly.

- This uses a CompositionLocal internally which is scoped to a sub-tree.
  - If we put this at the root level of the UI tree, it covers the entire tree underneath, thus behaving as a Global Provider.

- A CompositionLocalProvider can be put anywhere in the tree, and multiple providers can be used.

CompositionLocal & CompositionLocalProvider

- CompositionLocal is a tool for passing data down through the tree of composables implicitly
    - It uses the Provider pattern
    - https://developer.android.com/jetpack/compose/compositionlocal
    - https://medium.com/@gustavohenriques/compositional-locals-for-navigation-in-jetpack-compose-c75a261bd7ac

- First, create a top-level variable (outside of a Composable!) that will be used to access the provided value

    ```
    val LocalNavController = compositionLocalOf<NavHostController> { error("No NavController found!") }
    ```

- Next, wrap the NavHost with a CompositionLocalProvider that "provides" the navController value

    ```
    CompositionLocalProvider(LocalNavController provides navController) {
        NavHost(navController = navController, startDestination = "MainScreenRoute") {
            composable("MainScreenRoute") { MainScreen() }
            composable("AboutScreenRoute") { AboutScreen() }
        }
    }
    ```

- Finally, in a child composable, access the navController value using LocalNavController.current and then use that value in appropriate handler calls.

    ```
    val navController = LocalNavController.current
    Button(onClick = { navController.navigate("AboutScreenRoute") })
    ```

```kotlin
package com.example.kotlinwithcompose.screens

import androidx.compose.runtime.Composable
import androidx.compose.runtime.CompositionLocalProvider
import androidx.compose.runtime.compositionLocalOf
import androidx.navigation.NavController
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController

val LocalNavController = compositionLocalOf<NavController> { error("No NavController found!") }
@Composable
fun Router() {
    val navController = rememberNavController()

    CompositionLocalProvider(LocalNavController provides navController) {
        NavHost(navController = navController, startDestination = "MainScreenRoute") {
            composable("MainScreenRoute") { MainScreen() }
            composable("AboutScreenRoute") { AboutScreen() }
        }
    }
}
```

"Back"

`navController.popBackStack(` route `,` inclusive `)`

- In addition to navigating to a specific route, we can use the navController to go back to an earlier route on the navigation stack.

- navController.popBackStack()
  - This command will navigate to the previous screen that was shown.

- navController.popBackStack("MainScreenRoute", false)
  - This command will pop all screens on the stack until it reaches the indicated destination route (here: MainScreenRoute). If the second parameter is false, then it will show that screen. If it is true, then it will pop that too.

- We can also use navigateUp() to go back
  - The main difference is that if we pop off the last screen, then it will try to go to the app that called this app, if that was the case.

- If we want to hide the back button when there is nothing to go back to, then we can check the stack
  - canNavigateBack = navController.previousBackStackEntry != null,

Try It!

- Create 3 screens: MainScreen, AboutScreen, ContactScreen
- Create an appropriate Router with NavHost that defines routes to those screens
- Create buttons on those screens that let you navigate among the various screens, as well as appropriate back buttons.
  - Don't show the back button if there is nothing to go back to

Navigate with Arguments

- Navigation Compose supports passing arguments between composable destinations
  - https://developer.android.com/jetpack/compose/navigation
- For example, let's add a parameter to the AboutScreen composable
  @Composable
  fun AboutScreen(name: String) {
- To pass this new parameter, you need to add argument placeholders to your route id
  composable("AboutScreenRoute/{name}")
  - This adds an argument called 'name' to the route.
- This parameter is accessed via the trailing lambda for the composable() function, which provides a single parameter
  - Recall: We can just use `it`
  - This parameter has an `arguments` value that we can check. It is a nullable type, so the null must be handled as appropriate.
  - composable("AboutScreenRoute/{name}") { AboutScreen(it.arguments?.getString("name") ?: "") }
- Finally, to navigate, pass the desired value in as part of the route id
  navController.navigate("AboutScreenRoute/Jane")

Simple data only as arguments...

- It is strongly advised not to pass complex objects when navigating
- Instead, pass the minimum necessary information, such as a unique ID
- Complex objects should be stored as data in a single source of truth, such as a data layer
- Once you land on your destination after navigating, you can then load the required information from the single source of truth using the passed ID
- Without additional setup, you can pass string, int, long, float, bool, as well as arrays of those types
  - Note: Strings should not contain '/' or the app will crash due to an unexpected route.

Multiple arguments

- If you want to pass multiple arguments then you can define a route with multiple placeholders separated by slashes ("/").
  - https://www.linkedin.com/pulse/pass-arguments-destinations-jetpack-compose-sagar-malhotra/

```
composable("ContactScreenRoute/{name}/{location}") {
        val name = it.arguments?.getString("name") ?: ""
        val location = it.arguments?.getString("location") ?: ""

        ContactScreen(name, location)
    }
```

Navigation Animations

- You can animate the transition between composables when navigating between them.
- NavHost has enterTransition and exitTransition parameters that can specify default animations

```
NavHost(
    navController = navController, startDestination = "landing",
    enterTransition = { EnterTransition.None },
    exitTransition = { fadeout() }
)
```

- Each route can also have its own enter and/or exit transitions

```
composable("AboutScreenRoute/{name}",
            enterTransition = {fadeIn() + expandIn()},
            exitTransition = { ExitTransition.None}) { AboutScreen(it.arguments?.getString("name") ?: "") }
```

- There are several types of transition animations
    - fadeIn/Out, slideIn/Out, ScaleIn/Out, expandIn/shrinkOut, and more
    - Each has various parameters
    - https://developer.android.com/jetpack/compose/animation/composables-modifiers#enter-exit-transition
    - https://medium.com/@nitheeshag/navigation-in-jetpack-compose-with-animations-724037d7b119

Try It!

- Add parameters and animations to your routes

Next Steps

- Shared Layouts
- Nested Navigation
- Risk Management