Android Mobile Application Development
Week 6: Navigation

Navigation and Flow

https://m2.material.io/design/navigation/understanding-navigation.html
Understanding the different types of navigation available on Android is crucial to creating an app that's easy and intuitive to use. Keep in mind the goal of your app, its tasks, content, and target user.
There are three navigational directions:

- Forward navigation – moving between screens to complete a task
  - Downward in an app's hierarchy, from parent (higher level of hierarchy) to child (lower level of hierarchy)
    - We will look at how to implement this type of navigation using recycler views on Thursday
  - Sequentially through a flow, or sequence of screens (cards)
  - Directly from one screen to another (buttons or search)
  - Gestural navigation
    - Swipe or drag
- Reverse navigation – moving backwards through screens
  - reverse chronological navigation using the back button (within one app or across different apps)
  - hierarchically with up navigation from a child screen to their logical parent screen (within an app)
- Lateral navigation – moving between screens at the same level of hierarchy.
  - An app's primary navigation component should provide access to all destinations at the top level of its hierarchy.
    - Apps with two or more top-level destinations should provide lateral navigation

Lateral Navigation
Navigation drawer https://m2.material.io/design/components/navigation-drawer.html

- Apps with 5+ top-level destinations or two+ levels of navigation hierarchy
- Quick navigation between unrelated destinations
- Types
  - Standard – allows interaction with both screen content and the drawer at the same time. They can be used on tablet and desktop, but they aren't suitable for mobile due to limited screen size.
  - Modal – elevated above most of the app's UI and don't affect the screen's layout grid. They block interaction with the rest of an app's content. Used on mobile where screen space is limited.
  - Bottom – modal drawers used with bottom app bars that are anchored to the bottom of the screen instead of the left edge.
- Creates history for the back button
- Reduces visibility for infrequently visited destinations
- Supports nested navigation for a deep navigational structure
- As phones have gotten larger the nav drawer hamburger menu is harder to access as 49% of the time users use their right thumb to interact with their device. Using bottom navigation increases reachability and the use of your app's core features.

Bottom navigation bar https://m2.material.io/design/components/bottom-navigation.html
Apps with 3-5 top level destinations (no "more" …)

- Frequent switching between views
- Mobile or tablet use
- Should usually persistent across screens to provide consistency
- Must include icons. Also provide text labels if needed to clarify the meaning
- When navigating to a destination's top-level screen any prior user interactions and screen states are reset (although you could implement saving state)
- Ergonomic, easy to switch between views
- Cross-fade animation is suggested with bottom navigation
- Not suggested along with a navigation drawer or tabs
- Doesn't create history for the back button

Tabs https://m2.material.io/components/tabs

- Organize 2+ sets of data that are related, same level of hierarchy, siblings
- Easily switch between a few different categories of content
- Tap or swipe to navigate tabs
- Tabs exist inside the same parent screen so tabs don't create history for the system back button
- Also provides additional lateral navigation when paired with a top-level navigation component

Reverse Navigation
https://m2.material.io/design/navigation/understanding-navigation.html#reverse-navigation
Back:
When your app is launched, a new task is created and becomes the base destination of the app's back stack.
The top of the stack is the current screen, and the previous destinations in the stack represent the navigation history. The back stack always has the start destination of the app at the bottom of the stack. Operations that change the back stack always operate on the top of the stack, either by pushing a new destination onto the top of the stack or popping the top-most destination off the stack. Navigating to a destination pushes that destination on top of the stack.
The Navigation component manages all of your back stack ordering for you, though you can also choose to manage the back stack yourself.

The Back button is in the system navigation bar at the bottom of the screen and is used to navigate in reverse-chronological order through the history of screens the user has recently worked with. When you tap the Back button, the current destination is popped off the top of the back stack, and you then navigate to the previous destination. It also handles the following:

- Dismisses floating windows such as dialogs or pop-ups
- Dismisses contextual action bars
- Removes the highlight from selected items
- Hides on screen keyboard

Upward:
Upward navigation is a way for the user to navigate to the logical parent screen in the app's hierarchy. This differs from back navigation because it is not based on the user's history or path, but a defined parent activity. Up navigation will therefore never exit the app.

App Navigation components

App bar

The App Bar is a consistent navigation element that is standard throughout modern Android applications. https://www.material.io/components/app-bars-top

The top app bar provides content and actions related to the current screen. It's used for branding, screen titles, navigation, and actions.

The App Bar can consist of:

- Consistent navigation (including navigation drawer)
- An application icon
- Up navigation to logical parent
- An application or activity-specific title
- Primary action icons for an activity
- Overflow menu

In Compose Scaffold provides a slot for topBar which takes a TopAppBar composable.

Menus

https://m2.material.io/components/menus

Compose provides a DropdownMenu composable that can be used to display a list of choices in a menu. Menus can be displayed based on an event or in the options menu in the app bar.

Floating Action Button (FAB)

https://m2.material.io/components/buttons-floating-action-button

- Used to represent the most common or primary action in the screen
    - Add
    - Share
    - Play/pause
- The action associated with the FAB must be for the whole view, not just one part of it like one row
- There should usually only be one FAB on a screen

In Compose Scaffold provides a slot for floatingActionButton which takes a FloatingActionButton composable.

Navigation Architecture Component

https://developer.android.com/guide/navigation

The Navigation Architecture Component is an architectural component in Jetpack that helps us implement navigation in our application. The Navigation component is used for navigation ranging from simple button clicks to more complex patterns, such as app bars, the navigation drawer, and bottom navigation.

The Navigation component consists of three key parts:

- Navigation graph that includes all the destinations in your app
    - The route is a string value that uniquely identifies the destination within the context of the current navigation controller
    - The destination is a composable or activity to be called when the navigation is performed

    - Each destination visited is pushed on the back stack and then as the user uses the back button for reverse navigation the destinations are popped off the stack. The navigation controller manages this navigation stack.
- Navigation Host is an empty container where destinations are swapped in and out as a user navigates through your app.
    - The `NavHost` links the `NavController` with a navigation graph that specifies the composable destinations that you should be able to navigate between. As you navigate between composables, the content of the `NavHost` is automatically recomposed.
- Navigation Controller that manages app navigation within a `NavHost`. The `NavController` orchestrates the swapping of destination content in the `NavHost` as users move throughout your app.
    - Each `NavHost` has its own corresponding `NavController`
    - It is stateful and keeps track of the back stack of composables that make up the screens in your app and the state of each screen.

The Navigation component handles all navigation, up and back navigation, and even provides basic animations and transitions for navigating. It also supports deep linking and nested navigation.

Navigation in Compose
https://developer.android.com/jetpack/compose/navigation
To use the navigation component in compose you need to add the following dependency in your app module's build.gradle file:

```
implementation 'androidx.navigation:navigation-compose:2.5.3'
```

NavController
The first step in adding navigation is creating a navigation controller to manage the navigation stack. It is a stateful object so the back stack is maintained through recomposition.

- Create the `NavController` in your composable hierarchy where all composables that need to reference it have access to it
    - use the `rememberNavController()` method to create a `NavController`

NavHost
A navigation host is needed to define your navigation. It requires the following parameters:

- navController: a `NavController` object that is associated with a single NavHost
- startDestination: the route where navigation will start
- builder: builds a navigation graph with all the destinations defined as routes
    - it often uses the trailing lambda syntax so you'll see it passed as a trailing lambda that's pulled out of the parentheses and inside the body of the function

Navigation Graph
The navigation graph is comprised of destinations that are declared in the form of routes.

- The `composable()` method is used to add destinations to the graph
    - The route is a string value that uniquely identifies the destination
    - The destination is a composable or activity to be called when the navigation is performed

```
val navController = rememberNavController()
```

```
NavHost(navController = navController, startDestination = "home") {
     composable("home") {
          Home()
     }
     composable("customers") {
        Customers()
    }
     composable("purchases") {
        Purchases()
    }
}
```

Navigating to Destinations

To perform navigation you use the `navigate()` method in the NavController class, passing it the route you want to navigate to.

- other navigation options are available such as `popUpTo()` which allows you to pop additional destinations off of the back stack

```
navController.navigate("purchases")
```

Passing arguments

You can make navigation routing dynamic by passing one or more arguments to a route. This enables displaying different information based on the different arguments provided. To pass arguments during navigation:

- Add the arguments to the route
    - To pass the argument alongside your route when navigating, you need to append them together, following a pattern: `"route/{argument}"`
        - / marks a new path
        - {} is an argument placeholder
        - Additional parameters are built extending the pattern with additional /{}
- Make the composable aware that it should accept parameters by defining the arguments parameter
    - The arguments parameter of the `composable()` method accepts a list of arguments of `NamedNavArgument` objects
        - Use the `navArgument()` method which takes a name and a type to create the `NamedNavArgument` objects
        - By default all arguments are parsed as strings
        - Note that matching the key name exactly is needed
- Retrieve the passed arguments from `NavBackStackEntry` that is available in the lambda of the `composable()` function.
    - The `NavBackStackEntry` class holds the information on the current route and the passed arguments of an entry in the back stack.
    - Use the get methods to retrieve the exact argument you need from the `navBackStackEntry.arguments` list
    - Note again that matching the key name exactly is needed

```
composable("customers/{customerName}", arguments =
listOf(navArgument("customerName") { type = NavType.StringType }))
{
     backStackEntry ->
     Customers(navController = navController,
     backStackEntry.arguments?.getString("customerName"))
}
```

- Pass the data required for the arguments during navigation
    - Reminder that $ is used to escape variables

```
navController.navigate("customers/$customerName")
```

You should avoid passing the entire navigation controller to other composables. Instead, you should always provide callbacks that define the exact navigation actions you wish to trigger. This is done by passing the needed calls to navigate() as a callback to composables so they can be used as event handlers for the appropriate events. This follows the state hoisting pattern we've been using for all our variables that store state.
The benefit of using a NavHost to handle your app's navigation is that navigation logic is kept in once place that is separate from the UI that define our screens.

Book VM nav
We'll continue with our Book app to add some navigation so when the user clicks on a book the app navigates to a book detail screen. We'll keep it pretty simple but you can imagine showing more details, a button for purchasing the book, sharing it with a friend, etc.

Add the dependency in your app module's `build.gradle` file and sync.
```
implementation 'androidx.navigation:navigation-compose:2.5.3'
```

Book Detail
We need to create our book detail screen.

Select the ui.compose package and add a new Kotlin file called BookDetailScreen.
File | New | Kotlin file
Name: BookDetailScreen
Kind: File
In BookDetailScreen.kt define a composable that will be responsible for drawing the book detail screen. Our function takes in two parameters, the book name and author, both Strings. They are nullable because they'll need to be later when we pass in data.
We'll keep this simple and use a similar layout as we have in our book list.
For now we'll use parameter defaults to hard code some data in this screen.

```
@Composable
fun BookDetailScreen(bookName: String? = "Book", bookAuthor: String?
= "Author"){
    Surface(
    ) {
        Card(
            elevation = 4.dp,
```

```
            shape = RoundedCornerShape(10.dp),
            backgroundColor = MaterialTheme.colors.primary,
            contentColor = MaterialTheme.colors.onPrimary,
            border = BorderStroke(2.dp, color =
MaterialTheme.colors.primaryVariant),
            modifier = Modifier
                .padding(8.dp)
                .fillMaxWidth()
        ) {
            Column(
                modifier = Modifier.padding(16.dp)
            ) {
                bookName?.let { Text(text = it, style =
MaterialTheme.typography.h6) }
                bookAuthor?.let { Text(text = it, style =
MaterialTheme.typography.body1) }
            }

        }
    }
}
```

You can add a preview as well if you'd like (just make sure to use a different name than other preview functions).

```
@Preview(showBackground = true)
@Composable
fun DetailPreview() {
    BookTheme {
        BookDetailScreen()
    }
}
```

Navigation controller

In MainActivity in the ScreenSetup composable create a NavController by using the `rememberNavController()` method. Remember allows the navigation controller to keep the state of the navigation stack across recompositions.

```
val navController = rememberNavController()
```

NavHost

Then we need a NavHost to link the nav controller and the destinations/routes we want to navigate to. We use the builder parameter as a trailing lambda parameter to build the navigation graph. In the composable() function we assign a route string to the route parameter and a composable function for the desired destination to the content parameter as a trailing lambda parameter.

We'll create two routes and make the startDestination books which has BookScreen() as its destination.

```
NavHost(navController = navController, startDestination = "books")
{
    composable("books"){ BookScreen()}

    composable("bookDetail"){BookDetailScreen()}
}
```

With BookScreen() being our start destination we no longer need to call it directly in ScreenSetup().

```
@Composable
fun ScreenSetup(){
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination =
"books")
    {
        composable("books"){ BookScreen()}

        composable("bookDetail"){BookDetailScreen()}
    }

}
```

Since we want to navigate to BookDetailScreen from BookScreen when the user taps on a book, we need to pass a callback into BookScreen that will navigate to BookDetailScreen.

Update BookScreen() to take as a parameter a callback that will navigate to the book detail screen.

```
fun BookScreen(onNavigateToBookDetail:() -> Unit) {...}
```

Update BookItem() to take as a parameter an onItemClick callback.

```
fun BookItem(item: Book, context: Context, deleteBook: (Book) ->
Unit, onItemClick: () -> Unit) {...}
```

and use onItemClick as the handler for the onClick event instead of the Toast.

```
            .combinedClickable(
                onClick = onItemClick,
                onLongClick = { showDeleteDialog = true }
            )
```

In BookScreen() update the items loop in the LazyColumn to pass the callback to BookItem()

```
BookItem(item = book, context, {viewModel.delete(it)}, onItemClick =
onNavigateToBookDetail)
```

If you have a Preview composable you'll need to update it with a parameter.

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    BookTheme {
        BookScreen({})
```

```
    }
}
```

In MainActivity update ScreenSetup() for the NavHost to pass a callback to BookScreen that will navigate to the book detail screen using the navigate() method which takes a route as a parameter. Spelling becomes ***very*** important here, the string for the route must exactly match the name of the route in your navigation graph.

```
NavHost(navController = navController, startDestination = "books")
{
    composable("books"){ BookScreen(onNavigateToBookDetail =
{navController.navigate("bookDetail")})}

    composable("bookDetail"){BookDetailScreen()}
}
```

Now run the app, add a book and click on it. It should navigate to the book detail screen showing the book detail screen we set up with defaults.

Navigate with arguments
Now we want to update our app so the book name and author are passed in the navigation to BookDetailScreen.
In BookScreen update the BookScreen() onNavigateToBookDetail parameter lambda function to accept two Strings for the book name and author.

```
fun BookScreen(onNavigateToBookDetail:(bookName: String, author:
String) -> Unit) {...}
```

Since the onNavigateToBookDetail lambda expression is passed to BookItem() we need to update the onItemClick signature as well.

```
fun BookItem(item: Book, context: Context, deleteBook: (Book) ->
Unit, onItemClick: (bookName: String, author: String) -> Unit) {...}
```

In the onClick event handler we can now pass the book name and author. Remember to wrap the lambda expression in {}

```
onClick = {onItemClick(item.bookName, item.author)},
```

If you have a Preview composable you'll need to update the parameters. You can use underscore for unused variables.

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    BookTheme {
        BookScreen({ _, _ -> })
    }
}
```

In MainActivity now we have to update our navigation graph.

Arguments allow navigation routing to be dynamic by passing one or more arguments to a route. It enables displaying different information based on the different arguments provided.

There are two parts to this – update the route that is the landing destination to require arguments it needs, and the starting destination needs to pass the data to those arguments during navigation.

First we'll update the landing destination. Update the bookDetail route to include two parameters. To pass the argument alongside your route when navigating, you need to append them together, following a pattern: `"route/{argument}"`. Multiple parameters build on that pattern: `"route/{argument1}/{argument2}"`.

Define the arguments parameter so the composable is aware that it should accept parameters. The arguments parameter accepts a list of arguments of NamedNavArgument objects that you create using the navArgument() method which takes a name and a type.

```
composable("bookDetail/{bookName}/{author}", arguments =
listOf(navArgument("bookName"){type= NavType.StringType},
navArgument("author"){type= NavType.StringType})) {}
```

We also need to retrieve the passed argument value so we can pass it to BookDetailScreen(). In Compose Navigation, each `NavHost` composable function has access to the current `NavBackStackEntry` - a class which holds the information on the current route and passed arguments of an entry in the back stack. You can use this to get the required `arguments` list from `navBackStackEntry` and then use the get methods to retrieve the exact argument you need. Note again that matching the key name exactly is needed.

```
composable("bookDetail/{bookName}/{author}", arguments =
listOf(navArgument("bookName"){type= NavType.StringType},
navArgument("author"){type= NavType.StringType}))
{
    backStackEntry ->
    BookDetailScreen(backStackEntry.arguments?.getString("bookName"),
backStackEntry.arguments?.getString("author"))
}
```

Now for the starting destination. In the books route update the lambda function assigned to onNavigateToBookDetail in BookScreen to reflect the two parameters it now expects. We also have to change the route passed to the navigate() function to match the bookDetail route. We use the $ to escape the variable names so they're evaluated.

```
BookScreen(onNavigateToBookDetail = {bookName, author ->
navController.navigate("bookDetail/$bookName/$author")})}
```

Now when you run the app add a book and click on it and it should navigate to the book detail screen showing the book name and author of the book you clicked on.