MODUL 9
Building User Interfaces using Jetpack Compose

THEME DESCRIPTION

In this module, students will learn how to use Jetpack Compose to create user interfaces using Kotlin code, how Compose revolutionized the way we built user interfaces, and how they can translate existing applications to Jetpack Compose.

WEEKLY LEARNING OUTCOME (SUB-LEARNING OUTCOME)

Students will be familiar with the most common UI elements in Compose and how to handle user actions.

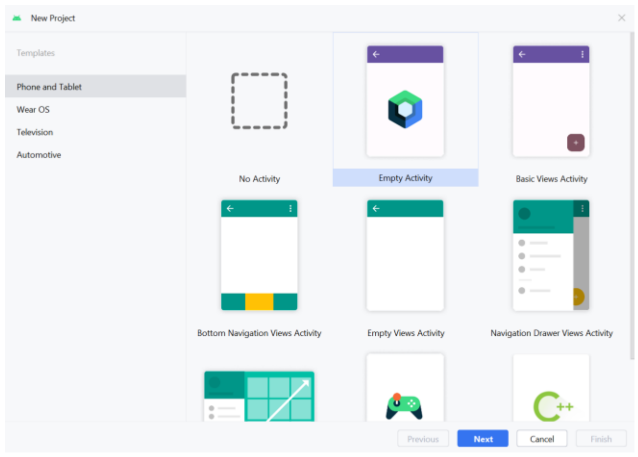
TOOLS/SOFTWARE USED

- Android Studio

PRACTICAL STEPS

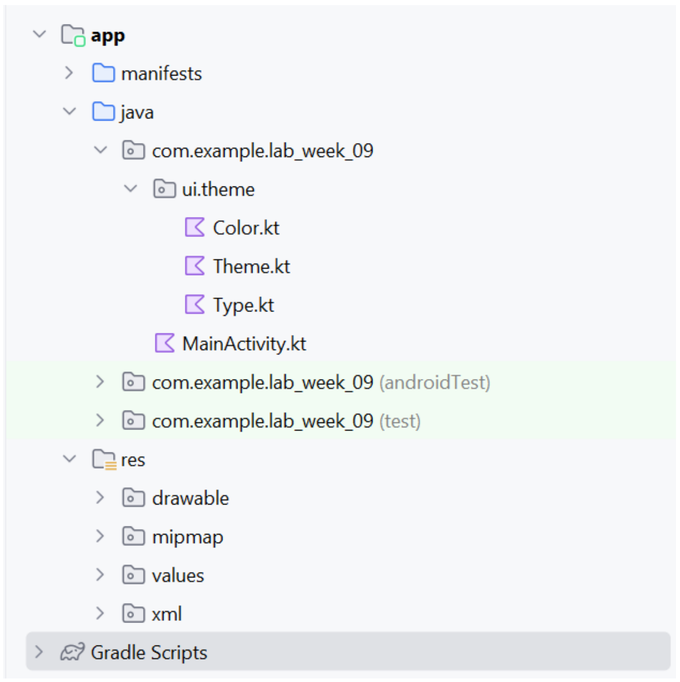
Part 1 - Building a Simple Jetpack Compose UI

1. Open Android Studio and click **New Project.**
2. Now instead of choosing Empty Views Activity**,** choose the **Empty Activity** to start with. **Empty Activity** is a **Starting Template** ready to be used specifically for **Jetpack Compose Project**.



3. Name your project **"LAB_WEEK_09"**.
4. Set the minimum SDK to **"API 24: Android 7.0 (Nougat)"**.
5. Click **Finish**, and let your android application build itself.

6. In this part, we will be focusing on how we can **make a simple Jetpack Compose application** in Android. First, you may notice that the **File Structure** is a little bit **Different** from before.



7. The **“layout” Folder** is completely gone due to the way **Jetpack Compose** is building the application **UI** and there’s an addition of the **“ui.theme” Folder** inside the package, which replaces the old way of providing **Styles** and **Themes** for the application.
8. Before we continue, update your **Strings.xml** to the code below.

```
<resources>
   <string name="app_name">LAB_WEEK_09</string>

   <string name="enter_item">Enter a name</string>
   <string name="button_click">Submit</string>
   <string name="button_navigate">Finish</string>
</resources>
```

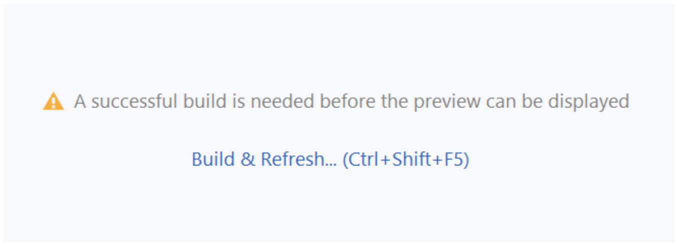
9. Also update the **Material 3 Dependency** version to the **1.1.2** version to fix any compatibility issues.

```
//implementation("androidx.compose.material3:material3")
implementation("androidx.compose.material3:material3:1.1.2")
```

10. Now let’s start by making a simple **UI** that consists of just a **TextView**. Instead of modifying the **XML files**, we can now do it all inside our **Kotlin files.** In your **MainActivity.kt**, ignore what’s already predefined from the template itself, update it to the code below.

```
//Previously we extend AppCompatActivity,
//now we extend ComponentActivity
class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       //Here, we use setContent instead of setContentView
       setContent {
           //Here, we wrap our content with the theme
           //You can check out the LAB_WEEK_09Theme inside Theme.kt
           LAB_WEEK_09Theme {
               // A surface container using the 'background' color from the
theme
               Surface(
                   //We use Modifier.fillMaxSize() to make the surface fill
the whole screen
                   modifier = Modifier.fillMaxSize(),
                   //We use MaterialTheme.colorScheme.background to get the
background color
                   //and set it as the color of the surface
                   color = MaterialTheme.colorScheme.background
               ) {
                   //Here, we call the Home composable
                   Home()
               }
           }
       }
   }
}

//Here, instead of defining it in an XML file,
//we create a composable function called Home
//@Preview is used to show a preview of the composable
@Preview(showBackground = true)
//@Composable is used to tell the compiler that this is a composable
function
//It's a way of defining a composable
@Composable
fun Home() {
   //Here, we use Column to display a list of items vertically
   //You can also use Row to display a list of items horizontally
   Column {
       //Here, we use Text to display a text
       Text(
           //We use stringResource to get the string from Strings.xml
           //and set it as the text
           text = stringResource(id = R.string.list_title)
       )
   }
}
```

11. Here are few tips summarized from the code above:
    - You now define your **Layouts** inside your **Kotlin File.** Previously, we used the **Imperative Approach**, which solely depends on the **Layout Folder**. Now with the removal of the folder, we use the **Declarative Approach**.
    - Instead of extending **AppCompatActivity**, now you extend **ComponentActivity.**
    - Instead of using **setContentView** to set your view, now you use **setContent.**
    - Inside **setContent**, a **Surface** is provided as a container for the activity which contains all the **Composable** and all its **Themes.**
    - Instead of a **Layout File**, now it's called **Composable**. You can consider **Composable** as **Component** much like in **React JS**. Yes, it's **Component-based.**
    - You can create a **Composable** using the `@Composable` annotation followed up with a **Function** that defines your views.
    - One of the views that you may know is **TextView**, now it's called **Text**, which you can call inside the **Composable Function**.
    - To wrap your **Views**, instead of using **ViewGroup** like **LinearLayout**, you now use **Row** to group items horizontally and **Column** to group items vertically.
    - To get a string resource, instead of calling from the **View's Attribute**, now you use **stringResource**.
    - To view the layout and what it looks like, you now have to **Build** it first, then it'll show you the **Preview** of the **Activity** layout. Don't forget to also add the `@Preview` annotation before your **Composable Function** to enable preview.


⚠ A successful build is needed before the preview can be displayed

Build & Refresh... (Ctrl+Shift+F5)

12. Still inside **MainActivity.kt**, update your **Home Composable** to the code below.

```
//Notice that we remove the @Preview annotation
//this is because we're passing a parameter into the composable
//When the compiler tries to build the preview,
//it doesn't know what to pass into the composable
//So, we create another composable function called PreviewHome
//and we pass the list as a parameter
@Composable
fun Home(
   //Here, we define a parameter called items
   items: List<String>,
) {
   //Here, we use LazyColumn to lazily display a list of items horizontally
   //LazyColumn is more efficient than Column
   //because it only composes and lays out the currently visible items
   //much like a RecyclerView
   //You can also use LazyRow to lazily display a list of items
horizontally
   LazyColumn {
       //Here, we use item to display an item inside the LazyColumn
       item {
           Column(
               //Modifier.padding(16.dp) is used to add padding to the
Column
               //You can also use Modifier.padding(horizontal = 16.dp,
vertical = 8.dp)
               //to add padding horizontally and vertically
               //or Modifier.padding(start = 16.dp, top = 8.dp, end =
16.dp, bottom = 8.dp)
               //to add padding to each side
               modifier = Modifier.padding(16.dp).fillMaxSize(),
               //Alignment.CenterHorizontally is used to align the Column
horizontally
               //You can also use verticalArrangement = Arrangement.Center
to align the Column vertically
               horizontalAlignment = Alignment.CenterHorizontally
           ) {
               Text(text = stringResource(
                   id = R.string.enter_item)
               )
               //Here, we use TextField to display a text input field
               TextField(
                   //Set the value of the input field
                   value = "",
                   //Set the keyboard type of the input field
                   keyboardOptions = KeyboardOptions(
                       keyboardType = KeyboardType.Number
                   ),
                   //Set what happens when the value of the input field
changes
                   onValueChange = {
                   }
               )
               //Here, we use Button to display a button
               //the onClick parameter is used to set what happens when the
button is clicked
               Button(onClick = { }) {
                   //Set the text of the button
                   Text(text = stringResource(
                       id = R.string.button_click)
                   )
               }
           }
       }
       //Here, we use items to display a list of items inside the
LazyColumn
       //This is the RecyclerView replacement
       items(items) { item ->
           Column(
               modifier = Modifier.padding(vertical = 4.dp).fillMaxSize(),
               horizontalAlignment = Alignment.CenterHorizontally
               ) {
               Text(text = item)
           }
```

```
        }
    }
}

//Here, we create a preview function of the Home composable
//This function is specifically used to show a preview of the Home
composable
//This is only for development purpose
@Preview(showBackground = true)
@Composable
fun PreviewHome() {
    Home(listOf("Tanu", "Tina", "Tono"))
}
```

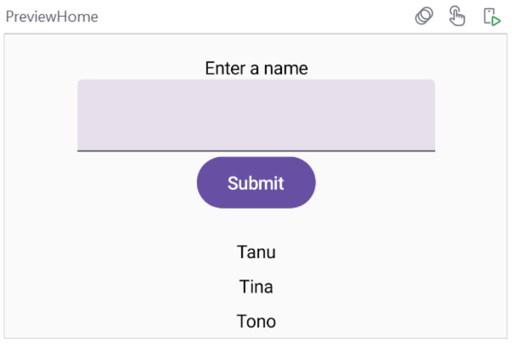
13. Also inside the **onCreate** callback, update your **Surface** to the code below.

```
Surface(
    //We use Modifier.fillMaxSize() to make the surface fill the whole
screen
    modifier = Modifier.fillMaxSize(),
    //We use MaterialTheme.colorScheme.background to get the background
color
    //and set it as the color of the surface
    color = MaterialTheme.colorScheme.background
) {
    val list = listOf("Tanu", "Tina", "Tono")
    //Here, we call the Home composable
    Home(list)
}
```

14. Here are few tips summarized from the code above:
    - You can pass parameters to your **Composable**, but the `@Preview` annotation cannot be used, as the compiler doesn't know what to fill in the parameters with, therefore you need to define another **Composable** specifically for previewing the **Layout**.
    - Instead of using **RecyclerView**, **ViewHolder, and Adapter**, it's now simplified to only using the **LazyColumn** (Vertical Layout) or **LazyRow** (Horizontal Layout).
    - Instead of using **InputText**, now it's called **TextField**.

- For any **Event Handlers** like **onClick** or **onValueChange**, you can now straight up put it within your **View.**
- Instead of defining **Styles** like **Padding** and such inside the **View's Attributes**, now you define it inside the **Modifier** object.

15. You can try previewing the **Composable** and it should look like this.



COMMIT to **GITHUB** at this point. **Commit Message: "Commit No. 1 – Building Jetpack Compose UI"**.

Part 2 - States and Adding Event Handlers

1. The application **UI** is now done, let's handle the **User Input** and the necessary **Event Handler** using **State**.
2. First, let's define a **Data Model** for the user input. The model will be used as the base of your **State**. Add the code below after the **MainActivity Class**.

```
//Declare a data class called Student
data class Student(
   var name: String
)
```

3. Next, let's divide the page content into a parent and a child. The parent will control all **Data State Changes** while the child will **Store** the **Content** of your page. Update your **Home Composable** to the code below.

```
@Composable
fun Home() {
   //Here, we create a mutable state list of Student
   //We use remember to make the list remember its value
   //This is so that the list won't be recreated when the composable
recomposes
```

```
   //We use mutableStateListOf to make the list mutable
   //This is so that we can add or remove items from the list
   //If you're still confused, this is basically the same concept as
using
   //useState in React
   val listData = remember { mutableStateListOf(
       Student("Tanu"),
       Student("Tina"),
       Student("Tono")
   )}
   //Here, we create a mutable state of Student
   //This is so that we can get the value of the input field
   var inputField = remember { mutableStateOf(Student("")) }

   //We call the HomeContent composable
   //Here, we pass:
   //listData to show the list of items inside HomeContent
   //inputField to show the input field value inside HomeContent
   //A lambda function to update the value of the inputField
   //A lambda function to add the inputField to the listData
   HomeContent(
     listData,
     inputField.value,
     { input -> inputField.value = inputField.value.copy(input) },
     {
         if (inputField.value.name.isNotBlank()) {
             listData.add(inputField.value)
             inputField.value = Student("")
         }
     }
  )
}
```

4. The **HomeContent** function should be highlighted in red because you haven't declared it yet. Now, declare a new **Composable** called **HomeContent**. Add it after the **Home Composable**.

```
//Here, we create a composable function called HomeContent
//HomeContent is used to display the content of the Home composable
@Composable
fun HomeContent(
   listData: SnapshotStateList<Student>,
   inputField: Student,
   onInputValueChange: (String) -> Unit,
   onButtonClick: () -> Unit
) {
   //Here, we use LazyColumn to display a list of items lazily
   LazyColumn {
       //Here, we use item to display an item inside the LazyColumn
       item {
           Column(
               //Modifier.padding(16.dp) is used to add padding to the
Column
               //You can also use Modifier.padding(horizontal = 16.dp,
vertical = 8.dp)
               //to add padding horizontally and vertically
               //or Modifier.padding(start = 16.dp, top = 8.dp, end =
16.dp, bottom = 8.dp)
               //to add padding to each side
               modifier = Modifier.padding(16.dp).fillMaxSize(),
               //Alignment.CenterHorizontally is used to align the Column
horizontally
               //You can also use verticalArrangement = Arrangement.Center
to align the Column vertically
               horizontalAlignment = Alignment.CenterHorizontally
           ) {
               Text(text = stringResource(
                   id = R.string.enter_item)
               )
               //Here, we use TextField to display a text input field
               TextField(
                   //Set the value of the input field
                   value = inputField.name,
                   //Set the keyboard type of the input field
                   keyboardOptions = KeyboardOptions(
                       keyboardType = KeyboardType.Text
```

```
                    ),
                    //Set what happens when the value of the input field
changes
                    onValueChange = {
                        //Here, we call the onInputValueChange lambda
function
                        //and pass the value of the input field as a
parameter
                        //This is so that we can update the value of the
inputField
                        onInputValueChange(it)
                    }
                )
                //Here, we use Button to display a button
                //the onClick parameter is used to set what happens when the
button is clicked
                Button(onClick = {
                    //Here, we call the onButtonClick lambda function
                    //This is so that we can add the inputField value to the
listData
                    //and reset the value of the inputField
                    onButtonClick()
                }) {
                    //Set the text of the button
                    Text(text = stringResource(
                        id = R.string.button_click)
                    )
                }
            }
        }
        //Here, we use items to display a list of items inside the
LazyColumn
        //This is the RecyclerView replacement
        //We pass the listData as a parameter
        items(listData) { item ->
            Column(
                modifier = Modifier.padding(vertical = 4.dp).fillMaxSize(),
                horizontalAlignment = Alignment.CenterHorizontally
                ) {
```

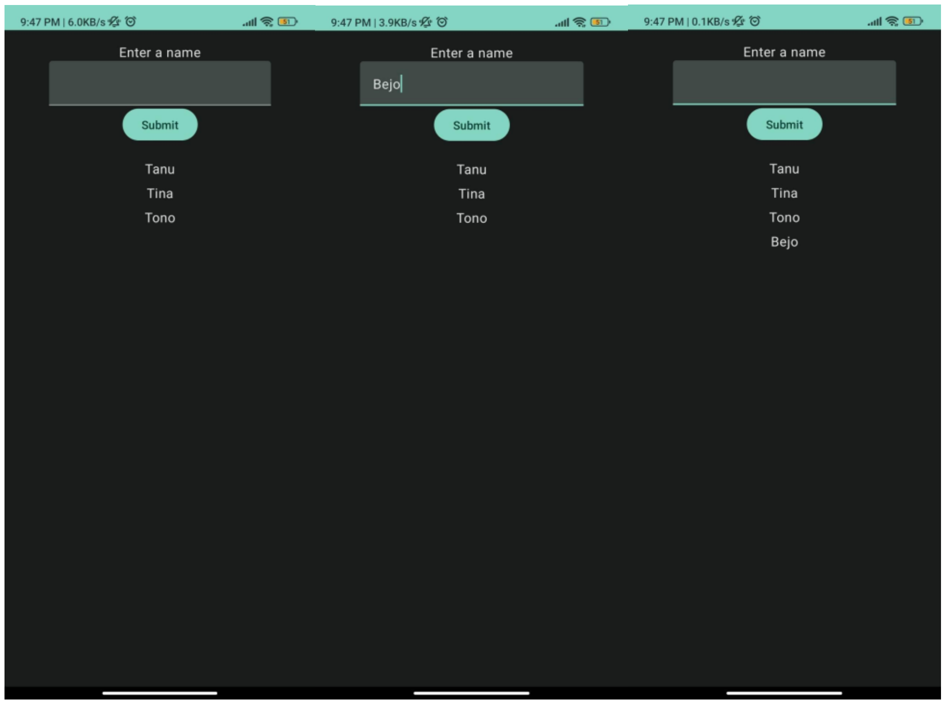
```
                Text(text = item.name)
            }
        }
    }
}
```

5. Here are few tips summarized from the code above:
    - You can add **States** into your application, just like in **React**. It'll automatically update the UI based on the data changed. Data/UI that doesn't change remains the same.
    - You declare **States** with the **mutableStateOf** or **mutableStateListOf** keyword. A **State** can store variables according to the data type.
    - Don't forget to always use the **remember** keyword to save **State Values** between **Recomposition**. **Recomposition** recomposes your **Composables** based only on the state changes. In react, this is called DOM Rerender.

6. Lastly, because you're now using **State** to control your **Data**, we don't need to pass anything into the **Home Composable**. Update your **Surface** inside the **setContent** to the code below.

```
Surface(
   //We use Modifier.fillMaxSize() to make the surface fill the whole
screen
   modifier = Modifier.fillMaxSize(),
   //We use MaterialTheme.colorScheme.background to get the background
color
   //and set it as the color of the surface
   color = MaterialTheme.colorScheme.background
) {
   Home()
}
```

7. **Run** your application, and it should be all functional. When you **Enter** a new name and press **Submit**, it should be added to the list.



COMMIT to **GITHUB** at this point. **Commit Message: "Commit No. 2 – State and Event Handler"**.

Part 3 - UI Elements and Theming

1. The application **UI** and **States** are done. Now, let's crank it up a notch by defining individual **UI Elements** with each of their own **Themes**.
2. First, let's create the **UI Elements** in a separate folder. Inside your **ui.theme Package**, create a new **Kotlin** file called **Elements.kt**.

```
//UI Element for displaying a title
@Composable
fun OnBackgroundTitleText(text: String) {
    TitleText(text = text, color =
    MaterialTheme.colorScheme.onBackground)
}
//Here, we use the titleLarge style from the typography
@Composable
fun TitleText(text: String, color: Color) {
    Text(text = text, style =
    MaterialTheme.typography.titleLarge, color = color)
}

//UI Element for displaying an item list
```

```
@Composable
fun OnBackgroundItemText(text: String) {
   ItemText(text = text, color =
   MaterialTheme.colorScheme.onBackground)
}
//Here, we use the bodySmall style from the typography
@Composable
fun ItemText(text: String, color: Color) {
   Text(text = text, style =
   MaterialTheme.typography.bodySmall, color = color)
}

//UI Element for displaying a button
@Composable
fun PrimaryTextButton(text: String, onClick: () -> Unit) {
   TextButton(text = text,
       textColor = Color.White,
       onClick = onClick
   )
}
//Here, we use the labelMedium style from the typography
@Composable
fun TextButton(text: String, textColor: Color, onClick: () -> Unit) {
   Button(
       onClick = onClick,
       modifier = Modifier.padding(8.dp),
       colors = ButtonDefaults
           .buttonColors(
               containerColor = Color.DarkGray,
               contentColor = textColor
           )
   ) {
       Text(text = text, style =
               MaterialTheme.typography.labelMedium)
   }
}
```

3. We've defined the **UI Elements**, now let's use it in **MainActivity.kt**. Update your **LazyColumn** to the code below.

```
LazyColumn {
   //Here, we use item to display an item inside the LazyColumn
   item {
       Column(
           //Modifier.padding(16.dp) is used to add padding to the Column
           //You can also use Modifier.padding(horizontal = 16.dp, vertical
= 8.dp)
           //to add padding horizontally and vertically
           //or Modifier.padding(start = 16.dp, top = 8.dp, end = 16.dp,
bottom = 8.dp)
           //to add padding to each side
           modifier = Modifier
               .padding(16.dp)
               .fillMaxSize(),
           //Alignment.CenterHorizontally is used to align the Column
horizontally
           //You can also use verticalArrangement = Arrangement.Center to
align the Column vertically
           horizontalAlignment = Alignment.CenterHorizontally
       ) {
           //Here, we call the OnBackgroundTitleText UI Element
           OnBackgroundTitleText(text = stringResource(
               id = R.string.enter_item)
           )

           //Here, we use TextField to display a text input field
           TextField(
               //Set the value of the input field
               value = inputField.name,
               //Set the keyboard type of the input field
               keyboardOptions = KeyboardOptions(
                   keyboardType = KeyboardType.Text
               ),
               //Set what happens when the value of the input field changes
               onValueChange = {
                   //Here, we call the onInputValueChange lambda function
                   //and pass the value of the input field as a parameter
                   //This is so that we can update the value of the
inputField
```

```
                    onInputValueChange(it)
                }
            )

            //Here, we call the PrimaryTextButton UI Element
            PrimaryTextButton(text = stringResource(
                id = R.string.button_click)
            ) {
                onButtonClick()
            }
        }
    }
    //Here, we use items to display a list of items inside the LazyColumn
    //This is the RecyclerView replacement
    //We pass the listData as a parameter
    items(listData) { item ->
        Column(
            modifier = Modifier
                .padding(vertical = 4.dp)
                .fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally
            ) {
            //Here, we call the OnBackgroundItemText UI Element
            OnBackgroundItemText(text = item.name)
        }
    }
}
```

4. Inside the preview, you may notice slight changes to button colors and others, but the overall layout should remain the same. You can do this for any other elements and apply the corresponding theme for that element.
5. To apply a **Custom Theme**, you may play around with **Color.kt** to make custom **Color** and **Type.kt** to make custom **Typography**.

COMMIT to **GITHUB** at this point. **Commit Message: "Commit No. 3 – UI Element and Theme"**.

Part 4 - Navigation

1. You’ve successfully made a working **Jetpack Compose Application.** Now let’s try integrating a **Navigation** into it.
2. Before you begin, add the **Navigation Library Dependency** into your **build.gradle.kts (Module :app)** and don’t forget to **Gradle Sync**.

```
implementation("androidx.navigation:navigation-compose:2.7.4")
```

3. You’ve successfully made a working **Jetpack Compose Application.** Now let’s try integrating a **Navigation** into it.
4. In order to navigate, you first need to define the routes for all the destination pages. Create a new **Composable** called **App**. This **Composable** will replace your root **Composable** (previously **Home**).

```
//Here, we create a composable function called App
//This will be the root composable of the app
@Composable
fun App(navController: NavHostController) {
   //Here, we use NavHost to create a navigation graph
   //We pass the navController as a parameter
   //We also set the startDestination to "home"
   //This means that the app will start with the Home composable
   NavHost(
       navController = navController,
       startDestination = "home"
   ) {
       //Here, we create a route called "home"
       //We pass the Home composable as a parameter
       //This means that when the app navigates to "home",
       //the Home composable will be displayed
       composable("home") {
           //Here, we pass a lambda function that navigates to
"resultContent"
           //and pass the listData as a parameter
           Home { navController.navigate(
               "resultContent/?listData=$it")
           }
       }
       //Here, we create a route called "resultContent"
       //We pass the ResultContent composable as a parameter
       //This means that when the app navigates to "resultContent",
       //the ResultContent composable will be displayed
       //You can also define arguments for the route
       //Here, we define a String argument called "listData"
       //We use navArgument to define the argument
       //We use NavType.StringType to define the type of the argument
       composable(
           "resultContent/?listData={listData}",
           arguments = listOf(navArgument("listData") {
               type = NavType.StringType }
           )
       ) {
           //Here, we pass the value of the argument to the ResultContent
composable
           ResultContent(
               it.arguments?.getString("listData").orEmpty()
           )
       }
   }
}
```

5. Don’t forget to also change your root from **Home()** to **App()** inside your **Surface** function.

```
Surface(
   //We use Modifier.fillMaxSize() to make the surface fill the whole
screen
   modifier = Modifier.fillMaxSize(),
   //We use MaterialTheme.colorScheme.background to get the background
color
   //and set it as the color of the surface
   color = MaterialTheme.colorScheme.background
) {
   val navController = rememberNavController()
   App(
       navController = navController
   )
}
```

6. Now let's update the **Home Composable** and add 1 more parameter into it.

```
@Composable
fun Home(
   navigateFromHomeToResult: (String) -> Unit
) {
```

7. Also update the **Home Content Composable** and add 1 more parameter.

```
@Composable
fun HomeContent(
   listData: SnapshotStateList<Student>,
   inputField: Student,
   onInputValueChange: (String) -> Unit,
   onButtonClick: () -> Unit,
   navigateFromHomeToResult: () -> Unit
) {
```

8. Now, update your calling function for **Home Content Composable** inside your **Home Composable** and add the extra parameter for navigating.

```
HomeContent(
   listData,
   inputField,
   { input -> inputField = inputField.copy(input) },
   {
       listData.add(inputField)
       inputField = inputField.copy("")
   },
   { navigateFromHomeToResult(listData.toList().toString()) }
)
```

9. Next still in your **HomeContent Composable**, add a **Finish Button** to navigate to the other page. Update your **LazyColumn** and add another button.

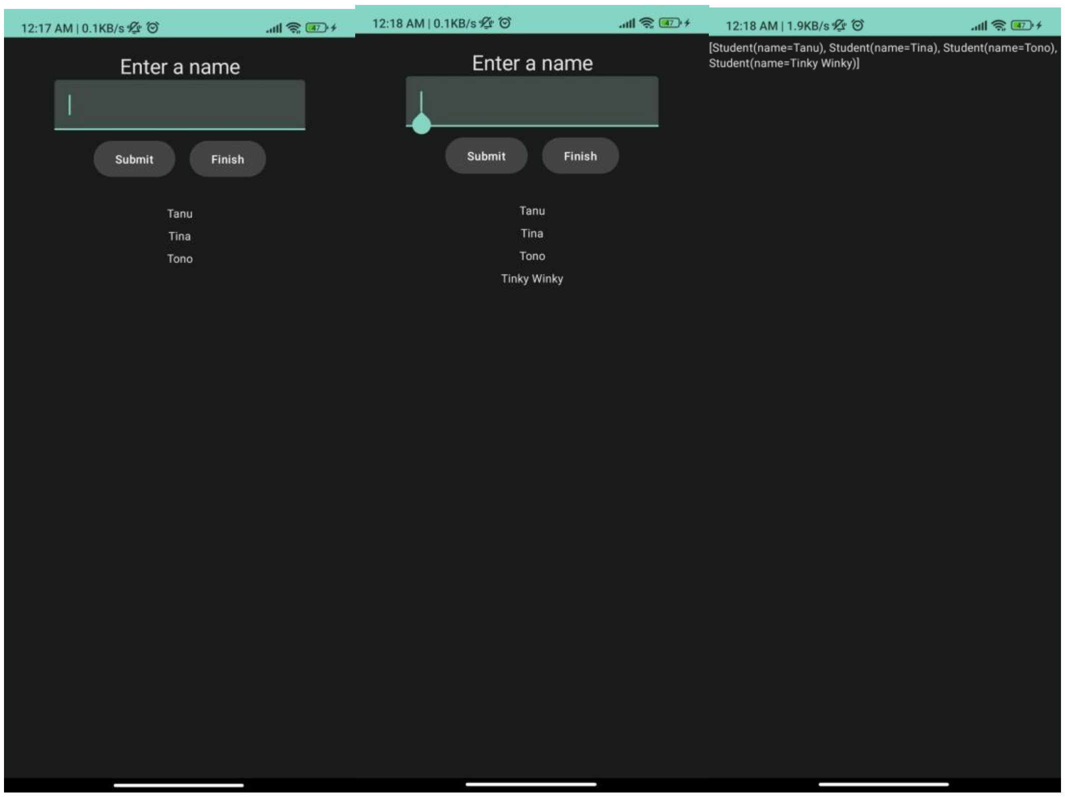
```
LazyColumn {
    item {
        Column(
            modifier = Modifier
```

```
                .padding(16.dp)
                .fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            OnBackgroundTitleText(text = stringResource(
                id = R.string.enter_item)
            )

            TextField(
                value = inputField.name,
                keyboardOptions = KeyboardOptions(
                    keyboardType = KeyboardType.Text
                ),
                onValueChange = {
                    onInputValueChange(it)
                }
            )

            Row {
                PrimaryTextButton(text = stringResource(id =
R.string.button_click)) {
                    onButtonClick()
                }
                PrimaryTextButton(text = stringResource(id =
R.string.button_navigate)) {
                    navigateFromHomeToResult()
                }
            }
        }
    }
    items(listData) { item ->
        Column(
            modifier = Modifier
                .padding(vertical = 4.dp)
                .fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            OnBackgroundItemText(text = item.name)
        }
    }
}
```

10. Lastly, add the **Result Content Composable** at the very end of your **MainActivity.kt**.

```
//Here, we create a composable function called ResultContent
//ResultContent accepts a String parameter called listData from the Home
composable
//then displays the value of listData to the screen
@Composable
fun ResultContent(listData: String) {
   Column(
       modifier = Modifier
           .padding(vertical = 4.dp)
           .fillMaxSize(),
       horizontalAlignment = Alignment.CenterHorizontally
   ) {
       //Here, we call the OnBackgroundItemText UI Element
       OnBackgroundItemText(text = listData)
   }
}
```

11. **Run** your application, **Submit** a new name, press the **Finish** button and it should transition from the **Home** page to the **Result Content** page carrying the previous list in String format.



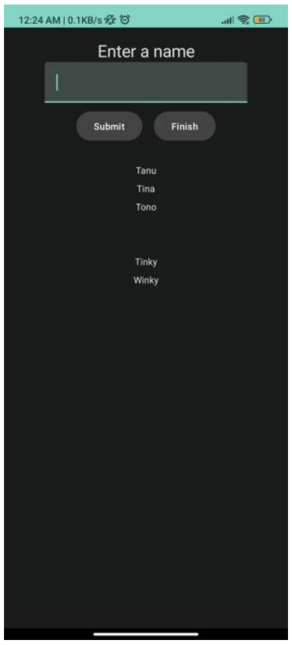
```
[Student(name=Tanu), Student(name=Tina), Student(name=Tono),
Student(name=Tinky Winky)]
```

COMMIT to **GITHUB** at this point. **Commit Message: "Commit No. 4 – Navigation"**.

ASSIGNMENT

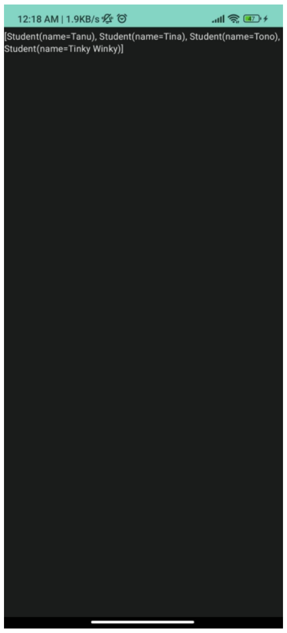
Continue your **LAB_WEEK_09** project. Your application has 2 problems:

1. If you **Submit** an **Empty** string, then the list will have a big gap in between. Fix this so that the user cannot submit if they haven't entered anything.



Tanu
Tina
Tono
Tinky
Winky

2. **[BONUS]** In the **Result Content** page, the list is shown only as a simple **String**. Use **Moshi** to convert the list to **JSON**, pass the **JSON** to **Result Content**, then display it using **LazyColumn**.



```
[Student(name=Tanu), Student(name=Tina), Student(name=Tono),
Student(name=Tinky Winky)]
```