Lab 3 Instructions

Preparation

1. Download the starter code from Canvas or GitHub
2. Extract the folder
3. Open Android Studio. Go to File, choose Open, and select the extracted folder

Note: - You may need to select the Code folder when opening with Android Studio to ensure the Project view tab appears.

Part 1 - Make CityList Dynamic

Goal

Make the city list changeable so it can support adding new cities later.

The starter code has four pieces: 1. City 2. CityListScreen 3. CityRepository 4. MainActivity

Steps

1. At the top of `CityRepository.kt`, add the following import:

```
import androidx.compose.runtime.mutableStateListOf
```

2. Find the _cities list inside `CityRepository` and replace with:

```
private val _cities = mutableStateListOf(
    City("Edmonton", "AB"),
    City("Vancouver", "BC"),
    City("Toronto", "ON")
)
```

3. Add an addCity function inside `CityRepository`

```
fun addCity(city: City) {
    _cities.add(city)
}
```

Note:

- You may now run the app to make sure that all steps have been completed successfully. The app should still display the same starter cities

Part 2 - addCity Implementation

Goal

In this part, we will add:

1. An `onAddCity` callback in `CityListScreen`

2. Text fields for city name and province
3. An **Add City** button
4. Code that creates a new `City` object from user input

Steps

> **Note**: - The Preview function will not work when you run the program at the end of Step 2. This is intentional; there is code at the end of Step 2 that will fix this when typed into Android Studio.

1. At the top of `CityListScreen.kt`, add the following imports:

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.width
import androidx.compose.material3.Button
import androidx.compose.material3.OutlinedTextField
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.setValue
```

2. Inside MainActivity, find the call to CityListScreen and update it to pass the onAddCity callback:

```
CityListScreen(
    cities = cityRepository.cities,
    onAddCity = { cityRepository.addCity(it) },
    modifier = Modifier.padding(innerPadding)
)
```

3. Update the `CityListScreen` function header to:

```
@Composable
fun CityListScreen(
    cities: List<City>,
    onAddCity: (City) -> Unit,
    modifier: Modifier = Modifier
) {
```

onAddCity: (City) -> Unit means CityListScreen receives a function, then that function takes a City object and performs an action.

4. Add input state inside `CityListScreen`, immediately after the opening brace `{`:

```
var newCityName by remember { mutableStateOf("") }
var newProvinceName by remember { mutableStateOf("") }
```

- *newCityName* stores the city name typed by the user.
- *newProvinceName* stores the province typed by the user.

- *remember* keeps these values available while CityListScreen is on screen.
- *mutableStateOf(“”)* tells Compose to update the UI when the value changes.

5. We need to add input fields above the list, so the screen needs a vertical layout. To do this, we need to wrap the existing LazyColumn inside a Column:

```
Column(modifier = modifier) {
    LazyColumn(modifier = Modifier) {
        itemsIndexed(cities) { index, city ->
            CityRow(city = city)

            if (index < cities.lastIndex) {
                HorizontalDivider()
            }
        }
    }
}
```

6. Inside the Column, add a Row above the LazyColumn:

```
Column(modifier = modifier) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(16.dp)
    ) {
    }

    LazyColumn {
        itemsIndexed(cities) { index, city ->
            CityRow(city = city)

            if (index < cities.lastIndex) {
                HorizontalDivider()
            }
        }
    }
}
```

7. Inside the empty Row braces, add the first text field for City:

```
OutlinedTextField(
    value = newCityName,
    onValueChange = { newCityName = it },
    label = { Text("City") },
    modifier = Modifier.weight(1f)
)
```

The label *City* tells the user this field is for the city name.

8. After the City text field, add the province text field:

```
Spacer(modifier = Modifier.width(8.dp))

OutlinedTextField(
    value = newProvinceName,
    onValueChange = { newProvinceName = it },
    label = { Text("Province") },
    modifier = Modifier.weight(1f)
)
```

9. Add the *Add City* button after the Province field:

```
Spacer(modifier = Modifier.width(8.dp))

Button(
    modifier = Modifier.padding(vertical = 12.dp),
    onClick = {
        if (newCityName.isNotBlank() && newProvinceName.isNotBlank()) {
            onAddCity(
                City(
                    name = newCityName,
                    province = newProvinceName
                )
            )
            newCityName = ""
            newProvinceName = ""
        }
    }
) {
    Text("Add City")
}
```

Now, run the app. You should see the top change to: `plain text City text field | Province text field | Add City button City list below`

Note: - To make the previewable work again here, modify the `CityListScreenPreview()` in `CityListScreen.kt` to this code:

```
fun CityListScreenPreview() {
    ListyCity3Theme {
        CityListScreen(
            cities = listOf(
                City("Edmonton", "AB"),
                City("Vancouver", "BC"),
                City("Calgary", "AB")
            ),
```

```
            onAddCity = {}
        )
    }
}
```

Part 3 - Update the View Design with a Floating Action Button

Goal

In this part, we will: 1. Update the *Add City* UI so the input fields are hidden at first 2. The user will press a + button to show the city and province input fields

Steps

1. At the top of `CityListScreen.kt`, add:

```
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.material3.FloatingActionButton
import androidx.compose.foundation.layout.fillMaxSize
```

2. Inside `CityListScreen`, add a new state variable below the existing `var newProvinceName`:

```
var showAddCityFields by remember { mutableStateOf(false) }
```

showAddCityFields controls whether the city and province input fields are visible.

3. Find the `Column` at the start of `CityListScreen` that wraps the input row and the city list and replace it with:

```
Column(modifier = modifier.fillMaxSize()) {
```

4. Inside the Column, before the first `Row` we have, add this new row:

```
Row(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.End
) {
    FloatingActionButton(
        modifier = Modifier.padding(16.dp),
        onClick = {
            showAddCityFields = !showAddCityFields
        }
    ) {
        Text("+")
    }
}
```

`Modifier.fillMaxWidth()` makes the row take up the full width of the screen. `horizontalArrangement = Arrangement.End` moves the button to the far right.

When the user presses the + button, showAddCityFields becomes true.

5. Find the `Row` that contains:

- the city text field
- the province text field
- the Add City button

Wrap that entire `Row` with this `if (showAddCityFields)` statement (do not change the contents of the `Row`):

```
if (showAddCityFields) {
        Row( ...
    }
```

6. Inside the Add City button's `onClick`, find where the text fields are cleared:

```
newCityName = ""
newProvinceName = ""
```

and below, add this line:

```
showAddCityFields = false
```

7. Update LazyColumn modifier from:

```
LazyColumn(modifier = Modifier) {
```

to:

```
LazyColumn(modifier = Modifier.fillMaxSize()) {
```

8. Run the app.

Lab Participation Exercise: Hints

1. You may need to keep track of which city the user selected

```
var selectedCity by remember { mutableStateOf<City?>(null) }
```

2. A city row can be made selectable by adding a click action to CityRow. You will need to import clickable:

```
import androidx.compose.foundation.clickable
```

3. To edit a city, you may need text fields for the updated city name and province

4. You may need to add an `updateCity` function to CityRepository

5. You do not need to use permanent storage. The edited city only needs to update while the app is running.

6. Because `City` uses `val` properties, you cannot directly change `city.name` or `city.province`. One approach is to replace the selected city with a new `City` object:

```
fun updateCity(oldCity: City, updatedCity: City) {
    val index = _cities.indexOf(oldCity)
    if (index != -1) {
        _cities[index] = updatedCity
    }
}
```