2

Creating Screens Using a Declarative UI and Exploring Compose Principles

Mobile applications require a **User Interface (UI)** for user interactions. For instance, the old way of creating the UI was imperative in Android. This meant having a separate prototype of the application's UI using unique **Extensible Markup Language (XML)** layouts and not the same language used to build your logic.

However, with Modern Android Development, there is a push to stop using imperative programming and start using a declarative way of making the UI, which means developers design the UI based on the data received. This design paradigm uses one programming

language to create an entire application.

It is fair to acknowledge it may seem difficult for new developers to decide what to learn when building a UI: the old way of creating views or opting for the new Jetpack Compose. However, suppose you've built an Android application before the Jetpack Compose era.

In such a case, you may already know using XML is a bit tedious, especially if your code base is complex. However, utilizing Jetpack Compose as your first choice makes work easier. In addition, it simplifies UI development by ensuring developers use less code, as they take advantage of the intuitive Kotlin APIs. Hence, there is a logical push by new developers when creating views to use Jetpack Compose instead of XML.

However, knowing both can be beneficial since many applications still use XML layouts, and you might have to maintain the view but build new ones using Jetpack Compose. In this chapter, we will look at Jetpack Compose basics by trying to implement small examples using columns, rows, boxes, lazy columns, and more.

In this chapter, we'll be covering the following recipes:

- Implementing Android views in Jetpack Compose
- Implementing a scrollable list in Jetpack Compose
- Implementing your first tab layout with a view pager using Jetpack Compose
- Implementing animations in Compose
- Implementing accessibility in Jetpack Compose
- Implementing declarative graphics using Jetpack Compose

Technical requirements

The complete source code for this chapter can be found at **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter-two**. To be able to view all the recipes, you will need to run all the preview functions separately. Hence, look for the **`@Preview`** composable function to view the UI created.

Implementing Android views in Jetpack Compose

In every Android application, having a UI element is very crucial. A view in Android is a simple building block for a UI. A view ensures users can interact with your application through a tap or other motion. This recipe will look at different Compose UI elements and see how we can build them.

Getting ready

In this recipe, we will create one project that we will re-use for the entire chapter, so let's go ahead and follow the steps in *Chapter 1*, *Getting Started with Modern Android Development Skills*, on how to create your first Android project.

Create a project and call it **`Compose Basics`**. In addition, we will mostly use the **Preview** section to view the UI element we create.

How to do it...

Once you have created the project, follow these steps to build several Compose UI elements:

1. Inside our project, let us go ahead and create a new package and call it components. This is where we will add all the components we create.

2. Create a Kotlin file and call it **`UIComponents.kt`**; inside **`UIComponent`**, go ahead and create a composable function, call it **`EditTextExample()`**, and call the **`OutlinedTextField()`** function; this will prompt you to import the required import, which is **`androidx.Compose.material.OutlinedTextField`**:

```
@Composable
fun EditTextExample() {
    OutlinedTextField()
}
```

3. When you look deep into **`OutlineTextField`** (see *Figure 2.1*), you will notice the function accepts several inputs, and this is very useful when you need to customize your own composable functions.

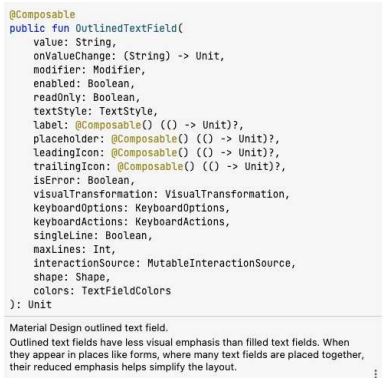

Figure 2.1 – The OutlinedTextField input

4. For our example, we will not do much with the UI we create and will rather just look at how we create them.

5. Now, to fully create our **`OutlinedTextField()`** based on the types of input we see it accepts, we can give it a text and color and we can decorate it using a **`Modifier()`**; that is, by giving it specific instructions such as **`fillMaxWidth()`**, which sets the max width. When we say fill, we are simply specifying it should be fully filled. We set **`.padding(top)`** to **`16.dp`**, which applies additional space along each edge of the content in **`dp`**. It also has a value, which is the value to be entered in the **`OutlinedTextField`**, and an **`onValueChange`** lambda that listens to the input change.
6. We also give our **`OutlinedText`** some border colors when focused and when not focused to reflect the different states. Hence, if you start entering input, the box will change color to blue, as specified in the code:

```
@Composable
fun EditTextExample() {
    OutlinedTextField(
        value = "",
        onValueChange = {},
        label = { Text(stringResource(id =
        R.string.sample)) },
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 16.dp),
        colors =
            TextFieldDefaults.outlinedTextFieldColors(
```

```
                    focusedBorderColor = Color.Blue,
                    unfocusedBorderColor = Color.Black
                )
        )
    }
```

7. We also have another type of **`TextField`**, which is not outlined, and if you compare what **`OutlinedTextField`** takes in as input, you will notice they are fairly similar:

```
@Composable
fun NotOutlinedEditTextExample() {
    TextField(
        value = "",
        onValueChange = {},
        label = { Text(stringResource(id =
        R.string.sample)) },
        modifier = Modifier
            .fillMaxWidth()
            .padding(top = 8.dp, bottom = 16.dp),
        colors =
            TextFieldDefaults.outlinedTextFieldColors(
                focusedBorderColor = Color.Blue,
                unfocusedBorderColor = Color.Black
            )
    )
}
```

8. You can run the application by adding the Compose functions inside the **`@Preview`** composable function. In our example, we can create **`UIElementPreview()`**, which is a preview function for displaying our UI. In *Figure 2.2*, the top view is

`OutlinedTextField`, whereas the second one is a normal **`TextField`**.

Figure 2.2 – OutlinedTextField and TextField

9. Now, let's go ahead and look at button examples. We will look at different ways to create buttons with different shapes. If you hover over the **`Button()`** composable function, you will see what it accepts as input, as shown in *Figure 2.3*.

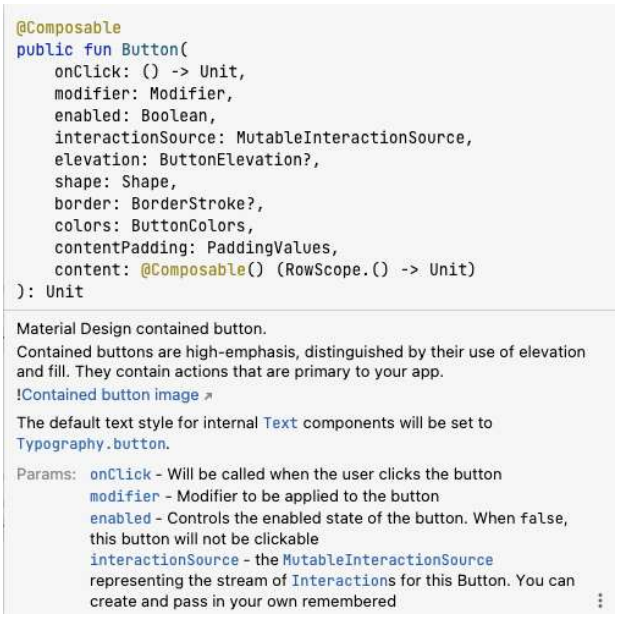

Figure 2.3 – Button input

In our second example, we will try to create a button with an icon on it. In addition, we will add text, which is crucial when creating buttons since we need to specify to users what action or what the button will be doing once it is clicked on.

10. So, go ahead and create a Compose function in the same Kotlin file and call it **`ButtonWithIcon()`**, and then import the **`Button()`** composable function.
11. Inside it, you will need to import an **`Icon()`** with **`painterResource`** input, a content description, **`Modifier`**, and **`tint`**. We will also need **`Text()`**, which will give our button a name. For our example, we will not use **`tint`**:

```
@Composable
fun ButtonWithIcon() {
    Button(onClick = {}) {
        Icon(
            painterResource(id =
                R.drawable.ic_baseline_shopping_bag_24                ),
            contentDescription = stringResource(
                id = R.string.shop),
            modifier = Modifier.size(20.dp)
        )
        Text(text = stringResource(id = R.string.buy),
            Modifier.padding(start = 10.dp))
    }
}
```

12. Let us also go ahead and create a new composable function and call it **`CornerCutShapeButton()`**; in this example, we will try to create a button with cut corners:

```
@Composable
fun CornerCutShapeButton() {
    Button(onClick = {}, shape = CutCornerShape(10)) {
        Text(text = stringResource(
            id = R.string.cornerButton))  }}}}
```

13. Let us also go ahead and create a new composable function and call it **`RoundCornerShapeButton()`**; in this example, we will try to create a button with round corners:

```
@Composable
fun RoundCornerShapeButton() {
    Button(onClick = {}, shape =
    RoundedCornerShape(10.dp)) {
        Text(text = stringResource(
            id = R.string.rounded))
```

```
        }
    }
```

14. Let us also go ahead and create a new composable function and call it **`ElevatedButtonExample()`**; in this example, we will try to create a button with elevation:

```
@Composable
fun ElevatedButtonExample() {
    Button(
        onClick = {},
        elevation = ButtonDefaults.elevation(
            defaultElevation = 8.dp,
            pressedElevation = 10.dp,
            disabledElevation = 0.dp
        )
    ) {
        Text(text = stringResource(
            id = R.string.elevated))
    }
}
```

15. When you run the application, you should have an image similar to *Figure 2.4*; the first button after **`TextField`** is **`ButtonWithIcon()`**, the second one is **`CornerCutShapeButton()`**, the third is **`RoundCornerShapeButton()`**, and, lastly, we have **`ElevatedButtonExample()`**.

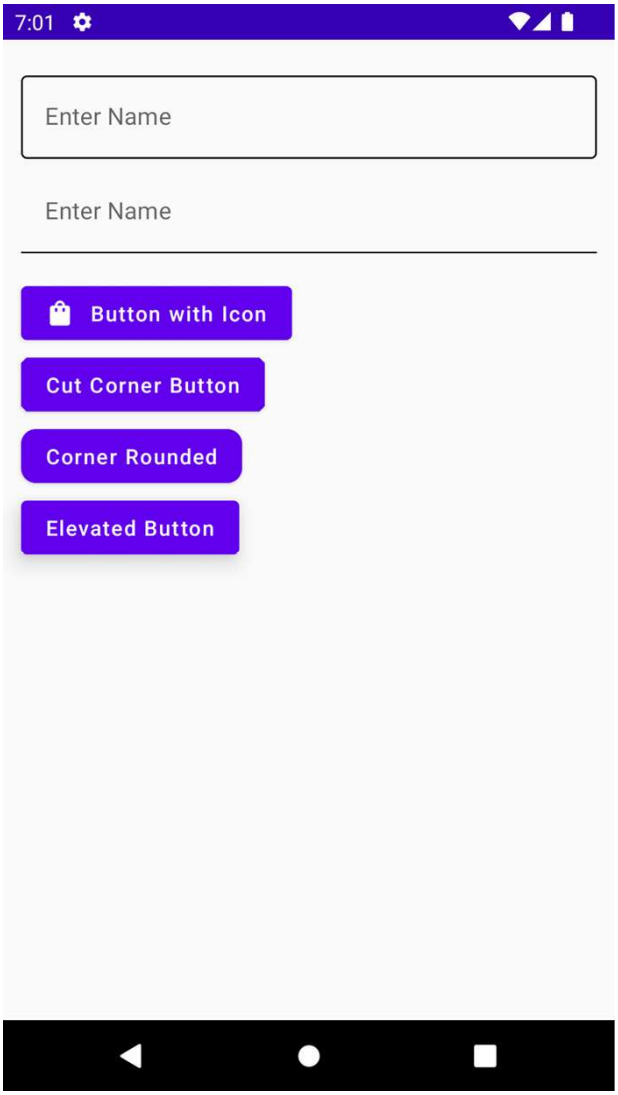

Figure 2.4 – The different button types and other UI elements

16. Now, let us look at one last example since we will be using different views and styles throughout the book and will learn more in the process. Now, let us look at an image view; the `Image()` composable function takes in several inputs, as shown in *Figure 2.5*.

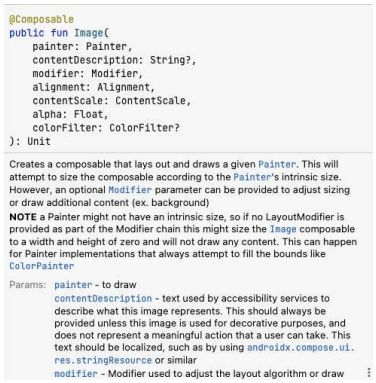

Figure 2.5 – Different ImageView input types

17. In our example, **`Image()`** will only have a painter, which is not nullable, meaning you need to provide an image for this composable function, a content description for accessibility, and a modifier:

```
@Composable
fun ImageViewExample() {
    Image(
        painterResource(id = R.drawable.android),
        contentDescription = stringResource(
            id = R.string.image),
        modifier = Modifier.size(200.dp)
    )
}
```

18. You can also try to play around with others things, such as adding **`RadioButton()`** and **`CheckBox()`** elements and customizing them. When you run your application, you should have something similar to *Figure 2.6*.

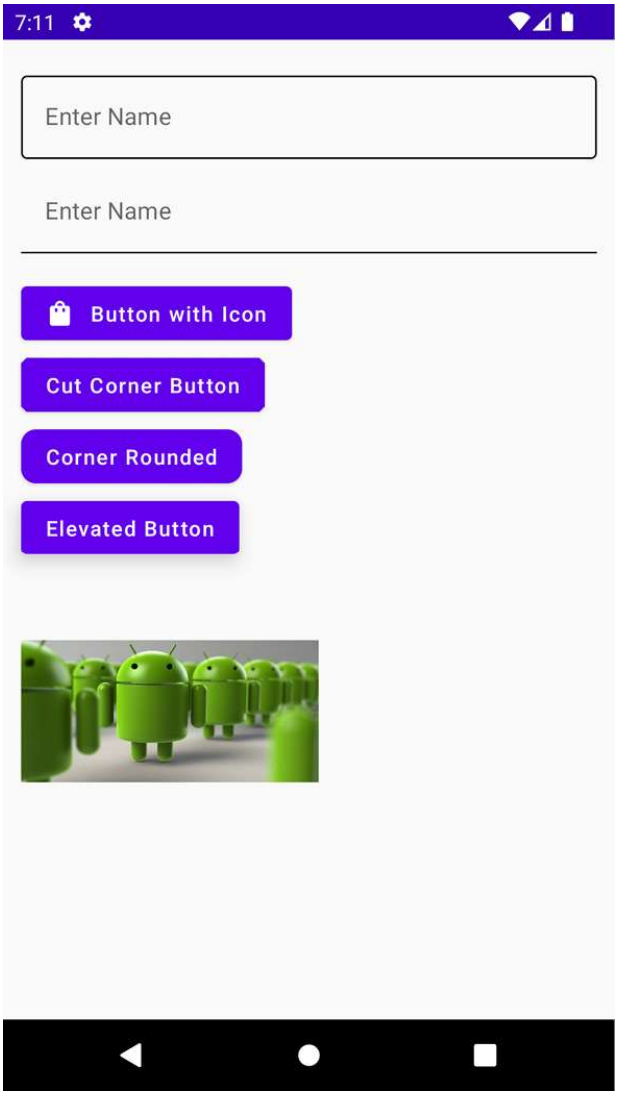

Figure 2.6 – Several UI components

How it works…

Every composable function is annotated with the **`@Composable`** annotation. This annotation tells the Compose compiler that the provided compiler is intended to convert the provided data into a UI. It is also important to note each composable function name needs to be a noun and not a verb or an adjective, and Google provides these guidelines. Any composable function you create can accept parameters that enable

the app logic to describe or modify your UI.

We mention the Compose compiler, which means that a compiler is any special program that takes the code we wrote, examines it, and translates it into something the computer can understand – or machine language.

In **`Icon()`**, **`painterResouce`** specifies the icon we will be adding to the button, the content description helps with accessibility, and the modifier is used to decorate our icon.

We can preview the UI elements we build by adding the **`@Preview`** annotation and adding **`showBackground = true`**:

```
@Preview(showBackground = true)
```

`@Preview` is powerful, and we will look at how you can utilize it better in future chapters.

Implementing a scrollable list in Jetpack Compose

When building Android applications, one thing that we can all agree on is

you must know how to build a **`RecyclerView`** to display your data. With our new, modern way of building Android applications, if we need to use **`RecyclerView`**, we can use **`LazyColumn`**, which is similar. In this recipe, we will look at rows, columns, and **`LazyColumn`**, and build a scrollable list using our dummy data.

In addition, we will be learning some Kotlin in the process.

Getting ready

We will continue using the **`Compose Basics`** project to build a scrollable list; hence, to get started, you need to have done the previous recipe.

How to do it...

Follow these steps to build your first scrollable list:

1. Let us go ahead and build our first scrollable list, but first, we need to create our dummy data, and this is the item we want to be displayed on our list. Hence, create a package called **`favoritecity`** where our scrollable example will live.
2. Inside the **`favoritecity`** package, create a new data class and call it

City; this will be our dummy data source – **data class City ()**.

3. Let us model our **City** data class. Make sure you add the necessary imports once you have added the annotated values:

```
data class City(
    val id: Int,
    @StringRes val nameResourceId: Int,
    @DrawableRes val imageResourceId: Int
)
```

4. Now, in our dummy data, we need to create a Kotlin class and call this class **CityDataSource**. In this class, we will create a function called **loadCities()**, which will return our list of **List<City>**, which we will display in our scrollable list. Check the *Technical requirements* section for all the required imports to get all the code and images:

```
class CityDataSource {
    fun loadCities(): List<City> {
        return listOf<City>(
            City(1, R.string.spain, R.drawable.spain),
            City(2, R.string.new_york,
                R.drawable.newyork),
            City(3, R.string.tokyo, R.drawable.tokyo),
            City(4, R.string.switzerland,
                R.drawable.switzerland),
            City(5, R.string.singapore,
                R.drawable.singapore),
            City(6, R.string.paris, R.drawable.paris),
        )
```

```
        }
    }
```

5. Now, we have our dummy data, and it is time to display this on our scrollable list. Let's create a new Kotlin file in our **`components`** package and call it **`CityComponents`**. In **`CityComponents`**, we will create our **`@Preview`** function:

```
@Preview(showBackground = true)
@Composable
private fun CityCardPreview() {
    CityApp()
}
```

6. Inside our **`@Preview`** function, we have another composable function, **`CityApp()`**; inside this function, we will call our **`CityList`** composable function, which has the list as a parameter. In addition, in this composable function, we will call **`LazyColumn`**, and **`items`** will be **`CityCard(cities)`**. See the *How it works* section for further explanation about **`LazyColumn`** and **`items`**:

```
@Composable
fun CityList(cityList: List<City>) {
    LazyColumn {
        items(cityList) { cities ->
            CityCard(cities)
        }
    }
}
```

7. Finally, let us construct our **`CityCard(city)`** composable function:

```
@Composable
fun CityCard(city: City) {
    Card(modifier = Modifier.padding(10.dp),
    elevation = 4.dp) {
        Column {
            Image(
                painter = painterResource(
                    city.imageResourceId),
                contentDescription = stringResource(
                    city.nameResourceId),
                modifier = Modifier
                    .fillMaxWidth()
                    .height(154.dp),
                contentScale = ContentScale.Crop
            )
            Text(
                text = LocalContext.current.getString(
                    city.nameResourceId),
                modifier = Modifier.padding(16.dp),
                style = MaterialTheme.typography.h5
            )
        }
    }
}
```

8. When you run the **`CityCardPreview`** composable function, you should have a scrollable list, as seen in *Figure 2.6*.

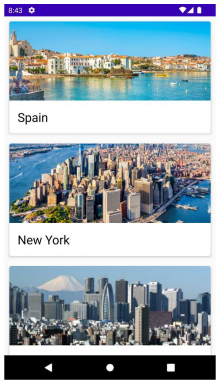

Figure 2.7 – A scrollable list of cities

How it works…

In Kotlin, a list has two types, **immutable** and **mutable**. Immutable lists are items that cannot be modified, whereas mutable lists are items in the list that can be modified. To define a list, we can say a list is a generic ordered collection of elements, and these elements can be in the form of integers, strings, images, and so on, which is mostly informed by the type of data we want our lists to contain. For instance, in our example, we have a string and image helping identify our favorite cities by name and image.

In our `City` data class, we use `@StringRes`, and `@DrawableRes` in order to just pull this directly from the `res` folders for `Drawable` and `String` easily, and they also represent the ID for the images and string.

We created `CityList` and annotated it with the composable function and de-

clared the list of city objects as our parameter in the function. A scrollable list in Jetpack Compose is made using **`LazyColumn`**. The main difference between **`LazyColumn`** and **`Column`** is that when using **`Column`**, you can only display small items, as Compose loads all items at once.

In addition, a column can only hold fixed composable functions, whereas **`LazyColumn`**, as the name suggests, loads the content as required on demand, making it good for loading more items when needed. In addition, **`LazyColumn`** comes with a scrolling ability inbuilt, which makes work easier for developers.

We also created a composable function, **`CityCard`**, where we import the **`Card()`** element from Compose. A card contains content and actions about a single object; in our example, for instance, our card has an image and the name of the city. A **`Card()`** element in Compose has the following inputs in its parameter:

```
@Composable
fun Card(
    modifier: Modifier = Modifier,
    shape: Shape = MaterialTheme.shapes.medium,
    backgroundColor: Color = MaterialTheme.colors.surface,
    contentColor: Color = contentColorFor(backgroundColor),
    border: BorderStroke? = null,
```

```
        elevation: Dp = 1.dp,
        content: @Composable () -> Unit
    ),
```

This means you can easily model your card to the best fitting; our card has padding and elevation, and the scope has a column. In this column, we have an image and text, which helps describe the image for more context.

See also

There is more to learn about lists and grids in Compose; you can use this link to learn more: **https://developer.android.com/jetpack/compose/lists**.

Implementing your first tab layout with a view pager using Jetpack Compose

In Android development, having a slide between pages is very common, with a significant use case being onboarding or even when you are trying to display specific data in a tabbed, carousel way. In this recipe, we will build a simple horizontal pager in Compose and see how we can utilize the new knowledge

to build better and more modern Android apps.

Getting ready

In this example, we will build a horizontal pager that changes colors when selected to show the state is selected. We will look into states in *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*, for better understanding. Open the **`Compose Basics`** project to get started.

How to do it...

Follow these steps to build your tab carousel:

1. Add the following pager dependencies to **`build.gradle(Module:app)`**:

```
implementation "com.google.accompanist:accompanist-pager:0.x.x"
implementation "com.google.accompanist:accompanist-pager-indicators:0.x.x"
implementation 'androidx.Compose.material:material:1.x.x'
```

Jetpack Compose offers **Accompanist**, a group of libraries that aims to support it with commonly required features by developers – for instance, in our case, the pager.

2. In the same project from previous recipes, let's create a package and call it **`pagerexample`**; inside it, create a Kotlin file and call it

CityTabExample; inside this file, create a composable function and call it **CityTabCarousel**:

```
@Composable
fun CityTabCarousel(){}
```

3. Now, let us go ahead and build our **CityTabCarousel**; for our example, we will create a dummy list of pages with our cities from the previous project:

```
@Composable
fun CityTabCarousel(
    pages: MutableList<String> = arrayListOf(
        "Spain",
        "New York",
        "Tokyo",
        "Switzerland",
        "Singapore",
        "Paris" )) {. . .}
```

4. We will need to change the color of the button based on the state, and to do this; we need to use **LocalContext**, which provides the context we can use. We will also need to create a **var pagerState = rememberPagerState()**, which will remember our pager state, and finally, when clicked, we will need to move to the next city in our pager, which will be very helpful. Hence, go ahead and add the following to the **CityTabCarousel** composable function:

```
val context = LocalContext.current
var pagerState = rememberPagerState()
val coroutineScope = rememberCoroutineScope()
```

5. Now, let's create the **`Column`** element and add our **`ScrollableTabRow()`** composable function:

```
Column {
    ScrollableTabRow(
        selectedTabIndex = pagerState.currentPage,
        indicator = { tabPositions ->
            TabRowDefaults.Indicator(...)
        },
        edgePadding = 0.dp,
        backgroundColor = Color(
            context.resources.getColor(R.color.white,
                null)),
    ) {
        pages.forEachIndexed { index, title ->
            val isSelected =
                pagerState.currentPage == index
            TabHeader(
                title,
                isSelected,
                onClick = { coroutineScope.launch {
                pagerState.animateScrollToPage(index)
                } },
            )
        }
    }
```

6. Add **`Text()`** and **`TabHeader()`** for **`HorizontalPager`**:

```
    HorizontalPager(
        count = pages.size,
        state = pagerState,
        modifier = Modifier
            .fillMaxWidth()
            .fillMaxHeight()
```

```
            .background(Color.White)
    ) { page ->
        Text(
            text = "Display City Name:
                ${pages[page]}",
            modifier = Modifier.fillMaxWidth(),
            style = TextStyle(
                textAlign = TextAlign.Center
            )
        )
    }
```

7. Please download the entire code for this recipe by following the link provided in the *Technical requirements* section to add all the required code. Finally, run the **`@Preview`** function, and your app should look like *Figure 2.8*.

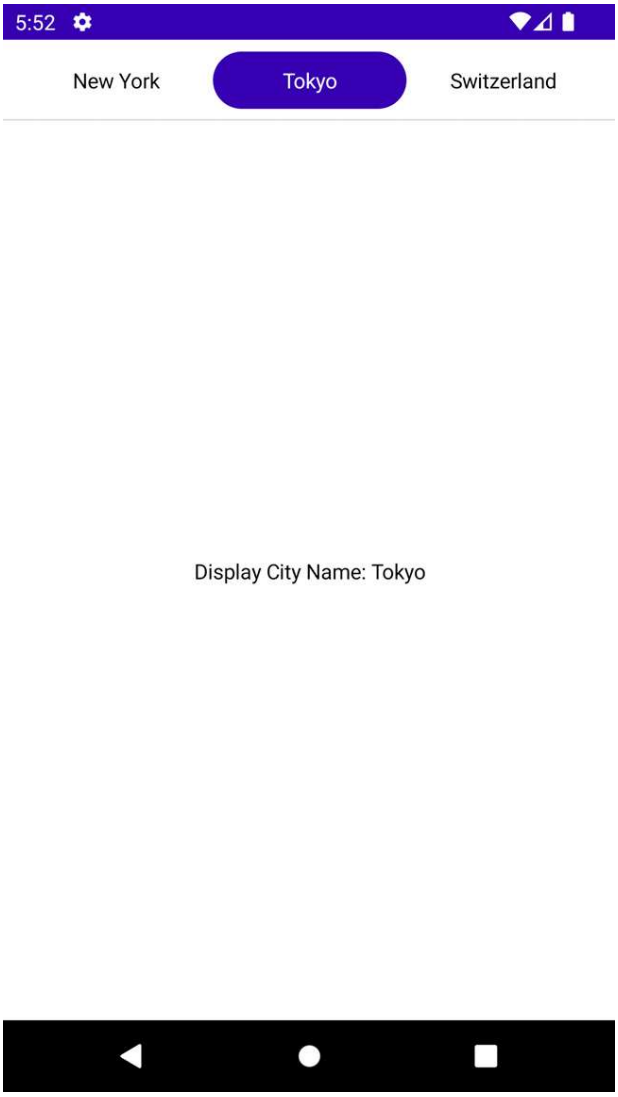

Figure 2.8 – Tabs with cities

How it works…

Accompanist comes with some significant libraries – for example, System UI Controller, AppCompact Compose Theme Adapter, Material Theme Adapter, Pager, Drawable Painter, and Flow Layouts, just to mention a few.

The **`ScrollableTabRow()`** that we use inside **`Column`** in the **`CityTabCarousel`** function contains a row of tabs and helps display an indicator underneath

the currently focused or selected tab. In addition, as the name suggests, it enables scrolling and you do not have to implement further scrolling tooling. It also places its tab offsets at the starting edge, and you can quickly scroll tabs that are off-screen, as you will see when you run the `@Preview` function and play around with it.

When we invoke `remember()`, in Compose, this means we keep any value consistent across recomposition. Compose provides this function to help us store single objects in memory. When we trigger our application to run, `remember()` stores the initial value. As the word means, it simply retains the value and returns the stored value so that the composable function can use it.

Furthermore, whenever the stored value changes, you can update it, and the `remember()` function will keep it. The next time we trigger another run in our app and recomposition occurs, the `remember()` function will provide the latest stored value.

You will also notice our `MutableList<String>` is indexed at each position, and we do this to check which is selected. It is within this Lambda that we call `TabHeader` and showcase the selected tab pages. `forE-`

achIndexed performs the given action on each element, providing a sequential index of elements. We also ensure when a user clicks on a specific tab, we are on the right page:

```
onClick = { coroutineScope.launch { pagerState.animateScrollToPage(index) } }
```

HorizontalPager is a horizontally scrolling layout that allows our users to flip between items from left to right. It takes in several inputs, but we supply it with the count, state, and modifier to decorate it in our use case. In the Lambda, we display text – in our example, showing which page we are on, which helps when navigating, as shown in *Figure 2.9*:

```
@ExperimentalPagerApi
@Composable
@ComposableInferredTarget
public fun HorizontalPager(
    count: Int,
    modifier: Modifier,
    state: PagerState,
    reverseLayout: Boolean,
    itemSpacing: Dp,
    contentPadding: PaddingValues,
    verticalAlignment: Alignment.Vertical,
    flingBehavior: FlingBehavior,
    key: ((Int) -> Any)?,
    userScrollEnabled: Boolean,
    content: @Composable() (PagerScope.(Int) -> Unit)
): Unit
```

Figure 2.9 – HorizontalPager

Our **TabHeader** composable function has a **Box()**; a box in Jetpack Compose will always size itself to fit the content, and this is subject to the specified constraints. In our example, we decorate our **Box** with the selectable modifier, which configures components to be selectable as part of a mutually exclusive

group, allowing each item to be selected only once at any given time.

IMPORTANT NOTE

Ensure your target and compile SDK targets 33. In addition, you will notice that most Accompanist's libraries are experimental, which means they can change. There is debate on whether to use this in your production, so you should always consult your team on these APIs. To see the entire list of libraries supported by Accompanist, you can follow this link: **https://github.com/google/accompanist**.

Implementing animations in Compose

Animation in Android is the process of adding motion effects to views. This can be achieved using images, text, or even starting a new screen where the transition is noticeable using motion effects. Animations are vital in Modern Android Development since modern UIs are more interactive and adaptive to smoother experiences, and users like them.

Furthermore, applications these days are rated based on how great their UI and user experiences are, hence the need to ensure your application is modern and robust. In this example, we will build a collapsing toolbar, an animation that is widely used in the Android world.

Getting ready

We will continue using the **`Compose Basics`** project.

How to do it...

We will be building a collapsing toolbar in this recipe; there are other great animations you can now build utilizing the power of Compose. The power is in your hands:

1. We will not need to add any dependency to this recipe. We already have everything in place. So, let us go ahead and create a new package and add a Kotlin file, **`collapsing-toolbar`**.
2. Inside the Kotlin file, go ahead and create a new composable function, **`CollapsingTool BarExample()`**:

```
@Composable
fun CollapsingToolbarExample() {...}
```

3. We will have all our needed composable functions in a box; you can refer to the previous recipe to refresh your memory on that. We will also need to define the height at which we will start to collapse our view, and this can be based on preference; for our example, we can set **`height`** to **`260.dp`**:

```
private val height = 260.dp
private val titleToolbar = 50.dp
```

4. Let us go ahead and add more composable functions with dummy text data to display once we scroll our content. We can assume this app is used for reading information about the cities we display:

```
@Composable
fun CollapsingToolbarExample() {
    val scrollState: ScrollState =
        rememberScrollState(0)
    val headerHeight = with(LocalDensity.current) {
        height.toPx() }
    val toolbarHeight = with(LocalDensity.current) {
        titleToolbar.toPx() }
    Box(
        modifier = Modifier.fillMaxSize()
    ) {
        CollapsingHeader(scrollState, headerHeight)
        FactsAboutNewYork(scrollState)
        OurToolBar(scrollState, headerHeight,
            toolbarHeight)
        City()
    }
}
```

5. In our **`CollapsingHeader`** function, we pass in the scroll state and the **`headerHeight`** a float. We decorate Box with a **`Modifier.graphicLayer`**, where we set a parallax effect to make it look good and presentable.
6. We also ensure we add a **`Brush()`** and set the colors we need, and specify where it should start:

```
Box(
    Modifier
        .fillMaxSize()
        .background(
            brush = Brush.verticalGradient(
                colors = listOf(Color.Transparent,
                Color(0xFF6D38CA)),
                startY = 1 * headerHeight / 5
            )
        )
)
...
```

7. **`FactsAboutNewYork`** is not a complex composable function, just dummy text; then, finally, in **`ToolBar`**, we utilize **`AnimatedVisibility`** and declare our **`enter`** and **`exit`** transition:

```
AnimatedVisibility(
    visible = showToolbar,
    enter = fadeIn(animationSpec = tween(200)),
    exit = fadeOut(animationSpec = tween(200))
) {
...
```

8. Finally, run the **`@Preview`** function, and you will have a collapsible toolbar, which brings a smooth experi-

ence to your UI. In addition, get the entire code in the *Technical requirements* section.

Figure 2.10 – A collapsible toolbar

How it works…

In Modern Android Development, the Jetpack Compose library has many animation APIs that are available as composable functions. For example, you might want your image or text to fade in and fade out.

Hence, if you are animating appearance and disappearance, which can be for an image, a text, a radio group, a button, and so on, you can use **`AnimatedVisibility`** to achieve this. Otherwise, if you are swapping content based on the state and want your content to crossfade, you can use **`CrossFade`**, or **`AnimatedContent`**.

`val headerHeight = with(LocalDensity.current) { height.toPx() }` provides density, which will be used to transform the DP and SP units, and we can use this when we provide the DP, which we will do and later convert into the body of our layout.

You can call the modifier and use **`graphicsLayer`** to update any of the content above it independently to minimize invalidated content. In addition, **`graphicsLayer`** can be used to apply effects such as scaling, rotation, opacity, shadow, or even clipping.

`translationY = -scroll.value.toFloat() / 2f` basically sets the vertical pixel offset of the layer relative to its top bound. The default value is always zero, but you can customize this to fit your needs. We also ensure the gradient is only applied to

wrapping the title in `startY = 1 * headerHeight / 5`.

`EnterTransition` defines how the target content should appear; a target here can be an image, a text, or even a radio group. On the other hand, `ExitTransition` defines how the initial target content should disappear when exiting the app or navigating away.

`AnimatedContent` offers `slideIntoContainer` and `slideOutOfContainer`, and it animates its content as it changes based on the target state, which is remarkable. In addition, you can also encapsulate a transition and make it reusable by creating a class that holds all your animation values and an `Update()`function, which returns an instance of that class.

It is also fair to mention that, as with the old ways of doing animation in Android using `MotionLayout`, there are many ways to do transitions in Jetpack Compose. For instance, in *Table 2.1*, you will see the different types of transitions:

EnterTransition	ExitTransition
`SlideIn`	`SlideOut`

`FadeIn`	`FadeOut`
`SlideInHorizontally`	`SlideOutHorizontally`
`SlideInVertically`	`SlideOutVertically`
`ScaleIn`	`SlaceOut`
`ExpandIn`	`ShrinkOut`
`ExpandHorizontally`	`ShinkHorizontally`
`ExpandVertically`	`ShrinkVertically`

Table 2.1 – A table showing different types of transitions

In addition, you can add your own custom animation effects in Jetpack Compose beyond the already built-in enter and exit animations by simply accessing the elemental transition instance via the `transition` property inside the content lambda for `AnimatedVisibility`. You will also notice any animation states that have been added.

Implementing accessibility in Jetpack Compose

As we build Android applications, we need to always have accessibility in the back of our minds because this makes technology inclusive and ensures all people with special needs are considered as we build applications.

Accessibility should be a team effort. If well handled, the advantages include having more people using your application. An accessible application is better for everyone. You also reduce the risk of being sued.

There are different types of disabilities, such as visual, aural, and motor impairments. If you open your **Accessibility** settings, you will see the different options that people with disabilities use on their devices.

Getting ready

Like previous recipes, we will continue using our sample project from previous recipes; you do not need to install anything.

How to do it...

For this recipe, we will describe the visual elements, which are very vital:

1. By default, when we add an `Image` function, you might notice that it has

two parameters, a painter for the image and a content description to visually describe the element:

```
Image(painter = , contentDescription = )
```

2. When you set the content description to **`null`**, you indicate to the Android framework that this element does not have an associated action or state. So, let's go ahead and update all our content descriptions:

```
Image(
    modifier = modifier
    painter = painterResource(city.imageResourceId),
    contentDescription =
        stringResource(R.string.city_images))
)
```

3. Make sure you add the string to the **`string res`** folder:

```
<string name="city_images">City Images</string>
```

4. So, go ahead and ensure you add a content description for every image that requires it.
5. In Compose, you can easily indicate whether a text is a heading by specifying this in the modifier and using semantics to show that that is a heading. Let's add that in our decorated text:

```
...
modifier = Modifier
    .padding(18.dp)
    .semantics { heading() }
...
```

6. Finally, we can go ahead and compile, run, and test whether our application is accessible by following this link on how to manually test using talkback or using automated testing: **https://developer.android.com/guide/topics/ui/accessibility/testing**.

How it works…

Jetpack Compose is built with accessibility in mind; that is to say, material components such as **`RadioButton`**, **`Switch`**, and so on have their size internally set, but only when these components can receive user interactions.

Furthermore, any screen element that users can click on or interact with should be large enough for reliable interaction. A standard format sets these elements to a size of at least **`48dp`** for **`width`** and **`height`**.

For example, **`Switch`** has its **`onCheckChanged`** parameter set to a non-null value, including width and height of at least **`48dp`**; we would have **`CheckableSwitch()`**, and **`NonCheckableSwitch()`**:

```
@Composable
fun CheckableSwitch(){
    var checked by remember { mutableStateOf(false) }
    Switch(checked = checked, onCheckedChange = {} )
```

```
}
@Composable
fun NonCheckableSwitch(){
    var checked by remember { mutableStateOf(false) }
    Switch(checked = checked, onCheckedChange = null )
}
```

Once you have implemented accessibility in your applications, you can easily test it by installing analysis tools from the Play Store – **`uiautomatorviewer`** and **`lint`**. You can also automate your tests using Espresso or Roboelectric to check for accessibility support.

Finally, you can manually test your application for accessibility support by going to **Settings**, then to **Accessibility**, and selecting **talkback**. This is found at the top of the screen; then press **On** or **Off** to turn the talkback functionality on or off. Then, navigate to the dialog confirmation, and click **OK** to confirm permission.

There's more…

There is more regarding accessibility that developers should consider as they build their applications, including a state with which they should be able to notify their users on whether a **`Switch`** button has been selected. This ensures their applications support accessibility and are up to standard.

Implementing declarative graphics using Jetpack Compose

In Android development, your application might have a different need, and this need might be building your own custom graphics for an intended purpose. This is very common in many stable and large Android code bases. The essential part of any custom view is its appearance. Furthermore, custom drawing can be a very easy or complex task based on the needs of your application. In Modern Android Development, Jetpack Compose makes it easier to work with custom graphics simply because the demand is immense. For example, many applications may need to control what happens on their screen accurately; the use case might be as simple as putting a circle on the screen or building more complex graphics to handle known use cases.

Getting ready

Open the **`Compose Basics`** project to get started with this recipe. You can find the entire code in the *Technical requirements* section.

How to do it...

In our project, let us create a new package and call it **circularexample**; inside this package, create a Kotlin file and call it **DrawCircleCompose**; inside the file, create a **CircleProgressIndicatorExample** composable function. You will not need to import anything for now:

1. Let us now go ahead and define our composable function. Since, in our example, we want to display a tracker in a circle, we need to float to fill in our circle. We will also define the colors to help us identify the progress:

```
@Composable
fun CircleProgressIndicatorExample(tracker: Float, progress: Float) {
    val circleColors = listOf(
        colorResource(id = R.color.purple_700),
        colorResource(id = R.color.teal_200)
    )
```

2. Now, let's call **Canvas** to draw our arc. We give our circle the size of **200.dp** with **8.dp** padding. Where it gets interesting is in **onDraw**. **startAngle** is set at **-90**; the start angle is set in degrees to understand it better.

The zero represents 3 o'clock, and you can also play around with your start angle to see how **-90** translates. The

useCenter Boolean indicates whether arc is to close the center of the bounds. Hence, in our case, we set it to **false**. Then, finally, we set the **style**, which can be anything based on our preference:

```
Canvas(
    modifier = Modifier
        .size(200.dp)
        .padding(8.dp),
    onDraw = {
        this.drawIntoCanvas {
            drawArc(
                color = colorSecondary,
                startAngle = -90f,
                sweepAngle = 360f,
                useCenter = false,
                style = Stroke(width = 55f, cap =
                    StrokeCap.Butt),
                size = Size(size.width, size.height)
            )
colorResource(id = R.color.teal_200)
 . . .
```

3. We have just drawn the first part of the circle; now, we need to draw the progress with a **Brush**, which utilizes **linearGradient**:

```
drawArc(
    brush = Brush.linearGradient(colors =
        circleColors),
    startAngle = -90f,
    sweepAngle = progress(tracker, progress),
    useCenter = false,
    style = Stroke(width = 55f, cap =
        StrokeCap.Round),
```

```
            size = Size(size.width, size.height)
        ) . . .
        . . .
```

4. Finally, our **`progress`** function tells **`sweepAngle`** where our progress should be based on our tracking abilities:

```
    private fun progress(tracker: Float, progress: Float): Float {
        val totalProgress = (progress * 100) / tracker
        return ((360 * totalProgress) / 100)
    }
    . . .
```

5. Run the **`preview`** function, and you should see a circular progress indicator as in *Figure 2.11*.

Figure 2.11 – Showing a circular progress image

IMPORTANT NOTE

The **Canvas** *composable function uses* **Canvas** *to Compose an object, which, in turn, creates and helps manage a view-base Canvas. It is also important to mention that Compose makes it easier for developers by maintaining the state and creating and freeing any necessary helper objects.*

How it works…

Generally, **`Canvas`** allows you to specify an area on the screen where you want to draw. In the old way of building Android applications, we also utilized **`Canvas`**, and now in Compose, it is more powerful and valuable.

`linearGradient` create a linear gradient with the specified colors along the provided start and end coordinates. For our example, we give it simple colors that come with the project.

The drawing functions have instrumental default parameters that you can use. For instance, by default, **`drawArc`**, as you can see, takes in several inputs:

```
fun drawArc(
    brush: Brush,
    startAngle: Float,
    sweepAngle: Float,
    useCenter: Boolean,
    topLeft: Offset = Offset.Zero,
    size: Size = this.size.offsetSize(topLeft),
    /*@FloatRange(from = 0.0, to = 1.0)*/
    alpha: Float = 1.0f,
    style: DrawStyle = Fill,
    colorFilter: ColorFilter? = null,
    blendMode: BlendMode = DefaultBlendMode
)
```

Figure 2.12 – Showing what drawArc takes as input

`sweepAngle` in our example, which is the size of the arc in the degree that is drawn clockwise relative to **`startAngle`**, returns a function that calculates progress. This function can be customized to fit your needs. In our example, we pass in a tracker and progress and return a float.

Since we want to fill the circle, we create **`cal totalProgress`**, which checks *progress * 100* divided by the tracker, and we return *360 (circle) * our progress divided by 100*. You can customize this function to fit your needs. You can also write code to listen to where you are and make the progress move based on your input value from a listener you create.

There's more...

There is more you can do with **`Canvas`** and custom drawing. One amazing way to enhance your knowledge on the topic is to look into old solutions posted on Stack Overflow, such as drawing a heart or any other shape, and see whether you can do the same in Compose.