Chapter 6: Putting Pieces Together

The previous chapters explored various aspects of Jetpack Compose. For example, *Chapter 2*, *Understanding the Declarative Paradigm*, compared the traditional View system to composable functions and explained the benefits of the declarative approach. *Chapter 4*,*Laying Out UI Elements*, gave you a solid understanding of some built-in layout composables such as `Box()`, `Row()`, and `Column()`. In *Chapter 5*, *Managing the State of Your Composable Functions*, we looked at state and learned about the important role it plays in a Compose app.

Now, it's time to see how these key elements work together in a real-world app. In this chapter, you will learn how Compose apps can be themed. We will also look at `Scaffold()`, an integrational UI element that picks up quite a few concepts that were originally related to activities, such as toolbars and menus, and we will learn how to add screen-based navigation.

In this chapter, we will cover the following topics:

- Styling a Compose app
- Integrating toolbars and menus
- Adding navigation

We will start by setting up a custom theme for a Compose app. You can define quite a few colors, shapes, and text styles that the built-in Material composables will use when drawing themselves. I will also show you what to keep in mind when you're adding additional Jetpack components that rely on app themes, such as *Jetpack Core Splashscreen*.

The following section, *Integrating toolbars and menus*, will introduce you to app bars and the options menu. You will also learn how to create snack bars.

In the final main section, *Adding navigation*, I will show you how to structure your app into screens. We will use the Compose version of *Jetpack Navigation* to navigate between them.

Technical requirements

This chapter includes one sample app, **`ComposeUnitConverter`**, as shown in the following screenshot:

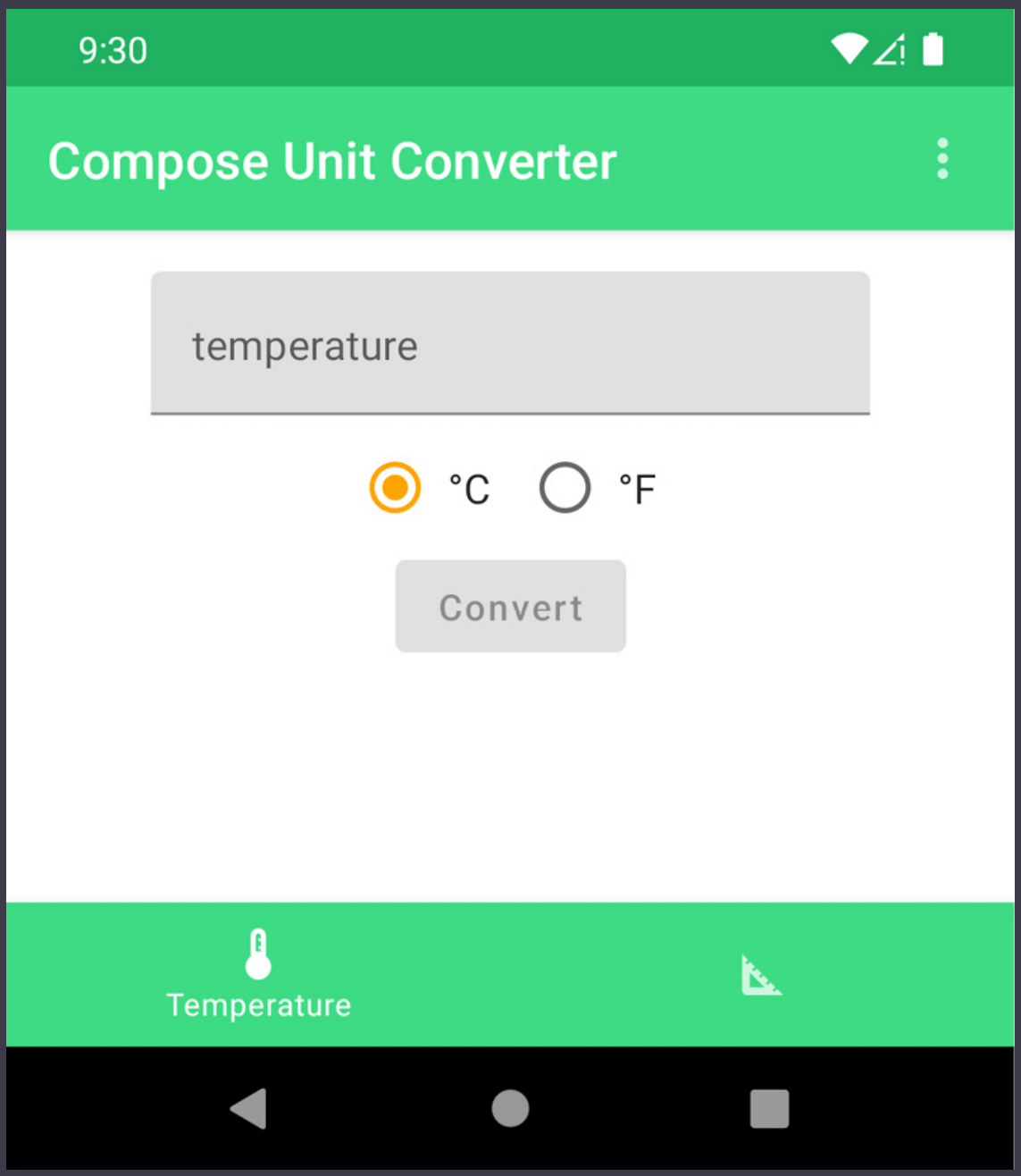

Figure 6.1 – The ComposeUnitConverter app

Please refer to the *Technical requirements* section of *Chapter 1*, *Building Your First Compose App*, for information about how to install and set up Android Studio, as well as how to get the repository that accompanies this book.

All the code files for this chapter can be found on GitHub at https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose/tree/main/chapter_06.

Styling a Compose app

Most of your Compose UI will likely use the built-in composable functions from the `androidx.compose.material` package. They implement the design language known as **Material Design** and its successor, **Material You** (which was introduced with Android 12). Material You is the native design language on Android, though it will also be available on other platforms. It expands on the ideas of a pen, paper, and cards, and makes heavy use of grid-based layouts, responsive animations, and transitions, as well as padding and depth effects. Material You advocates larger buttons and rounded corners. Custom color themes can be generated from the user's wallpaper.

Defining colors, shapes, and text styles

While apps should certainly honor both system and user preferences regarding visual appearance, you may want to add colors, shapes, or text styles that reflect your brand or corporate identity. So, how can you modify the look of the built-in Material composable functions?

The main entry point to Material Theming is `MaterialTheme()`. This composable may receive custom colors, shapes, and text styles. If a value is not set, a corresponding default (`MaterialTheme.colors`, `MaterialTheme.typography`, or `MaterialTheme.shapes`) is used. The follow-

ing theme sets custom colors but leaves the text styles and shapes as their defaults:

```
@Composable
fun ComposeUnitConverterTheme(
  darkTheme: Boolean = isSystemInDarkTheme(),
  content: @Composable () -> Unit
) {
  val colors = if (darkTheme) {
    DarkColorPalette
  } else {
    LightColorPalette
  }
  MaterialTheme(
    colors = colors,
    content = content
  )
}
```

The **`isSystemInDarkTheme()`** composable detects if the device is currently using a dark theme. Your app should use colors that suit this configuration. My example has two palettes, **`DarkColorPalette`** and **`LightColorPalette`**. Here's how the latter one is defined:

```
private val LightColorPalette = lightColors(
  primary = AndroidGreen,
  primaryVariant = AndroidGreenDark,
  secondary = Orange,
  secondaryVariant = OrangeDark
)
```

`lightColors()` is a top-level function inside the **`androidx.compose.material`** package. It provides a complete color definition for the Material color specification. You canfind more information about this at https://material.io/design/color/the-color-system.html#color-theme-creation. **`LightColorPalette`** overrides the default values for primary, **`prima-`**

ryVariant, secondary, and **secondaryVariant**. All the others (there are, for example, **background**, **surface**, and **onPrimary**) remain unchanged.

primary will be displayed most frequently across your app's screens and components. With **secondary**, you can accent and distinguish your app. It is, for example, used for radio buttons. The checked thumb color of switches is **secondaryVariant**, whereas the unchecked thumb color is taken from **surface**.

TIP

Material composables typically receive their default colors from composable functions called **colors()**, *which belong to their accompanying* **…Defaults** *objects. For example,* **Switch()** *invokes* **SwitchDefaults.colors()** *if no color parameter is passed to* **Switch()**. *By looking at these* **colors()** *functions, you can find out which color attribute you should set in your theme.*

You may be wondering how I defined, for example, **AndroidGreen**. The simplest way to achieve this is like this:

```
  val AndroidGreen = Color(0xFF3DDC84)
```

This works great if your app does not require other libraries or components that rely on the traditional Android theming system. We will turn to such scenarios in the *Using resource-based themes* section.

Besides colors, **MaterialTheme()** allows you to provide alternative shapes. Shapes direct attention and communicate state. Material composables are grouped into shape categories based on their size:

- Small (buttons, snack bars, tooltips, and more)
- Medium (cards, dialog, menus, and more)
- Large (sheets and drawers, and more)

To pass an alternative set of shapes to **MaterialTheme()**, you must instantiate **androidx.compose.material.Shapes** and provide implementations of

the **androidx.compose.foundation.shape.CornerBasedShape** abstract class for the categories you want to modify (**small**, **medium**, and **large**). **AbsoluteCutCornerShape**, **CutCornerShape**, **AbsoluteRoundedCornerShape**, and **RoundedCornerShape** are direct subclasses of **CornerBasedShape**.

The following screenshot shows a button with cut corners. While this makes the button look less familiar, it gives your app a distinctive look. You should, however, ensure that you want to add this:

Figure 6.2 – A button with cut corners

To achieve this, just add the following line when invoking **MaterialTheme()**:

```
  shapes = Shapes(small = CutCornerShape(8.dp)),
```

You can find more information about applying shapes to UIs at https://material.io/design/shape/applying-shape-to-ui.html#shape-scheme.

To alter the text styles that are used by Material composable functions, you need to pass an instance of **androidx.compose.material.Typography** to **MaterialTheme()**. **Typography** receives quite a few parameters, such as **h1**, **subtitle1**, **body1**, **button**, and **caption**. All of these are instances of **androidx.compose.ui.text.TextStyle**. If you do not pass a value for a parameter, a default is used.

The following code block increases the text size of buttons:

```
  typography = Typography(button = TextStyle(fontSize =
                                             24.sp)),
```

If you add this line to the invocation of **`MaterialTheme()`**, the text of all the buttons using your theme will be 24 scale-independent pixels tall. But how do you set the theme? To make sure that your complete Compose UI uses it, you should invoke your theme as early as possible:

```
class ComposeUnitConverterActivity :
ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)
    val factory = …
    setContent {
      ComposeUnitConverter(factory)
    }
  }
}
```

In my example, **`ComposeUnitConverter()`** is the root of the app's composable UI hierarchy since it is invoked inside **`setContent {}`**:

```
@Composable
fun ComposeUnitConverter(factory: ViewModelFactory) {
  …
  ComposeUnitConverterTheme {
    Scaffold( ...
```

`ComposeUnitConverter()` immediately delegates to **`ComposeUnitConverterTheme {}`**, which receives the remaining UI as its content. **`Scaffold()`** is a skeleton for real-world Compose UIs. We will be taking a closer look at this in the *Integrating toolbars and menus* section.

If you need to style parts of your app differently, you can nest themes by overriding your parent theme (*Figure 6.3*). Let's see how this works:

```
@Composable
@Preview
fun MaterialThemeDemo() {
```

```
    MaterialTheme(
      typography = Typography(
        h1 = TextStyle(color = Color.Red)
      )
    ) {
      Row {
        Text(
          text"= "He"lo",
          style = MaterialTheme.typography.h1
        )
        Spacer(modifier = Modifier.width(2.dp))
        MaterialTheme(
          typography = Typography(
            h1 = TextStyle(color = Color.Blue)
          )
        ) {
          Text(
            text"= "Comp"se",
            style = MaterialTheme.typography.h1
          )
        }
      }
    }
  }
```

In the preceding code snippet, the base theme configures any text that is styled as **`h1`** so that it appears in red. The second **`Text()`** uses a nested theme that styles **`h1`** to appear in blue. So, it overrides the parent theme:

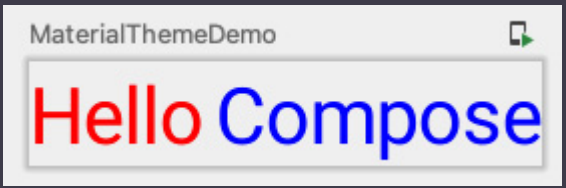

Figure 6.3 – Nesting themes

PLEASE NOTE

All the parts of your app must have a consistent look. Consequently, you should use nested themes carefully.

In the next section, we will continue exploring styles and themes. We will look at how themes are set in the manifest file, as well as how libraries may influence the way you define your Compose theme.

Using resource-based themes

App styling or theming has been present on Android since API level 1. It is based on resource files. Conceptually, there is a distinction between styles and themes. A **style** is a collection of attributes that specify the appearance (for example, font color, font size, or background color) of a single View. Consequently, styles do not matter for composable functions. A **theme** is also a collection of attributes, but'it's applied to an entire app, activity, or View hierarchy. Many elements of a Compose app are provided by Material composables; for them, a resource-based theme does not matter either. However, themes can apply styles to non-View elements, such as the status bar and window background. This may be relevant for a Compose app.

Styles and themes are declared in XML files inside the `res/values` directory and are typically named `styles.xml` and `themes.xml`, depending on the content. A theme is applied to the application or activity inside the manifest file with the `android:theme` attribute of the `<application />` or `<activity />` tag. If none of them receives a theme, `ComposeUnitConverter` will look as follows:

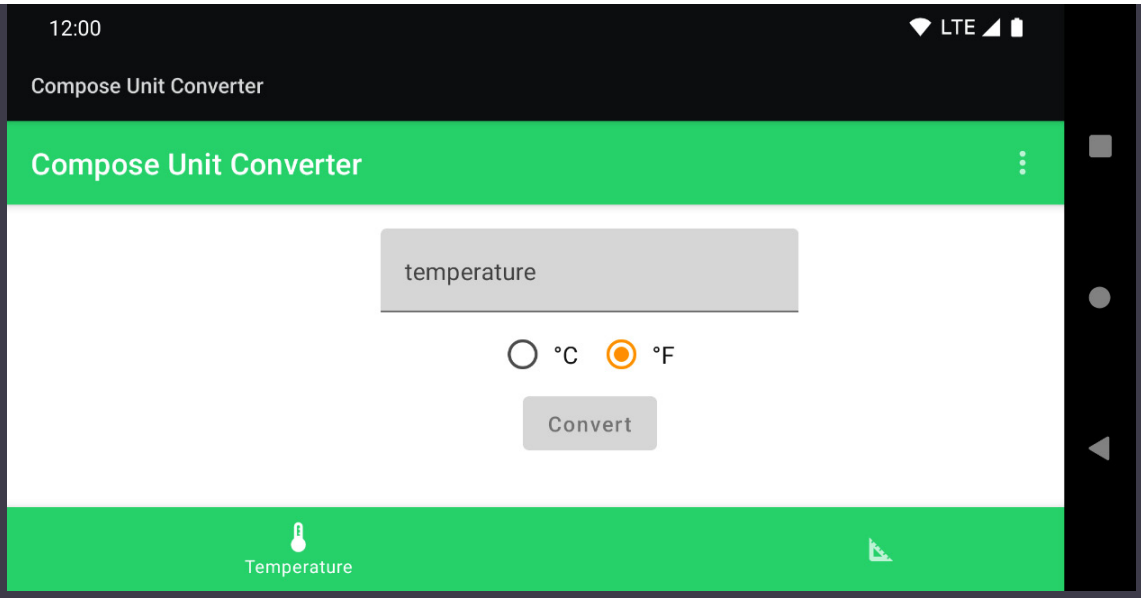

Figure 6.4 – Compose Unit Converter showing an additional title bar

To avoid the unwanted additional title bar, Compose apps must configure a theme without action bars, such as **`Theme.AppCompat.DayNight.NoActionBar`**, using **`android:the"e="@styl"/…"`** for **`<application />`** or **`<activity />`**. This way, **`ComposeUnitConverter`** looks like *Figure 6.1*. Have you noticed that the status bar has a dark gray background?

When **`Theme.AppCompat.DayNight`** is used, the status bar receives its background color from the **`colorPrimaryDark`** theme attribute (or **`android:statusBarColor`** since API level 21). If no value is specified, a default is used. Therefore, to make sure that the status bar is shown in a color that fits the remaining UI elements, you must add a file named **`themes.xml`** to **`res/values`**:

```
<resources>
  <style name="Theme.ComposeUnitConverter"

parent="Theme.AppCompat.DayNight.NoActionBar">
    <item

name="colorPrimaryDark">@color/android_green_dark
    </item>
```

```
    </style>
  </resources>
```

In the manifest file, the value of **`android:theme`** must then be changed to **`@style/Theme.ComposeUnitConverter`**. **`@color/android_green_dark`** represents the color. Instead of this expression, you could also pass the value directly; for example, **`#FF20B261`**. It is, however, best practice to store it in a file named **`colors.xml`** inside **`res/values`**:

```
  <resources>
    <color name="android_green_dark">#FF20B261</color>
    <color name="orange_dark">#FFCC8400</color>
  </resources>
```

This way, you can assign a different value for the Dark theme. The following version of **`themes.xml`** should be put in **`res/values-night`**:

```
  <resources>
    <style name="Theme.ComposeUnitConverter"

  parent="Theme.AppCompat.DayNight.NoActionBar">
      <item
  name="colorPrimaryDark">@color/orange_dark</item>
    </style>
  </resources>
```

The status bar now has a background color that fits the remaining UI elements. However, we need to define colors in two places: **`colors.xml`** and the Compose theme. Fortunately, this is rather easy to fix. Usually, we pass a literal, like this:

```
  val AndroidGreenDark = Color(0xFF20B261)
```

Instead of doing this, we should obtain the value from the resources. The **`colorResource()`** composable function belongs to the **`androidx.compose.ui.res`** package. It returns the color associated with a resource that's identified by an ID.

The following palette does not specify a **`secondary`** color:

```
private val LightColorPalette = lightColors(
  primary = AndroidGreen,
  primaryVariant = AndroidGreenDark,
  secondaryVariant = OrangeDark
)
```

Adding a color using **`colorResource()`** works as follows:

```
@Composable
fun ComposeUnitConverterTheme(
  darkTheme: Boolean = isSystemInDarkTheme(),
  content: @Composable () -> Unit
) {
  val colors = if (darkTheme) {
    DarkColorPalette
  } else {
    LightColorPalette.copy(secondary = colorResource(
      id = R.color.orange_dark))
  }
  MaterialTheme(
    colors = colors,
    …
```

You saw most of this in the *Defining colors, shapes, and text styles* section. The important difference is that I created a modified version of **`LightColorPalette`** (with a secondarycolor) using **`copy()`**, which is then passed to **`MaterialTheme()`**. If you store all the colors inside **`colors.xml`**, you should create your palettes completely inside your theme composable.

As you have seen, you may need to provide some values for resource-based themes, depending on how heavily you want to brand your app. Additionally, certain non-Compose Jetpack libraries use themes too, such as *Jetpack Core Splashscreen*. This component makes the advanced splash screen features of Android 12 available on older platforms. The images and colors of the splash screen are configured through theme attributes. The library requires that the theme of the starting activity has

`Theme.SplashScreen`as its parent. Additionally, the theme must provide the `postSplashScreenTheme`attribute, which refers to the theme to use once the splash screen has been dismissed.You can find more information about splash screens on Android at https://developer.android.com/guide/topics/ui/splash-screen.

TIP

To ensure consistent use of colors, the `colors.xml` *file should be the single point of truth in your app, if more than one component relies on resource-based themes.*

This concludes our look at Compose themes. In the next section, we will turn to an important integrational UI element called **Scaffold**. `Scaffold()` acts as a frame for your content, providing support for top and bottom bars, navigation, and actions.

Integrating toolbars and menus

Early Android versions did not know about action or app bars. They were introduced with API level 11 (Honeycomb). The options menu, on the other hand, has been around since the beginning but was opened by pressing a dedicated hardware button and shown at the bottom of the screen. With Android 3, it moved to the top and became a vertical list. Some elements could be made available permanently as actions. In a way, the options menu and the action bar merged. While originally, all the aspects of the action bar were handled by the hosting activity, the `AppCompat` support library introduced an alternative implementation (`getSupportActionBar()`). It is still widely used today as part of Jetpack.

Using Scaffold() to structure your screen

Jetpack Compose includes several app bar implementations that closely follow Material Design or Material You specifications. They can be added

to a Compose UI through **`Scaffold()`**, a composable function that acts as an app frame or skeleton. The following code snippet is the root of the **`ComposeUnitConverter`** UI. It sets up the theme and then delegates it to **`Scaffold()`**:

```
@Composable
fun ComposeUnitConverter(factory: ViewModelFactory) {
  val navController = rememberNavController()
  val menuItems = listOf("Item #1", "Item #2")
  val scaffoldState = rememberScaffoldState()
  val snackbarCoroutineScope =
rememberCoroutineScope()
  ComposeUnitConverterTheme {
    Scaffold(scaffoldState = scaffoldState,
      topBar = {
        ComposeUnitConverterTopBar(menuItems) { s ->
          snackbarCoroutineScope.launch {
            scaffoldState.snackbarHostState.showSnack
bar(s)
          }
        }
      },
      bottomBar = {
        ComposeUnitConverterBottomBar(navController)
      }
    ) {
      ComposeUnitConverterNavHost(
        navController = navController,
        factory = factory
      )
    }
  }
}
```

`Scaffold()` implements the basic Material Design visual layout structure. You can add several other Material composables, such as **`TopAppBar()`** or

`BottomNavigation()`. Google calls this a **slot API** because a composable function is customized by inserting another composable into an area or space (slot) of the parent. Passing an already configured child provides more flexibility than exposing (lots of) configuration parameters. Depending on which children you slot in, **`Scaffold()`** may need to remember different states. You can pass a **`ScaffoldState`**, which can be created with **`rememberScaffoldState()`**.

My example uses **`ScaffoldState`** to show a snack bar, a brief temporary message that appears toward the bottom of the screen. As **`showSnackbar()`** is a suspending function, it must be called from a coroutine or another suspending function. Therefore, we must create and remember a **`CoroutineScope`** using **`rememberCoroutineScope()`** and invoke its **`launch {}`** function.

In the next section, I will show you how to create a top app bar with an options menu.

Creating a top app bar

App bars at the top of the screen are implemented using **`TopAppBar()`**. You can provide a navigation icon, a title, and a list of actions here:

```
@Composable
fun ComposeUnitConverterTopBar(menuItems:
List<String>,
                               onClick: (String) ->
Unit) {
  var menuOpened by remember { mutableStateOf(false)
}
  TopAppBar(title = {
    Text(text = stringResource(id =
R.string.app_name))
  },
    actions = {
```

```
        Box {
          IconButton(onClick = {
            menuOpened = true
          }) {
            Icon(Icons.Default.MoreVert, "")
          }
          DropdownMenu(expanded = menuOpened,
            onDismissRequest = {
              menuOpened = false
            }) {
            menuItems.forEachIndexed { index, s ->
              if (index > 0) Divider()
              DropdownMenuItem(onClick = {
                menuOpened = false
                onClick(s)
              }) {
                Text(s)
              }
            }
          }
        }
      }
    )
  }
```

`TopAppBar()` has no specific API for an options menu. Instead, the menu is treated as an ordinary action. Actions are typically **`IconButton()`** composables. They are displayed at the end of the app bar in a horizontal row. An **`IconButton()`** receives an **`onClick`** callback and an optional **`enabled`** parameter, which controls if the user can interact with the UI element.

In my example, the callback only sets a **`Boolean`** mutable state (**`menuOpened`**) to **`false`**. As you will see shortly, this closes the menu. **`content`** (usually an icon) is drawn inside the button. The **`Icon()`** composable receives an instance of **`ImageVector`** and a content description. You can get icon data from the resources, but you should use predefined graphics if possible –

in my example, **`Icons.Default.MoreVert`**. Next, let's learn how to display a menu.

A Material Design drop-down menu (**`DropdownMenu()`**) allows you to display multiple choices compactly. It usually appears when you interact with another element, such as a button. My example places **`DropdownMenu()`** in a **`Box()`** with an **`IconButton()`**, which determines the location on-screen. The expanded parameter makes the menu visible (open) or invisible (closed). **`onDismissRequest`** is called when the user requests to dismiss the menu, such as by tapping outside the menu's bounds.

The content should consist of **`DropdownMenuItem()`** composables. **`onClick`** is called when the corresponding menu item is clicked. Your code must make sure that the menu is closed. If possible, you should pass the domain logic to be executed as a parameter to make your code reusable and stateless. In my example, a snack bar is shown.

This concludes our look at top app bars. In the next section, I will show you how to use **`BottomNavigation()`** to navigate to different screens using the Compose version of Jetpack Navigation.

PLEASE NOTE

To use the Compose version of Jetpack Navigation in your app, you must add an implementation dependency of **`androidx.navigation:navigation-compose`** *to your module-level* **`build.gradle`** *file.*

Adding navigation

`Scaffold()` allows you to put content in a slot at the bottom of the screen using its **`bottomBar`** parameter. This can, for example, be a **`BottomAppBar()`**. Material Design bottom app bars provide access to a bottom navigation drawer and up to four actions, including a floating action button. **`ComposeUnitConverter`** adds **`BottomNavigation()`** instead. Material Design

bottom navigation bars allow movement between primary destinations in an app.

Defining screens

Conceptually, primary destinations are *screens*, something that, before Jetpack Compose, may have been displayed in separate activities. Here's how screens are defined in **`ComposeUnitConverter`**:

```
sealed class ComposeUnitConverterScreen(
  val route: String,
  @StringRes val label: Int,
  @DrawableRes val icon: Int
) {
  companion object {
    val screens = listOf(
      Temperature,
      Distances
    )
    const val route_temperature = "temperature"
    const val route_distances = "distances"
  }
  private object Temperature :
ComposeUnitConverterScreen(
    route_temperature,
    R.string.temperature,
    R.drawable.baseline_thermostat_24
  )
  private object Distances :
ComposeUnitConverterScreen(
    route_distances,
    R.string.distances,
    R.drawable.baseline_square_foot_24
  )
}
```

ComposeUnitConverter consists of two screens – **Temperature** and **Distances**. **route** uniquely identifies a screen. **label** and **icon** are shown to the user. Let's see how this is done:

```
@Composable
fun ComposeUnitConverterBottomBar(navController:
   NavHostController) {
  BottomNavigation {
    val navBackStackEntry by
          navController.currentBackStackEntryAsState(
)
    val currentDestination =
navBackStackEntry?.destination
    ComposeUnitConverterScreen.screens.forEach {
screen ->
      BottomNavigationItem(
        selected = currentDestination?.hierarchy?.any
{
          it.route == screen.route } == true,
        onClick = {
          navController.navigate(screen.route) {
            launchSingleTop = true
          }
        },
        label = {
          Text(text = stringResource(id =
screen.label))
        },
        icon = {
          Icon(
            painter = painterResource(id =
screen.icon),
            contentDescription = stringResource(id =
              screen.label)
          )
```

```
            },
            alwaysShowLabel = false
        )
      }
    }
  }
```

The content of **`BottomNavigation()`** consists of **`BottomNavigationItem()`** items. Each item represents a *destination*. We can add them with a simple loop:

```
ComposeUnitConverterScreen.screens.forEach { screen -
>
```

As you can see, the **`label`** and **`icon`** properties of a **`ComposeUnitConverterScreen`** instance are used during the invocation of **`BottomNavigationItem()`**. **`alwaysShowLabel`** controls if the label is visible when an item is selected. An item will be selected if the corresponding screen is currently displayed. When a **`BottomNavigationItem()`** is clicked, its **`onClick`** callback is invoked. My implementation calls **`navigate()`** on the provided **`NavHostController`** instance, passing **`route`** from the corresponding **`ComposeUnitConverterScreen`** object.

So far, we have defined screens and mapped them to **`BottomNavigationItem()`** items. When an item is clicked, the app navigates to a given route. But how do routes relate to composable functions? I will show you in the next section.

Using NavHostController and NavHost()

An instance of **`NavHostController`** allows us to navigate to different screens by calling its **`navigate()`** function. We can obtain a reference to it inside **`ComposeUnitConverter()`** by invoking **`rememberNavController()`**, and then passing it to **`ComposeUnitConverterBottomBar()`**. The mapping between a route and a composable function is established through **`NavHost()`**. It belongs to the **`androidx.navigation.compose`** package. Here's how this composable is invoked:

```
@Composable
fun ComposeUnitConverterNavHost(
  navController: NavHostController,
  factory: ViewModelProvider.Factory?
) {
  NavHost(
    navController = navController,
    startDestination =
        ComposeUnitConverterScreen.route_temperature
  ) {
    composable(ComposeUnitConverterScreen.route_tempe
rature) {
      TemperatureConverter(
        viewModel = viewModel(factory = factory)
      )
    }
    composable(ComposeUnitConverterScreen.route_dista
nces) {
      DistancesConverter(
        viewModel = viewModel(factory = factory)
      )
    }
  }
}
```

`NavHost()` receives three parameters:

- A reference to our **`NavHostController`**
- The route for the start destination
- The builder that was used to construct the navigation graph

Before Jetpack Compose, the navigation graph was usually defined through an XML file. **`NavGraphBuilder`** provides access to a simple domain-specific language. **`composable()`** adds a composable function as a destination. Besides the route, you can pass a list of arguments and a list of deep links.

TIP

A detailed description of Jetpack Navigation is beyond the scope of this book. You can find more information at https://developer.android.com/guide/navigation.

Summary

This chapter showcased how key elements of Jetpack Compose work together in a real-world app. You learned how to theme Compose apps and how to keep your Compose theme in sync with resource-based themes.

I also showed you how `Scaffold()` acts as an app frame or skeleton. We used its slot API to plug in a top app bar with a menu, as well as a bottom bar to navigate between screens using the Compose version of Jetpack Navigation.

In the next chapter, *Tips, Tricks, and Best Practices*, we will discuss how to separate UI and business logic. We will revisit `ComposeUnitConverter`, this time focusing on its use of ViewModels.