10

Implementing Your First Wear OS Using Jetpack Compose

Wear OS is an operating system developed by Google for smartwatches and other wearable devices. There are several reasons why creating Wear OS for Android is essential in our Modern Android Development. First, this means expanding the Android ecosystem; Wear OS extends the Android ecosystem by allowing developers to create apps and services that can be accessed through a smartwatch or other wearable device.

This expands the reach of Android and creates new opportunities for developers and users. In addition, it provides seamless integration with Android smartphones, allowing users to easily access notifications, calls, and other information on their smartwatches, hence providing a more convenient and efficient way for users to interact with your application.

Let's not forget the most notable apps that can benefit from this are health and fitness tracking apps, including heart rate monitoring, step tracking, and workout tracking. This allows users to track their fitness goals and stay motivated to achieve them. Finally, Wear OS allows users to customize their smartwatch with different watch faces, apps, and widgets. This provides a personalized experience that meets individual needs and preferences.

Wearable technology is a rapidly growing market, and as the technology continues to evolve, Wear OS has the potential to become a key player in the wearable technology market. Wear OS is still very new, and in this chapter, we will explore simple basic examples as many of the APIs might change in the future. Therefore, getting an idea of how it works and how to create cards, buttons, and show lists will be helpful.

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

- Getting started with your first Wear OS in Android Studio
- Creating your first button
- Implementing a scrollable list
- Implementing cards in Wear OS (`TitleCard` and `AppCard`)
- Implementing a chip and a toggle chip
- Implementing `ScalingLazyColumn` to showcase your content

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_ten**.

Getting started with your first Wear OS in Android Studio

The Android OS is used worldwide, and one of the use cases is Wear OS (by *wear*, we mean smart-

watches). This is good news for Android developers because this means more jobs. Furthermore, many applications now have to support Wear OS, such as Spotify, fitness tracking apps, heart monitoring apps, and more, which implies more use cases will arise, and companies will adopt building for *Wear OS* even if it's only for notification purposes. Therefore, this chapter will explore how to get started.

Getting ready

In this recipe, we will look into getting started with Wear OS and how to set up your virtual watch testing environment.

How to do it...

To create your first project on Wear OS in Jetpack Compose, follow these steps:

1. First, create a new Android project in Android Studio and ensure you have the latest version of Android Studio and the Wear OS SDK installed.
2. Then, following the procedure of creating your first application, *Chapter 1, Getting Started with Modern Android Development Skills,* pick **Wear OS** instead of **Phone and Tablet**, as shown in *Figure 10.1*.

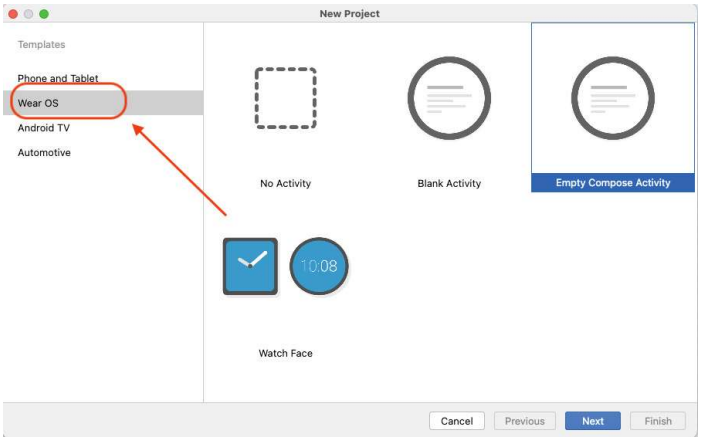

Figure 10.1 – Selecting Wear OS

3. Choose **Empty Compose Activity** (see *Figure 10.1*); as you might know, it is much better to use Compose while building for Wear OS since Google recommends this. Then, hit **Next**, and name your Wear OS project **`WearOSExample`**. You will notice it uses a minimum SDK of **`API 30: Android 11.0 (R)`**.

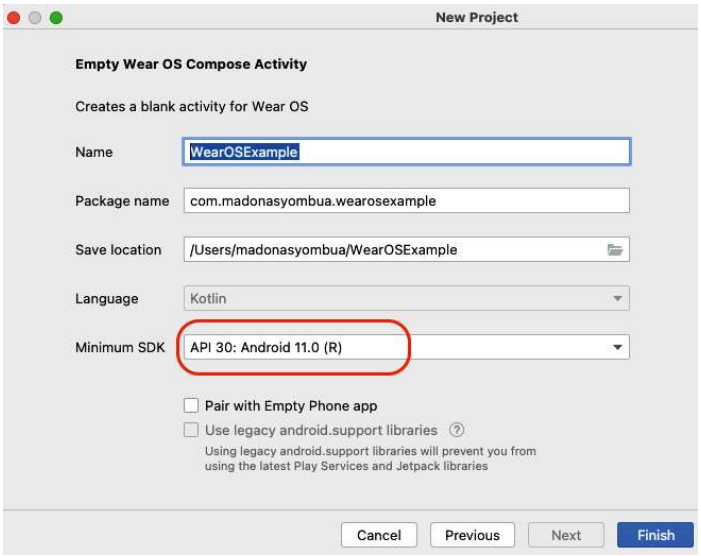

Figure 10.2 – Minimum SDK version

4. Click **Finish**, and you should be able to see a sample code template provided for you.
5. Now, let's go ahead and get our virtual Wear OS testing device set up to run the already provided code template. Navigate to **Tools** | **Device Manager**, then create a new device. If you need help in this section, refer to *Chapter 1*, *Getting Started with Modern Android Development Skills*.
6. Now, see *Figure 10.3* to choose your Wear OS virtual testing device. Note that you can also choose either a round, square, or rectangular device. We will use round.

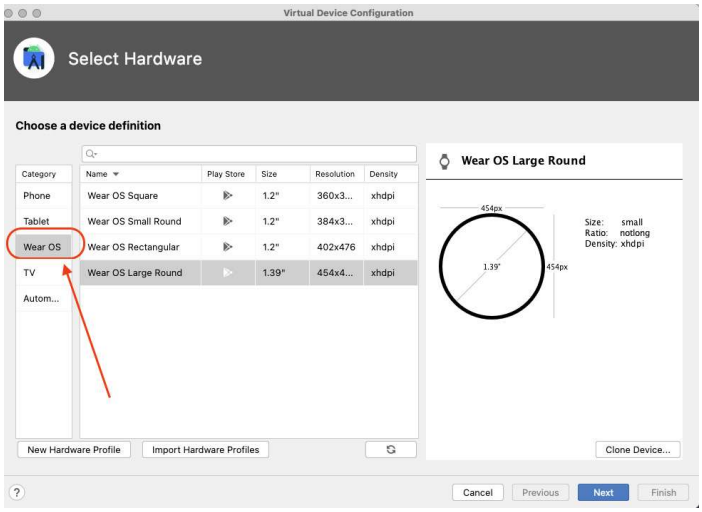

Figure 10.3 – Wear OS virtual device set up

7. Hit **Next**, then download the system image – in our case, **R**, which is API level 30.

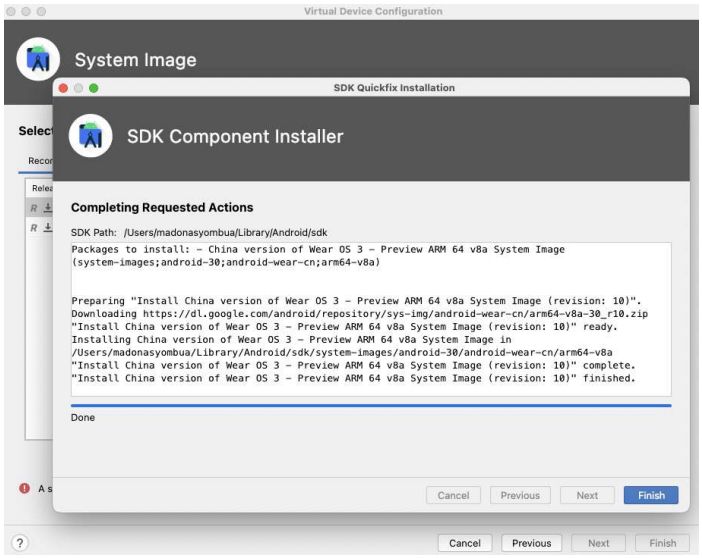

Figure 10.4 – Installing the system image for testing

8. Then press **Finish**, and you should have a ready-to-use Wear OS virtual testing device.
9. Now, go ahead and change the text in `Greeting()` to say `"Hello, Android Community"` in the code template and run, and you should have something similar to *Figure 10.5*. If everything is installed correctly, you should not have a build error.

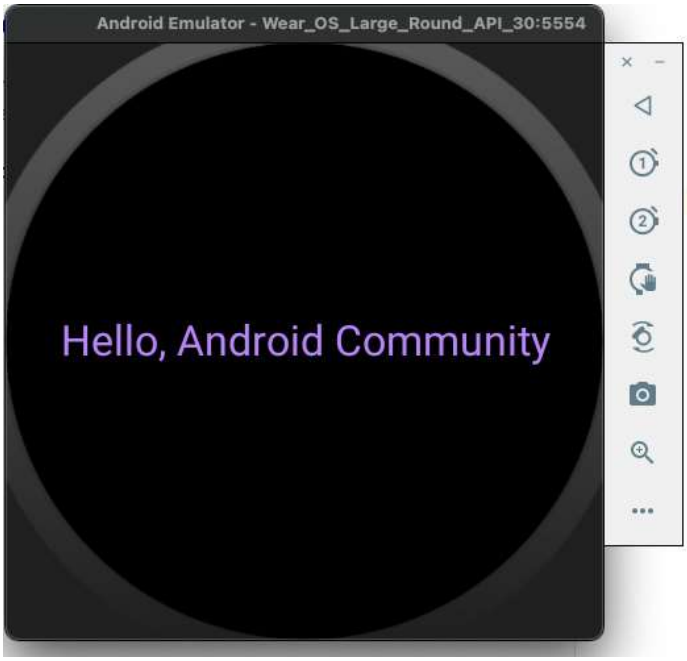

Figure 10.5 – Displaying a greeting on a Wear OS virtual device

10. Also, ensure you change the text on the round string resource too.

That's it, you have successfully set up your first Wear OS, and we were able to run the already provided `Greeting()`. In the following recipe, we will look at creating a simple button.

How it works...

You will notice the template looks precisely how you'd build Android applications, the only difference being the libraries used. The template uses Compose, which makes our work easier while developing since we will be using most of the concepts that we learned in previous chapters.

The following is a comparison to help you know the difference between the Wear OS dependency and the standard dependency:

Wear OS Dependency(androidx.wear.*)	Comparison	Standard Dependency(androidx.*)
androidx.wear.compose:compose-material	*instead of*	androidx.compose.material:material
androidx.wear.compose:compose-navigation	*instead of*	androidx.navigation:navigation-compose
androidx.wear.compose:compose-foundation	*in addition to*	androidx.compose.foundation:foundation

Figure 10.6 – Different types of dependencies (source: developer.android.com)

Creating your first button

In this recipe, we will create our first button in Wear OS to explore the principles and best practices of building in Wear OS.

Getting ready

You need to have completed the previous recipe to get started on this one. We will be building upon our already created **`WearOSExample`** project.

How to do it...

To create your first button on Wear OS in Jetpack Compose, you can follow these steps:

1. Using the already-created project, we will be adding a new button. Let's go ahead and remove some of the already provided code, **`fun Greeting(greetingName: String)`**:

```
@Composable
fun Greeting(greetingName: String) {
    Text(
        modifier = Modifier.fillMaxWidth(),
        textAlign = TextAlign.Center,
        color = MaterialTheme.colors.primary,
        text = stringResource(R.string.hello_world, greetingName)
    )
}
```

Figure 10.7 – A screenshot showing what to be deleted

2. Removing the **`Greeting()`** function called in **`WearOSExampleTheme`** will complain; go ahead and remove that too.
3. Then create a new **`Composable`** function that will define your button. You can use the **`Button`** function provided by Jetpack Compose:

```
@Composable
fun SampleButton() {
    Button(
       onClick = { /* Handle button click */ },
       modifier = Modifier.fillMaxWidth()
    ) {
        Text("Click me")
    }
}
```

4. Then, call the new function in our **`WearApp()`** function:

```
@Composable
fun WearApp() {
    WearOSExampleTheme {
        /* If you have enough items in your list, use
           [ScalingLazyColumn] which is an optimized
           version of LazyColumn for wear devices with
           some added features. For more information,
           see d.android.com/wear/compose./
        */
        Column(
            modifier = Modifier
                .fillMaxSize()
                .background(
                    MaterialTheme.colors.background),
            verticalArrangement = Arrangement.Center
        ) {
            SampleButton()
        }
    }
}
```

5. Then, in our activity, call the **`setContent`** method with your button's **`Composable`** function as the parameter:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?)
```

```
        {
            super.onCreate(savedInstanceState)
            setContent {
                WearApp()
            }
        }
    }
```

6. You can also utilize the already provided **`Preview`** function to view the changes. You will notice that we explicitly specify the device, **`@Preview(device = Devices.WEAR_OS_SMALL_ROUND, showSystemUi = true)`**:

```
@Preview(device = Devices.WEAR_OS_SMALL_ROUND, showSystemUi = true)
@Composable
fun DefaultPreview() {
    WearApp()
}
```

7. Run your Wear OS app, and you should see your button displayed on the screen, as shown in *Figure 10.8*:

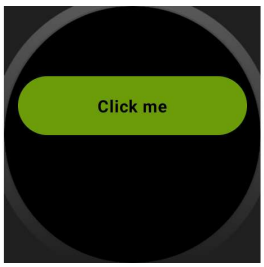

Figure 10.8 – A button in Wear OS

8. Let's look at another example, which is a button with an icon; this is pretty similar to the first button, but in this case, we will just be adding an icon instead of text.
9. Create a new function called **`SampleButton2()`** and add the following code:

```
@Composable
fun SampleButton2(
) {
    Row(
        modifier = Modifier
```

```
                .fillMaxWidth()
                .padding(bottom = 10.dp),
            horizontalArrangement = Arrangement.Center
        ) {
            Button(
                modifier = Modifier.size(
                    ButtonDefaults.LargeButtonSize),
                onClick = { /* Handle button click */ },
            ) {
                Icon(
                    imageVector =
                        Icons.Rounded.AccountBox,
                    contentDescription = stringResource(
                        id = R.string.account_box_icon),
                    modifier = Modifier
                        .size(24.dp)
                        .wrapContentSize(
                            align = Alignment.Center)
                )
            }
        }
    }
```

10. Finally, comment out **SampleButton**, add **SampleButton2**, and run; you should see something similar to *Figure 10.9*:

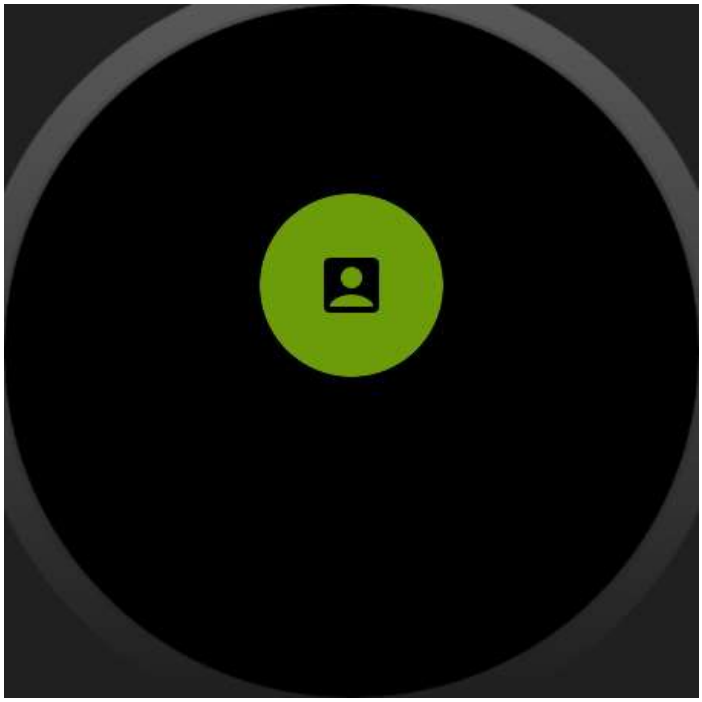

Figure 10.9 – A button with an icon in Wear OS

IMPORTANT NOTE

It is important to note that the Wear OS platform has some unique considerations when it comes to designing and testing apps, such as the smaller screen size and the need to optimize battery life. It's essential to test your app on an actual device to ensure it works as expected on Wear OS.

How it works...

Judging from your previous knowledge of Compose, everything we have worked on so far should look familiar. In our example, we're using `SampleButton` and `WearOSExampleTheme` from the Wear OS Compose library to create a button that's designed specifically for Wear OS devices.

`SampleButton` takes in an `onClick` lambda that gets called when the button is clicked and a modifier that sets the size of the button based on what we specify, which, in our example, is a simple `fillMaxWidth()`.

We're using `horizontalArrangement` in the column to center our button and using the `MaterialTheme` color to paint the background. In the case of Wear OS, Google recommends using the default material wear shapes; these are already optimized for non-round and round devices, which makes our work easier as developers. See the following link for more information on shapes: **https://developer.android.com/reference/kotlin/androidx/wear/compose/material/Shapes**.

Finally, we're using the `Text` composable to display the button text, which is vital since it tells users what the button's intended use is.

Implementing a scrollable list

Implementing a scrollable list is essential for creating an effective and user-friendly Android app that meets the needs of your users. A scrollable list allows you to display a large amount of information on a small screen, which can be beneficial, especially in a tiny device such as a watch. By scrolling through the list, users can quickly and easily access all of the items without navigating to different screens or pages.

Users expect a smooth and responsive scrolling experience when interacting with lists. Implementing a scrollable list with optimized performance can help ensure the app feels fast and responsive to the user. Scrollable lists can be customized to suit a variety of use cases and design requirements. You can adjust the layout, appearance, and behavior of the list to fit your app's specific needs and provide a unique user experience. In this recipe, we will look at how you can implement a scrollable list in Wear OS.

Getting ready

You need to have completed the previous recipe to get started on this one. We will be using our already created **`WearOSExample`** project to continue this part.

How to do it...

Follow these steps to build a scrollable list in Wear OS using Jetpack Compose:

1. In your **`MainActivity.kt`** file, let's create a new **`Composable`** function containing your scrollable

list. You can call it anything you like, but for this example, we'll call it **`WearOSList`**.

2. Another option is to create a new package to organize our code better and call the package **`components`**. Inside **`components`**, create a new Kotlin file and call it **`WearOsList`**.
3. In our **`WearOSList`** function, we will need a list of strings for our example; we can just create sample dummy data to showcase an example:

```
@Composable
fun WearOSList(itemList: List<String>) {. . .}
```

4. Inside our **`WearOSList`** function, create **`ScalingLazyColumn`**, which is optimized for Wear OS. This will be the container for our scrollable list. We will talk about **`ScalingLazyColumn`** later in the chapter:

```
@Composable
fun  WearOSList(itemList: List<String>) {
    ScalingLazyColumn() {
        // TODO: Add items to the list here
    }
}
```

Building for Wear OS might be challenging due to content size, hence the need to be familiar with Wear's best practices.

5. For our items, we will create a new **`Composable`** function called **`WearOSListItem`**, which will just have a **`text`** since we are just showcasing a text:

```
@Composable
fun WearOSListItem(item: String) {
    Text(text = item)
}
```

6. For our data, we will create a dummy list, so go ahead and add the following in the **`WearApp()`** function:

```
val itemList = listOf(
    "Item 1",
    "Item 2",
    "Item 3",
    "Item 4",
    "Item 5",
    . . .
)
```

7. Finally, comment out the two buttons we created, call **`WearOSList`**, pass in **`itemList`**, and run the application:

```
{
    // SampleButton()
    //SampleButton2()
    WearOSList(
        itemList = itemList,
        modifier = contentModifier
    )
}
```

8. You should see a list similar to *Figure 10.10*:

Figure 10.10 – Scrollable list of items

How it works...

In this example, we're using `WearOsList` and `WearOSExampleTheme` from the Wear OS Compose library to create a list that's designed specifically for Wear OS devices.

We start by creating a `WearOSList` composable that takes in a list of items as a parameter. Inside `ScalingLazyColumn`, we use the `items` function to iterate through the list of items and create a `WearOSListItem` for each.

The `WearOSListItem` composable has a `Composable text` function.

Implementing Cards in Wear OS (TitleCard and AppCard)

When building for Wear OS, we have two significant cards that we need to consider: `AppCard` and `TitleCard`. A good use case for cards would be **Notification** and **Smart Reply**. If you use a wearable device, you might know what these are; if you don't use a wearable device, you can look them up, but in this recipe, we will also explore examples.

Furthermore, if you create a Notification card, you intend to provide a quick and easy way to view and respond to notifications from your apps. When a notification arrives, it appears as a card on your watch face, which you can then swipe away or tap to open and interact with the notification.

As for Smart Reply cards, this feature uses machine learning to suggest responses to messages you receive based on the context of the message. These cards appear as a response option to notifications

and allow you to quickly send a message without needing to type it out manually.

Both Notification and Smart Reply cards are essential because they provide an efficient and streamlined way to manage notifications and respond to messages without having to pull out your phone constantly. They allow you to stay connected while on the go and keep you informed of important information without disrupting your daily routine, which is why Wear OS is here to stay, and knowing how to build for it will come in handy. In this recipe, we will create a simple card and see how to handle navigation in Wear OS.

Getting ready

You will need to have completed the previous recipes to continue with this recipe.

How to do it...

Here's an example of creating a card in Wear OS using Jetpack Compose. Open the **`WearOSExample`** project and code along:

1. Inside the **`components`** package, let's create a new Kotlin file and call it **`MessageCardExample`**.
2. Inside **`MessageCardExample`**, create a new composable function called **`MessageCard`**:

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

3. We must now call **`AppCard()`** since this is what we want. **`AppCard`** takes in **`appName`**, **`time`**, **`title`**, and more, as shown in *Figure 10.11*. This means you can customize your **`AppCard ()`** to fit your needs:

```
@Composable
public fun AppCard(
    onClick: () -> Unit,
    appName: @Composable () -> Unit,
    time: @Composable () -> Unit,
    title: @Composable () -> Unit,
    modifier: Modifier = Modifier,
    appImage: @Composable (() -> Unit)? = null,
    backgroundPainter: Painter = CardDefaults.cardBackgroundPainter(),
    appColor: Color = MaterialTheme.colors.onSurfaceVariant,
    timeColor: Color = MaterialTheme.colors.onSurfaceVariant,
    titleColor: Color = MaterialTheme.colors.onSurface,
    contentColor: Color = MaterialTheme.colors.onSurfaceVariant,
    content: @Composable () -> Unit,
) {
    Card(
        onClick = onClick,
        modifier = modifier,
        backgroundPainter = backgroundPainter,
        enabled = true,
    ) {
```

Figure 10.11 – AppCard composable function

4. This makes our work easier as developers since we know exactly what we need when building, thereby increasing developer productivity:

```
@Composable
fun MessageCard() {
    AppCard(
        onClick = { /*TODO*/ },
        appName = { /*TODO*/ },
        time = { /*TODO*/ },
        title = { /*TODO*/ }) {
        }
}
```

5. Now, let's go ahead and implement our **`AppCard()`** and send a message to our users. For our example, we will hardcode the data, but if you have an endpoint, you can pull data and display it as needed:

```
@Composable
fun MessageCard() {
    AppCard(
        modifier = Modifier
            .fillMaxWidth()
            .padding(bottom = 8.dp),
        appImage = {
            Icon(
                modifier = Modifier
                    .size(24.dp)
```

```
                        .wrapContentSize(
                            align = Alignment.Center),
                    imageVector = Icons.Rounded.Email,
                    contentDescription = stringResource(
                        id = R.string.message_icon)
                )
            },
            onClick = { /*Do something*/ },
            appName = {  stringResource(
                id = R.string.notification_message) },
           time = {  stringResource(id = R.string.time) },
           title = { stringResource(
               id = R.string.notification_owner) }) {
           Text(text = stringResource(
               id = R.string.hi_android))
           }
    }
```

6. In **`MainActivity`**, comment out other composable functions, for now add **`MessageCard()`**, and run it:

Figure 10.12 – AppCard with a notification

How it works...

`TitleCard` and **`AppCard`** are both used to display information on Wear OS, but they have different pur-

poses. In our example, we use **`AppCard()`**, but as you can see in *Figure 10.13*, **`TitleCard()`** takes in several inputs that are similar to **`AppCard()`**:

```
@Composable
public fun TitleCard(
    onClick: () -> Unit,
    title: @Composable () -> Unit,
    modifier: Modifier = Modifier,
    time: @Composable (() -> Unit)? = null,
    backgroundPainter: Painter = CardDefaults.cardBackgroundPainter(),
    titleColor: Color = MaterialTheme.colors.onSurface,
    timeColor: Color = MaterialTheme.colors.onSurfaceVariant,
    contentColor: Color = MaterialTheme.colors.onSurfaceVariant,
    content: @Composable () -> Unit,
) {
    Card(
        onClick = onClick,
        modifier = modifier,
        backgroundPainter = backgroundPainter,
        enabled = true,
    ) {
```

Figure 10.13 – TitleCard input

You can use **`TitleCard()`** to display information that is relevant to the current context, such as the name of a song that is playing or the title of a movie that is being watched. It is typically displayed at the top of the screen and can be dismissed by swiping it away. A good example is Spotify.

When using **`AppCard()`**, you can display information about an app that is currently running, such as the name of the app and a brief description of what it does, as we did in our example. It is typically displayed on a smaller card that can be tapped to open the app. That is why it has **`onClick{/**TODO*/}`**, which can lead to more information.

When deciding whether to use **`TitleCard()`** or **`AppCard()`**, you should consider the following factors:

- The amount of information that you need to display
- The relevance of the information to the current context
- The desired user experience

If you need to display a lot of information, `TitleCard()` may be a better option. If you only need to display a small amount of information, `AppCard()` may be a better option. If you want the information to be relevant to the current context, `TitleCard()` may be a better option. If you want the information displayed on a smaller card that can be tapped to open the app, `AppCard()` may be a better option.

Implementing a chip and a toggle chip

In this recipe, we will explore significant Wear components; a chip and a toggle chip are both used to display and interact with data.

A **chip** is a small, rectangular element that can be used to display text, icons, and other information. It is typically used to display items that are related or that have a common theme.

A **toggle chip** is a component that can be used to represent a binary value. It is typically used to represent things such as on/off, yes/no, or true/false.

It is fair to mention that you can use these components in your regular application, and we will explore them more in *Chapter 11*. When deciding which component to use, you should consider the following factors:

- The type of data that you want to display
- The type of interaction that you want to enable
- The look and feel that you want to achieve

Getting ready

We will be using our already-created project for this section.

How to do it...

We will create a chip and a toggle chip in this recipe. Follow these steps:

1. Let's go ahead and build our first chip; inside the **`components`** package, create a Kotlin file and call it **`ChipExample.kt`**.
2. Inside the file, create a composable function called **`ChipWearExample()`**.
3. Now, let's go ahead and call the **`Chip()`** composable function. You can also use the **`Chip`** component to display dynamic information. To do this, you can use the **`modifier`** property to specify a function that will be called to update the information displayed on the chip:

```
@Composable
fun ChipWearExample(){
    Chip(
        modifier = Modifier
            .fillMaxWidth()
            .padding(bottom = 8.dp),
        onClick = { /*TODO */ },
        label = {
            Text(
                text = stringResource(
                    id = R.string.chip_detail),
                maxLines = 1,
                overflow = TextOverflow.Ellipsis
            )
        },
        icon = {
            Icon(
                imageVector = Icons.Rounded.Phone,
                contentDescription = stringResource(
                    id = R.string.phone),
                modifier = Modifier
                    .size(24.dp)
                    .wrapContentSize(
                        align = Alignment.Center)
```

```
                )
            },
        )
    }
```

4. In **`MainActivity`**, go ahead and comment out the existing **`Composable`** functions, add **`ChipWearExample()`**, and run the app:

Figure 10.14 – A chip with a message

5. Now, let's go ahead and create a toggle chip; inside our **`component`** package, create a Kotlin file and call it **`ToggleChipExample`**.
6. Inside **`ToggleChipExample`**, create a **`Composable`** function and call it **`ToggleChipWearExample()`**. We will use the **`ToggleChip()`** component:

```
@Composable
fun ToggleChipWearExample() {
    var isChecked by remember { mutableStateOf(true) }
        ToggleChip(
            modifier = Modifier
                .fillMaxWidth()
                .padding(bottom = 8.dp),
            checked = isChecked,
            toggleControl = {
                Switch(
```

```
                checked = isChecked
            )
        },
        onCheckedChange = {
            isChecked = it
        },
        label = {
            Text(
                text = stringResource(
                    id = R.string.alert),
                maxLines = 1,
                overflow = TextOverflow.Ellipsis
            )
        }
    )
}
```

7. Finally, run the code, and you should be able to toggle the chip on and off depending on whether you want to get any notifications or not:

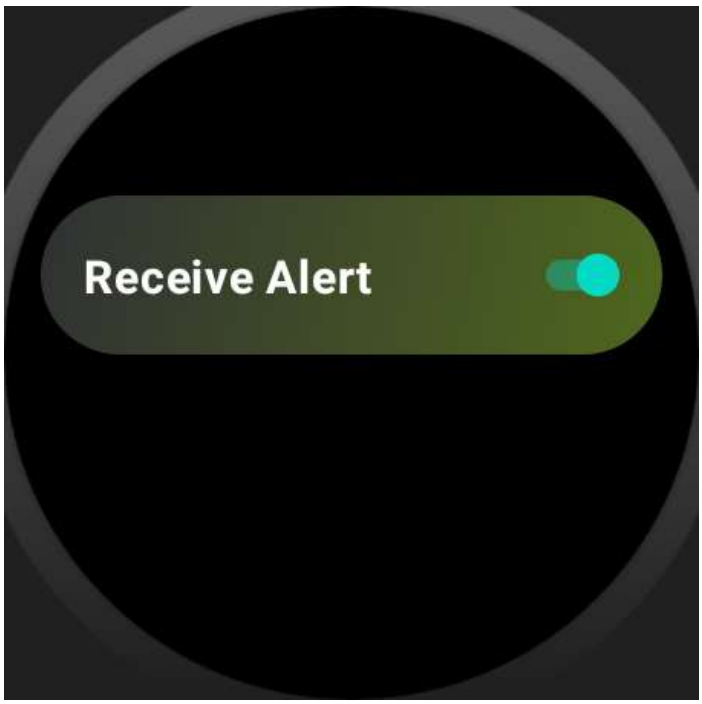

Figure 10.15 – A toggle chip

How it works...

To implement a chip in Wear OS Jetpack Compose, we need to use the already provided **`Chip()`** component. The **`Chip()`** component is stadium shaped and

has a maximum height designed to take no more than two lines of text and can be used to display text, icons, and other information.

You can also use the **`Chip()`** component to display dynamic information. To do this, you can use the **`modifier`** property to specify a function that will be called to update the information displayed on the chip. You can look at the **`Chip()`** component to see what it accepts as its parameters.

The **`ToggleChip()`** composable function takes in several parameters; here are a few significant ones:

- **`checked`**: A Boolean value that represents whether the toggle chip is currently checked
- **`onCheckedChange`**: A lambda function that will be called when the checked state of the toggle chip changes
- **`modifier`**: An optional modifier that can be used to customize the appearance or behavior of the toggle chip
- **`colors`**: An optional **`ToggleChipColors`** object that can be used to customize the colors of the toggle chip

We use **`TextOverflow`** to handle overflowing text since we are dealing with small screens. Check out *Figure 10.15* for more details on what **`ToggleChip`** takes in as parameters:

```
@Composable
public fun ToggleChip(
    checked: Boolean,
    onCheckedChange: (Boolean) -> Unit,
    label: @Composable () -> Unit,
    modifier: Modifier = Modifier,
    toggleControl: @Composable () -> Unit = { ToggleChipDefaults.CheckboxIcon(checked = che
    appIcon: @Composable (() -> Unit)? = null,
    secondaryLabel: @Composable (() -> Unit)? = null,
    colors: ToggleChipColors = ToggleChipDefaults.toggleChipColors(),
    enabled: Boolean = true,
    interactionSource: MutableInteractionSource = remember { MutableInteractionSource() },
    contentPadding: PaddingValues = ToggleChipDefaults.ContentPadding,
    shape: Shape = MaterialTheme.shapes.small,
) {
```

Figure 10.16 – What the ToggleChip composable function accepts as parameters

Implementing ScalingLazyColumn to showcase your content

ScalingLazyColumn extends **LazyColumn**, which is very powerful in Jetpack Compose. You can think of **ScalingLazyColumn** as a component in Wear OS that is used to display a list of items that can be scrolled vertically. The items are scaled and positioned based on their position in the list, and the entire list can be scrolled by dragging the top or bottom of the list.

You can use it, for example, to display a list of components; in our example, we will use it to display all the elements we created in previous recipes. You will also notice we used it in the *Implementing a scrollable list* recipe, where we have a list and displayed the items.

Getting ready

You will need to have completed the previous recipes to continue with this recipe. In addition, in this recipe, instead of commenting on all the elements we created, we will display them as items in **ScalingLazyColumn**.

How to do it...

Follow these steps to build your first **ScalingLazyColumn**:

1. In **MainActivity**, you will notice a comment:

```
/* If you have enough items in your list, use [ScalingLazyColumn] which is an optimized
* version of LazyColumn for wear devices with some added features. For more information,
* see d.android.com/wear/compose.
*/
```

The comment is a callout to developers to utilize **`ScalingLazyColumn`**, which is an optimized version of **`LazyColumn`** for Wear OS.

2. We need to start by creating a **`scalingListState`** value and initialize it to **`rememberScal-ingLazyListState()`**:

```
val scalingListState = rememberScalingLazyListState()
```

The **`rememberScalingLazyListState()`** function simply does as its definition implies, which is to remember the state.

3. We will now need to clean up our Composable function by removing the modifiers we added and using one for all the views. Let's create a **`contentModifier = Modifier`**, and one for our icons:

```
val contentModifier =  Modifier
    .fillMaxWidth()
    .padding(bottom = 8.dp)
val iconModifier =  Modifier
    .size(24.dp)
    .wrapContentSize(align = Alignment.Center)
```

4. We will also need to create a **`Scaffold()`**, which implements the Wear Material Design visual layout structure. **`Scaffold()`** uses **`modifier`**, **`vignette`**, **`positionIndicator`**, **`pageIndicator`**, **`timeText`**, and **`content`**.
5. Let's go ahead and build our screen. In **`Scaffold`**, we will use three parameters: **`vignette`** (which is a full-screen slot for applying a vignette over the content of the scaffold), **`positionIndicator`**, and **`timeText`**. Look at the *How it works...* section to learn more about the parameters:

```
Scaffold(timeText = {} , vignette = {}, positionIndicator = {}) {. . .}
```

6. For **`TimeText`**, we will call **`Modifier.scrollAway`** and pass in **`scalingListState`**:

```
TimeText(modifier = Modifier.scrollAway(scalingListState))
```

7. Since we only have one screen for our project sample, which is scrollable, we will try to show all items simultaneously and all the time. Hence, in **`vignette`**, we will say the position will be **`TopAndBottom`**:

```
Vignette(vignettePosition = VignettePosition.TopAndBottom)
```

8. Finally, on **`positionIndicator`**, we will just pass **`scalingListState`**:

```
PositionIndicator( scalingLazyListState = scalingListState)
```

9. Now, we can finally build our **`ScalingLazyColumn()`**. We will use **`fillMaxSize`** for the modifier, and **`autoCentering`** will be set to index zero; then for **`state`**, pass our already created **`scalingListState`**, and in the items, pass our components:

```
Scaffold(
    timeText = { TimeText(modifier =
        Modifier.scrollAway(scalingListState)) },
    vignette = { Vignette(vignettePosition =
        VignettePosition.TopAndBottom) },
    positionIndicator = {
        PositionIndicator(
            scalingLazyListState = scalingListState
        )
    }
) {
    ScalingLazyColumn(
        modifier = Modifier.fillMaxSize(),
        autoCentering = AutoCenteringParams(
            itemIndex = 0),
        state = scalingListState
    ){
        item { /*TODO*/ }
        item { /*TODO*/ }
        item { /*TODO*/ }
        item { /*TODO*/ }
```

```
        }
    }
```

10. You can get the entire code in the *Technical requirements* section. To clean up some of the code in **`item{}`**, we have the following:

```
item { SampleButton(contentModifier) }
item { SampleButton2(contentModifier, iconModifier) }
item { MessageCard(contentModifier, iconModifier) }
item { ChipWearExample(contentModifier, iconModifier) }
item { ToggleChipWearExample(contentModifier) }
```

11. Finally, when you run the application, you should be able to see all the items displayed and be able to scroll smoothly.

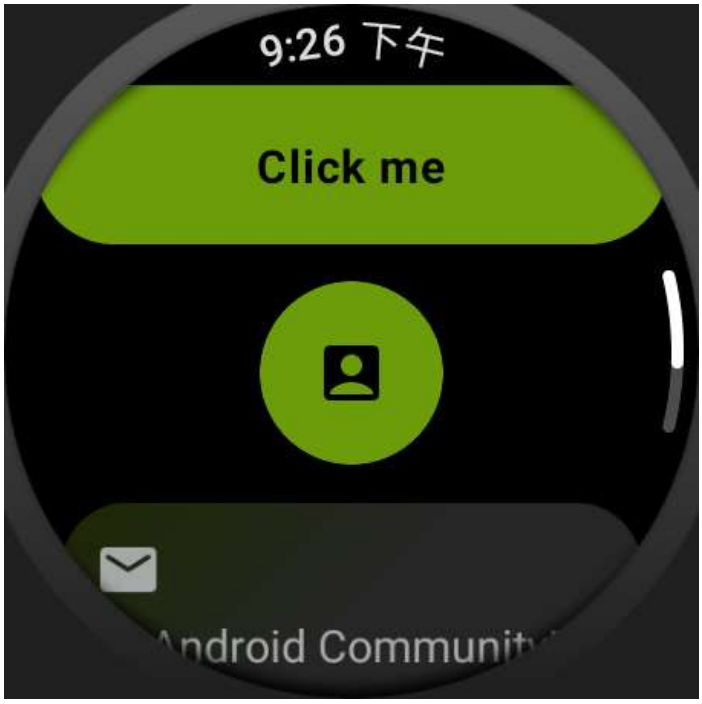

Figure 10.17 – Our composable elements on Wear OS

How it works...

Wear OS Jetpack Compose is a UI toolkit for building Wear OS apps using the Jetpack Compose framework. It is designed to make it easier and more efficient for developers to create wearable apps with a modern and responsive UI. As mentioned before, the **`Composable`** function called

`Scaffold()` has several inputs. In *Figure 10.18*, you will see their meaning and why you might want to use them:

Figure 10.18 – Scaffold function parameters

Some of the significant advantages of Wear OS in Jetpack Compose is that it provides a set of pre-built UI components that are optimized for the unique features of Wear OS devices. And one of the critical benefits is that it simplifies the development process by reducing the amount of boilerplate code that is required to create a UI.

It also provides a consistent and flexible UI design language that can be used across different apps. There is more to learn about Wear OS; also, since this is a new technology, many of the concepts here might change or advance due to API changes in the future, but for now, you can learn more by following this link: **https://developer.android.com/wear**.

IMPORTANT NOTE

There is more to build in Wear OS; for instance, you can build a tile and react when the tile items get clicked to perform an action. To learn more about how you can create your first tile, follow this link: **https://developer.android.com/codelabs/wear-tiles**.