11
Developing for Different Form Factors

This book shows you how to write beautiful, fast, and maintainable Jetpack Compose apps. In *Chapters 1* to *3*, I introduced you to the fundamentals of Jetpack Compose, and explained the core techniques and principles, as well as important interfaces, classes, packages, and, of course, composable functions. *Chapters 4* to *7* focused on building Compose **user interfaces** (**UIs**). You learned how to manage the state and navigate to different screens. We also explored the ViewModel and Repository patterns. *Chapters 8* to *10* covered advanced topics such as animation, interoperability, testing, and debugging.

You hopefully enjoyed digging through all of this and may thus be asking yourself what's left to tackle. The remaining two chapters show you how to bring Compose UIs to devices other than smartphones. We'll start with foldables and tablets. The final chapter helps you leverage your Jetpack Compose skills beyond Android. Wouldn't it be great to see your app on other platforms, such as iOS and desktop? With Compose Multiplatform, you can. *Chapter 12*, *Bringing Your Compose UI to Different Platforms*, will help you take the first steps.

In this chapter, you will learn how to make the most of the available screen real estate using Window Size Classes, Jetpack WindowManager, and Canonical Layouts. The main sections are as follows:

- Understanding different form factors
- Using Jetpack WindowManager
- Organizing the screen content

We'll start by investigating how screen sizes, form factors, and hardware features influence the app layout. I will introduce you to Window Size Classes and explain how they help structure your user interface and how to compute them during runtime.

Relying solely on Window Size Classes is not enough to create awesome layouts for tablets *and* foldables. The second main section, *Using Jetpack WindowManager*, discusses such scenarios. I will show you how to query hardware features such as hinge orientation and device posture and explain how this helps to finetune your user interface.

Finally, the *Organizing the screen content* section explains a Material Design concept called Canonical Layouts. You will learn which Canonical Layouts have been defined so far and in which scenarios they work best.

Technical requirements

Please refer to the *Technical requirements* section in *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 sample apps. This chapter covers the `WindowSizeClassDemo sample`.

The code examples for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Android-UI-development-with-Jetpack-Compose-2nd-edition`.

Understanding different form factors

Android has always been great at supporting different screen sizes, pixel densities, and aspect ratios. These criteria contribute to two terms, **form factor** and **device class**. The latter assigns hardware to broad categories, such as smartphones, tablets, foldables, TVs, and watches. Smartwatches have tiny screens. We need to consider carefully what content should be displayed. Television sets feature huge screens but are watched from a greater distance and are operated with remote controls; we need to make sure content remains readable and easily navigable.

Smartphones usually have smaller screens than foldables, which in turn have displays smaller than or like tablets. All are held in our hands and can be rotated. That's where the form factor becomes important: it describes the size, shape, and *natural orientation* (the way we hold it most of the time) of a device. Smartphones typically are more tall than wide, resembling a portrait. Tablets may favor portrait or landscape (the latter term meaning being wider than tall). Closed foldables usually are held in portrait mode. When opened, they become landscape devices, although they are not rotated. Some devices have a horizontally running hinge. They need to be opened to reveal their main display, which is not particularly large but roughly matches the size of a smartphone. Being foldable does not necessarily mean having a large screen.

Preparing for adaptive layouts

At this point, you may be wondering whether the device class or form factor can help us decide how to lay out the UI. Devices can be rotated, folded, and unfolded at any time. We therefore must not assume they are held in their natural orientation. Here are two examples:

- While we may be using a smartphone in portrait mode most of the time, we will likely rotate it when taking a picture

- Even if a tablet has a portrait form factor, we will likely rotate it while watching a movie because movies are wider than they are tall

Some Android features change the location and size of the area available to your app without *any* orientation or posture change; for example, see the following:

- Picture-in-picture mode
- Multi-window mode
- Resizable windows

These affect the *app window size*.

> **Important**
> Do not make assumptions about the available screen real estate based on device classes and form factors. Instead, rely on the app window size and react to events sent by the system.

In the subsequent sections, I will show you how to do that, but first, I need to elaborate on what happens when you don't. *Figure 11.1* shows the *WindowSizeDemo* sample on a smartphone in portrait mode (the device is held in its natural orientation).

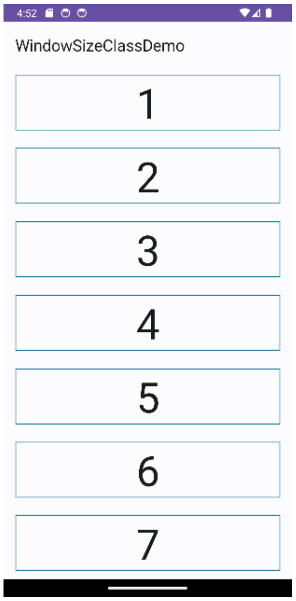


Figure 11.1 – The WindowSizeClassDemo sample in portrait mode

The vertical list is easy to navigate, provides just the right amount of content, and does not waste screen space. Let's have a look at the code:

```
class WindowSizeClassDemoActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MaterialTheme(
        colorScheme = defaultColorScheme()
      ) {
        // In subsequent sections we'll add code here
        WindowSizeClassDemoScreen()
      }
    }
  }
}
```

The `WindowSizeClassDemoActivity` activity class is short and straightforward: inside `setContent {}`, we set up a Material theme and invoke a composable called `WindowSizeClassDemoScreen()`. Have you spotted the comment regarding the code to be added later? This refers to making the UI adaptive. The color scheme is received from a composable function called `defaultColorScheme()`:

```
@Composable
fun defaultColorScheme() = with(isSystemInDarkTheme()) {
  val hasDynamicColor =
        Build.VERSION.SDK_INT >= Build.VERSION_CODES.S
  val context = LocalContext.current
  when (this) {
    true -> if (hasDynamicColor) {
      dynamicDarkColorScheme(context)
    } else {
      darkColorScheme()
    }

    false -> if (hasDynamicColor) {
      dynamicLightColorScheme(context)
    } else {
      lightColorScheme()
    }
  }
}
```

`defaultColorScheme()` makes sure your app honors light mode, dark mode, and (on systems supporting this feature) dynamic colors. Next, let's look at `WindowSizeClassDemoScreen()`. Here is its implementation:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun WindowSizeClassDemoScreen() {
  val scrollBehavior =
          TopAppBarDefaults.pinnedScrollBehavior()
  Scaffold(modifier = Modifier
    .fillMaxSize()
    .nestedScroll(scrollBehavior.nestedScrollConnection),
    topBar = {
      TopAppBar(
        title = {
          Text(text = stringResource(id =
                  R.string.app_name))
        },
        scrollBehavior = scrollBehavior
      )
    }) {
    val list = (1..42).toList()
    SimpleScreen(
      paddingValues = it,
      list = list
    )
  }
}
```

It shows a `Scaffold()` with a `TopAppBar()` and a content composable named `SimpleScreen()`. The top app bar is configured to change its background color when the list is scrolled through. This is achieved by passing `scrollBehavior` to `TopAppBar()` and the `nestedScroll()` modifier to `Scaffold()`. Next, let's look at `SimpleScreen()`:

```
@Composable
fun SimpleScreen(
  paddingValues: PaddingValues,
  list: List<Int>
) {
  NumbersList(
    paddingValues = paddingValues,
    list = list
  )
}
```

`SimpleScreen()` receives a list of `Int` values, which represent the data to be visualized, and `paddingValues`. We will apply the `PaddingValues` instance to the `Scaffold()` content root via `Modifier.padding()` in order to properly offset the top and bottom bars. This happens inside `NumbersList()`:

```
@Composable
fun NumbersList(
  paddingValues: PaddingValues, list: List<Int>
) {
  LazyColumn(
    modifier = Modifier.padding(paddingValues =
                 paddingValues),
    verticalArrangement = Arrangement.spacedBy(8.dp)
  ) {
    items(
      items = list
    ) {
      NumbersItem(it)
    }
  }
}
```

`NumbersList()` uses `LazyColumn()` to show the list of `Int` values as a vertically scrolling list. Each item is rendered using the `NumbersItem()` composable:

```
@Composable
private fun NumbersItem(number: Int) {
  Text(
    modifier = Modifier
      .fillMaxWidth()
      .padding(vertical = 8.dp, horizontal = 16.dp)
      .border(width = 1.dp,
              color =
              MaterialTheme.colorScheme.primary),
    text = number.toString(),
    style = MaterialTheme.typography.displayLarge,
    textAlign = TextAlign.Center
  )
}
```

`NumbersItem()` just shows a large body of text that is centered inside a colored border one density-independent pixel wide. Before we move on, let's briefly recap. We used a vertically scrolling list to show several items, which looks great in portrait mode, but what happens if we rotate the smartphone? Let's find out in the following section.

Enhancing the UI

Figure 11.2 shows the *WindowSizeClassDemo* sample running on a smartphone rotated to landscape mode.

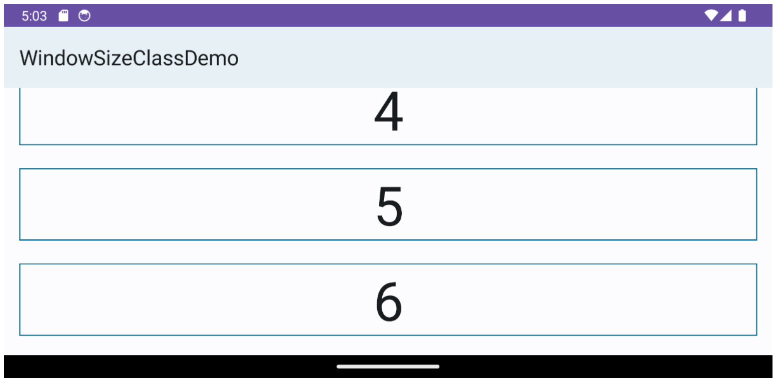


Figure 11.2 – The WindowSizeClassDemo sample in landscape mode

Obviously, the vertical list no longer seems right. That's because there is a lot of empty space inside each list item. Also, the user only sees a few rows. There are several ways to enhance the user experience. One that is easy to implement is switching to a grid. Here's the corresponding composable function:

```
@Composable
fun NumbersGrid(
  paddingValues: PaddingValues,
  list: List<Int>,
  columns: Int = 2
) {
  LazyVerticalGrid(
    columns = GridCells.Fixed(columns),
    modifier = Modifier.padding(paddingValues =
                paddingValues),
    verticalArrangement = Arrangement.spacedBy(8.dp)
  ) {
    items(
      items = list
    ) {
      NumbersItem(number = it)
    }
  }
}
```

Its general structure resembles `NumbersList()`. The content of `LazyVerticalGrid()` is an extension function of `LazyGridScope`, so we can invoke items(), passing the list of `Int` values. `GridCells.Fixed` keeps the size of the columns equal. Like in `NumbersList()`, we set padding using `paddingValues` (which came from `Scaffold()`). Have you noticed the vertical arrangement? `Arrangement.spacedBy()` makes sure that every two adjacent grid elements are spaced by a fixed distance across the main axis.

To use the new composable, we just need to replace the call to `NumbersList()` in `SimpleScreen()` with `NumbersGrid()`. The app will then look as it does in *Figure 11.3* when in landscape mode.

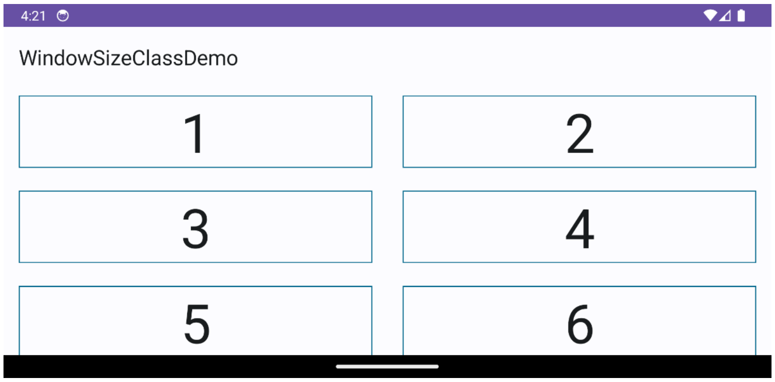


Figure 11.3 – The NumbersGrid() composable in landscape mode

This looks much better than the stretched list items we used before. However, when the app is rotated back to portrait mode, it will still show the grid with two columns. What we need to do instead is display either `NumbersList()` or `NumbersGrid()`, but how do we decide which one? Please recall that we should not consider screen size, posture, or orientation changes but instead rely on the app window size.

In the following section, I will introduce you to Window Size Classes and explain how they help make your app layout adaptive.

Introducing Window Size Classes

Adaptive layouts react to changes in the app window size. Such changes can occur for various reasons, for example, a foldable device being opened (unfolded), or a smartphone being brought into landscape mode.

`View`-based apps can utilize so-called resource qualifiers to signal, for example, that a layout file should be used for displays whose smallest width is at least 600 density-independent pixels (which, by the way, matches a 7-inch tablet). This is done by adding a suffix to the `values` directory name, for example, `res/layout-sw600dp/`. The available qualifiers define a couple of common display sizes. At runtime, the system determines which layout matches the actual hardware closest and uses that one.

Jetpack Compose doesn't rely on layout files. Although any Compose UI hierarchy is hosted by a `ComposeView` instance, that object is not inflated from XML but instantiated inside `setContent {}`. So, we cannot easily use resource qualifiers. Well, we could by providing layout files that contained `<ComposeView .../>` and invoking `setContentView()`; but this is not the endorsed procedure and it fortunately is not necessary.

Like layout-related resource qualifiers, **window size classes** are a set of viewport breakpoints. They categorize the display area available to your app as *compact*, *medium*, or *expanded*. Width and height are classified separately. *Figure 11.4* shows an app running on the Microsoft Surface Duo foldable when folded.

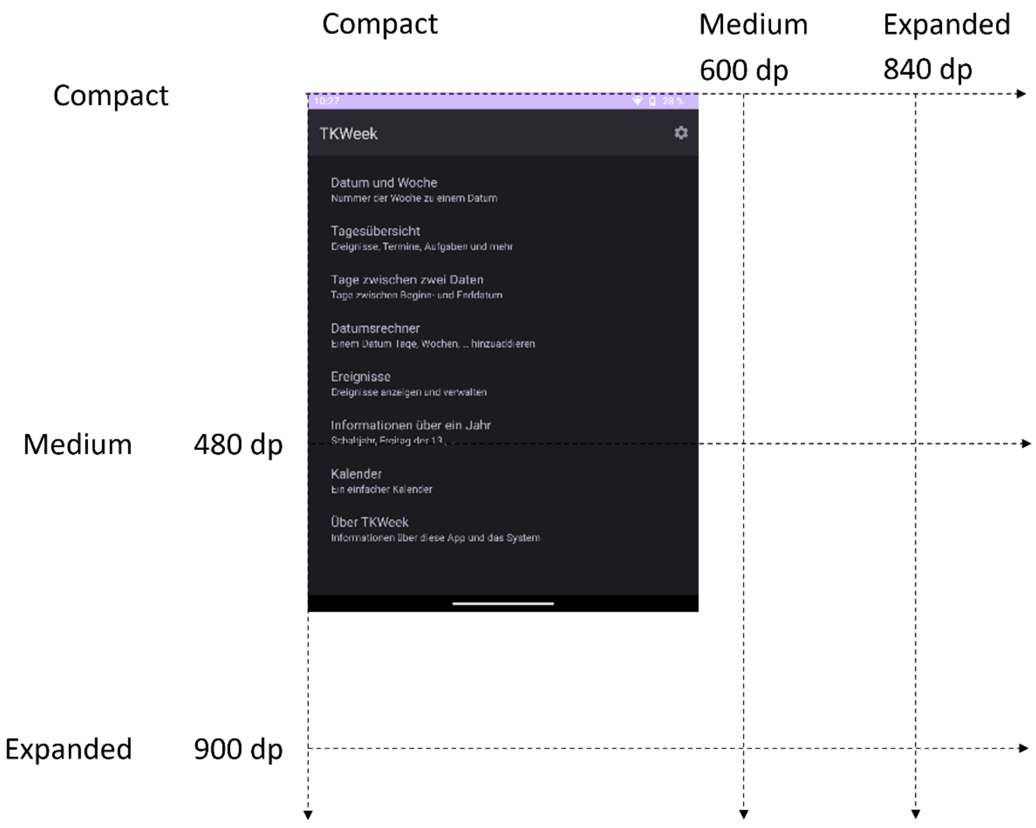

Compact
Medium
600 dp
Expanded
840 dp
Compact
Medium
480 dp
Expanded
900 dp
10:27
28 %
TKWeek
Datum und Woche
Nummer der Woche zu einem Datum
Tagesübersicht
Ereignisse, Termine, Aufgaben und mehr
Tage zwischen zwei Daten
Tage zwischen Beginn- und Enddatum
Datumsrechner
Einem Datum Tage, Wochen, ... hinzuaddieren
Ereignisse
Ereignisse anzeigen und verwalten
Informationen über ein Jahr
Schaltjahr, Freitag der 13., ...
Kalender
Ein einfacher Kalender
Über TKWeek
Informationen über diese App und das System

Figure 11.4 – An app running on a foldable device when folded

Your app has two window size classes – horizontal and vertical. They are independent of each other. So, while the horizontal window size class may be compact, while the vertical one could be medium. Please note that the two values will be swapped when the device is rotated by 90 degrees. Next, let's look at what happens when a foldable device is opened.

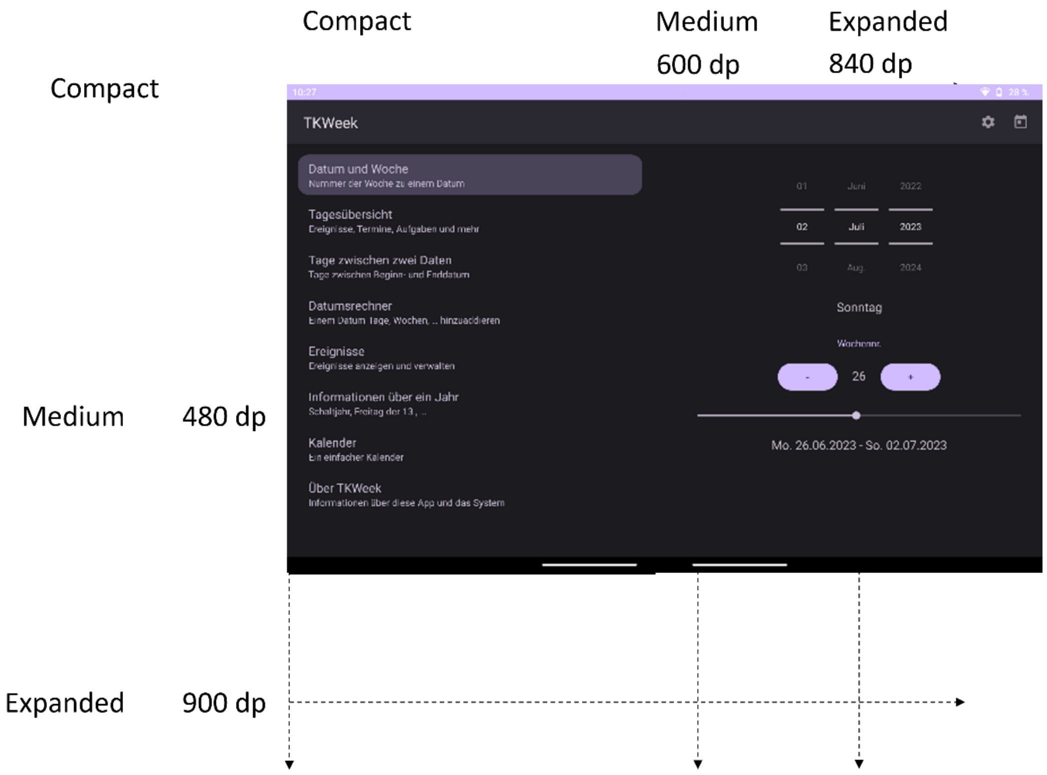

Compact
Medium
600 dp
Expanded
840 dp
Compact
Medium
480 dp
Expanded
900 dp
10:27
28 %
TKWeek
Datum und Woche
Nummer der Woche zu einem Datum
Tagesübersicht
Ereignisse, Termine, Aufgaben und mehr
Tage zwischen zwei Daten
Tage zwischen Beginn- und Enddatum
Datumsrechner
Einem Datum Tage, Wochen, ... hinzuaddieren
Ereignisse
Ereignisse anzeigen und verwalten
Informationen über ein Jahr
Schaltjahr, Freitag der 13., ...
Kalender
Ein einfacher Kalender
Über TKWeek
Informationen über diese App und das System
01 Juni 2022
02 Juli 2023
03 Aug 2024
Sonntag
Wochennr.
-
26
+
Mo. 26.06.2023 - So. 02.07.2023

Figure 11.5 – An app running on a foldable device when unfolded

While the vertical window size class remains the same, the horizontal one becomes expanded. Please recall that besides foldable-specific posture changes, a couple of other events can lead to app window size changes, and therefore changes to the horizontal or vertical window size class. You can read more about this in the *Preparing for adaptive layouts* section.

It is often argued that the available app window width is more important than the available height due to the possibility of vertical scrolling. I encourage you to take the vertical window size class into account too, when crafting your app layout, because in some scenarios, switching to a two-row layout will provide a better user experience. I will give you an example in the *Using Jetpack WindowManager* section.

Before we move on, let's quickly recap: window size classes put the horizontal and vertical app window size in one of three buckets – compact, medium, and expanded. The horizontal and vertical window sizes can change independently; therefore, both the horizontal and vertical window size classes can, too.

Let's return to the *WindowSizeClassDemo* sample. Our task is to show either `NumbersList()` or `NumbersGrid()`. We will make this decision based on the horizontal window size class. Here's a composable function that achieves this:

```
@Composable
fun AdaptiveScreen(
  paddingValues: PaddingValues,
  list: List<Int>
) {
  with(LocalWindowSizeClass.current) {
    if (widthSizeClass == WindowWidthSizeClass.Compact) {
      NumbersList(
        paddingValues = paddingValues,
        list = list
      )
    } else {
      NumbersGrid(
        paddingValues = paddingValues,
        list = list,
        columns = when (widthSizeClass) {
          WindowWidthSizeClass.Medium -> 2
          else -> 3
        }
      )
    }
  }
}
```

`AdaptiveScreen()` uses `LocalWindowSizeClass.current` to obtain a `WindowSizeClass` instance. This class belongs to the `material3-window-size-class` library. To use it, we need to add an implementation dependency:

```
implementation "androidx.compose.material3:material3-window-size-
class:1.1.1"
```

`WindowSizeClass` has two properties, `widthSizeClass` and `heightSizeClass`. The decision on whether to show a list or a grid is determined by this condition: `widthSizeClass == WindowWidthSizeClass.Compact`. The horizontal window size class is also used to determine the number of grid columns. If it is medium, we'll use two columns (`WindowWidthSizeClass.Medium -> 2`); otherwise, we use three.

`LocalWindowSizeClass` is a so-called **CompositionLocal**. You have seen in many examples throughout this book that data usually flows down the UI tree to each composable function that requires it using function arguments. This makes the dependencies of composables explicit. However, for widely and frequently used data, it may be unnerving having to pass it to every function. `CompositionLocal` allow data to flow through the UI tree implicitly – the data is *just there*. My example provides a `WindowSizeClass` instance using this technique. It is available through `LocalWindowSizeClass.current`:

```
val LocalWindowSizeClass = compositionLocalOf
  { WindowSizeClass.calculateFromSize(DpSize.Zero) }
```

We first define a `CompositionLocal` key; `compositionLocalOf {}` receives a factory that supplies a default value if no value is *provided* by `CompositionLocalProvider`. You will see shortly what this means. The `WindowSizeClass` companion object function, `calculateFromSize()`, returns a `WindowSizeClass` instance based on a given size in density-independent pixels, here `DpSize.Zero`. Next, let's look at the provider:

```
setContent {
  MaterialTheme(
    colorScheme = defaultColorScheme()
  ) {
    CompositionLocalProvider(
      LocalWindowSizeClass provides
          calculateWindowSizeClass(activity = this)
    ) {
      WindowSizeClassDemoScreen()
    }
  }
}
```

`CompositionLocalProvider()` is a composable function; it therefore must be invoked inside a Compose UI hierarchy. The `provides` infix function belongs to the `ProvidableCompositionLocal` abstract class, which is the return type of `compositionLocalOf()`. I am using `calculateWindowSizeClass()` to create a `WindowSizeClass` instance. The function receives a reference to the activity whose window size class should be calculated.

Please note

CompositionLocals are a great way to implicitly pass data down the Compose UI tree. If you need data only in a few places, it is best to remain explicit and pass it as parameters. You can read more about using this mechanism at `https://developer.android.com/jetpack/compose/compositionlocal`.

In this section, I showed you how to use window size classes to create an adaptive app layout. We achieved this without looking at the device class or form factor. Instead, we relied solely on the app window size. Unfortunately, more complex app layouts may require additional information. In the following section, I will introduce you to a such scenario and explain how to deal with it, using only information provided by the system.

Using Jetpack WindowManager

In the *Understanding different form factors* section, I introduced you to the *WindowSizeClassDemo* sample. The app evolved from always showing one layout (a vertically scrolling list) to utilizing an adaptive layout based on window size classes: depending on the width of the app window, either a list or a two- or three-column grid will be shown. This works great on smartphones and tablets. But how about foldable devices? *Figure 11.6* shows the sample on an unfolded Microsoft Surface Duo.

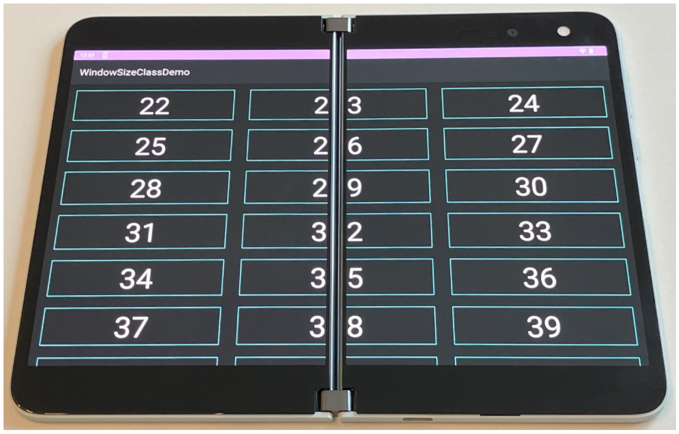


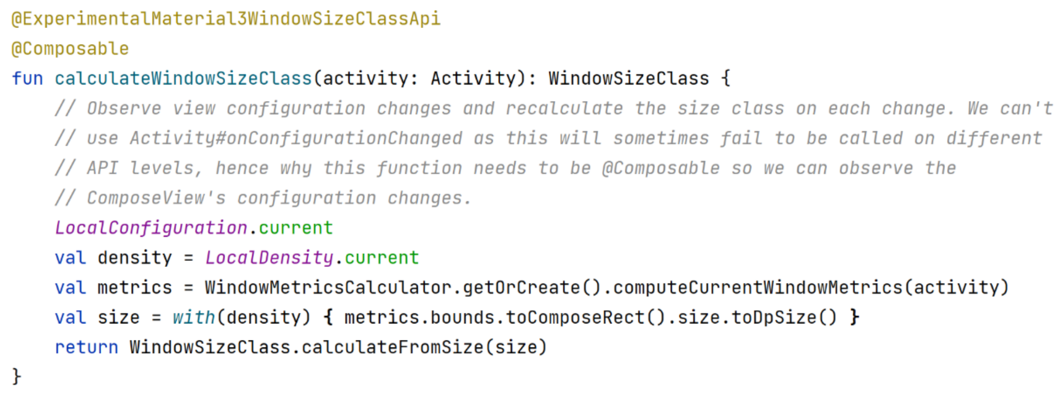
Figure 11.6 – WindowSizeClassDemo running on an unfolded Surface Duo

Foldable devices feature a hinge or fold, which allows the user to switch between two display area sizes. Often, this means it is either *smartphone-sized* or *tablet-sized*. However, there are also products (so-called flip phones) that need to be unfolded to be fully operable. Their screen size resembles smartphones. Consequently, the presence of a hinge does not warrant a certain display area size. Therefore, please avoid building layouts based on conditions such as *is foldable*, *has a hinge*, or *is a tablet*.

Depending on the hardware technology being used, a hinge may obstruct content. In such cases, you should not show important information in the area intersecting with the hinge. How to achieve this depends on your general app layout. You will learn more about global app layout in the *Organizing the screen content* section. Before that, let's look at how to collect information about a hinge or fold using *Jetpack WindowManager*. This library has three main use cases:

- Showing two activities side by side
- Getting window metrics and window size classes
- Getting information related to foldable devices

The first use case, showing two activities side by side, is important to make existing multi-activity apps work great on large screens. This is beyond the scope of this book. What about the second, getting window metrics and window size classes? Didn't we tackle this in the *Introducing Window Size Classes* section? Jetpack WindowManager provides `WindowMetricsCalculator` for obtaining **window metrics**. This is important for both `View`-based and Compose apps because window metrics are used to calculate window size classes. Consequently, `WindowMetricsCalculator` is used inside `calculateWindowSizeClass()` (see *Figure 11.7*).



```
@ExperimentalMaterial3WindowSizeClassApi
@Composable
fun calculateWindowSizeClass(activity: Activity): WindowSizeClass {
    // Observe view configuration changes and recalculate the size class on each change. We can't
    // use Activity#onConfigurationChanged as this will sometimes fail to be called on different
    // API levels, hence why this function needs to be @Composable so we can observe the
    // ComposeView's configuration changes.
    LocalConfiguration.current
    val density = LocalDensity.current
    val metrics = WindowMetricsCalculator.getOrCreate().computeCurrentWindowMetrics(activity)
    val size = with(density) { metrics.bounds.toComposeRect().size.toDpSize() }
    return WindowSizeClass.calculateFromSize(size)
}
```

Figure 11.7 – Source code of calculateWindowSizeClass()

Next, let's focus on the third use case, getting information related to foldable devices. The defining feature of this device class is the **hinge** or **fold**. Which word is used depends on the content being obstructed or separated or not. If it is, we usually say *hinge*. Speaking of a *fold* indicates that its impact on the screen content is less noticeable. The Microsoft Surface Duo has two equal-sized screens, which are connected by a hinge. It can be unfolded up to 360 degrees. The Google Pixel Fold, however, has a fold. Such devices usually can be unfolded up to 180 degrees.

Hinges and folds have a couple of features; among them are the following:

- Orientation
- Location and size
- **Influence on the content**: Obstructing/separating or not

Before I show you how to query them, please recall *Figure 11.6*. If the Surface Duo is unfolded and held in its natural orientation, its hinge runs vertically, and the horizontal window size class is *expanded*; therefore, *WindowSizeClassDemo* shows three columns. The middle one inevitably intersects with the hinge because all the columns are the same width. If the areas to the left and the right of the hinge are equally sized, too, we can lower the impact of the hinge just by making sure that we show an even number of columns.

Here's how to do that. First, we add a dependency to Jetpack WindowManager.

```
dependencies {
  ...
  implementation 'androidx.window:window:1.1.0'
}
```

Second, we make sure that our app receives foldable-related events. This is done like this:

```
class WindowSizeClassDemoActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    lifecycleScope.launch {
      lifecycle.repeatOnLifecycle(
        Lifecycle.State.STARTED
      ) {
        val activity = this@WindowSizeClassDemoActivity
        setContent {
          MaterialTheme(
            colorScheme = defaultColorScheme()
          ) {
            val windowLayoutInfo by
              WindowInfoTracker.getOrCreate(
              context = activity
            )
            .windowLayoutInfo(activity = activity)
            .collectAsState(initial = null)
            CompositionLocalProvider(
              LocalWindowSizeClass provides
              calculateWindowSizeClass(
                activity = activity
              )) {
              WindowSizeClassDemoScreen(
                windowLayoutInfo = windowLayoutInfo
              )
            }
          }
        }
      }
    }
  }
}
```

We need to wrap `setContent {}` inside a lifecycle-aware coroutine using `launch {}` and `repeatOnLifecycle {}` because foldable-related events are received through an instance of `WindowLayoutInfo`. Getting it requires us to do the following:

1. Getting an instance of `WindowInfoTracker` using `getOrCreate()`.
2. Invoking `windowLayoutInfo()` on it; this returns a flow.
3. Collecting the flow as a state using `collectAsState()`.

The third step is to pass the `WindowLayoutInfo` instance down the Compose UI hierarchy by adding a parameter to `WindowSizeClassDemoScreen()` and `AdaptiveScreen()`. I suggest using `WindowLayoutInfo?` and setting its default value to `null` to indicate scenarios in which no foldable-related information is present, such as on ordinary smartphones and tablets:

```
@Composable
fun AdaptiveScreen(
  …,
  windowLayoutInfo: WindowLayoutInfo? = null
) {
  var numColumnsWhenExpanded = 3
  windowLayoutInfo?.displayFeatures?.forEach {
    displayFeature ->
      (displayFeature as FoldingFeature).run {
      if (orientation ==
            FoldingFeature.Orientation.VERTICAL) {
        numColumnsWhenExpanded = 2
      }
    }
  }

  with(LocalWindowSizeClass.current) {
    if (widthSizeClass == WindowWidthSizeClass.Compact) {
      …
    } else {
      NumbersGrid(
        …,
        columns = when (widthSizeClass) {
          WindowWidthSizeClass.Medium -> 2
          else -> numColumnsWhenExpanded
        }
      )
    }
  }
}
```

The enhanced version of AdaptiveScreen() stores the number of columns to be displayed when the horizontal window size class is neither Compact nor Medium, in a variable named numColumnsWhenExpanded. Its initial value is 3. If there is a hinge or fold that runs vertically, numColumnsWhenExpanded becomes 2. Granted, this is a simplistic approach. I will expand on this soon, but first, let me show you how to read hinge and fold features.

Reading hinge and fold features

The WindowLayoutInfo class has a property called displayFeatures. It's a list containing elements of type DisplayFeature. DisplayFeature is an interface with a property named bounds. As the name suggests, display features describe a physical feature on a display, such as a fold or hinge. The bounds property contains the bounding rectangle of the feature within the application window, in the window coordinate space. But how is the fold- or hinge-specific information provided?

The FoldingFeature interface extends DisplayFeature. It provides a couple of properties, for example, orientation (Orientation.VERTICAL or Orientation.HORIZONTAL), isSeparating (the window is split into multiple physical areas), occlusionType (OcclusionType.FULL or OcclusionType.NONE), and state (State.FLAT or State.HALF_OPENED). Please consult the API documentation of these properties for further information, as digging deeper into them unfortunately is beyond the scope of this book.

Which ones you will be querying depends on the purpose of your app. Consider a media player: if the device reports State.HALF_OPENED and the hinge is running horizontally, this posture could indicate that one half of the device is lying flat on a desk while the other one is in an upright position. Consequently, your app could show two rows, one containing the media and the other one media controls and additional information. You will learn more about global app layout in the *Organizing the screen content* section.

Before we move on, let's return to something I wrote a little earlier: I considered my example with the numColumnsWhenExpanded variable simplistic. Why is that? Each display feature provides its location and size on the screen through the bounds property. However, I didn't use it but instead assumed that both display area halves have the same size. To check, we would need the total display size – which you can obtain using WindowMetricsCalculator, another Jetpack WindowManager interface – and do some arithmetic. How to calculate, depends on the orientation of the hinge.

Another simplification I made was to ignore the size of the fold or hinge. While having an even number of columns makes sure that most of the grid item content is visible, the rightmost area of items to the left of the fold or hinge and the leftmost area of items to the right of the fold or hinge may still be invisible if the hinge is obstructing them. Fixing this requires support from the composable rendering the grid.

To finish this section, let's recap what we have learned so far. If a device exposes display features, you can query them by iterating over the displayFeatures list provided by windowLayoutInfo(). Currently, there is only FoldingFeature, but future versions of Jetpack WindowManager might provide additional ones. Depending on the horizontal window size class, the *WindowSizeClassDemo* sample uses either a vertically scrolling list or a grid with two or three columns for its global app layout. The final main section of this chapter, *Organizing the screen content*, will dig deeper into this topic.

Organizing the screen content

In the previous sections, I explained that to make your app look great on a wide range of devices, you should build its layout on top of Window Size Classes and foldable-related events emitted by Jetpack WindowManager. But what does *layout* refer to? *Figure 11.8* shows the *ComposeUnitConverter* sample from *Chapters 6* and *7*.

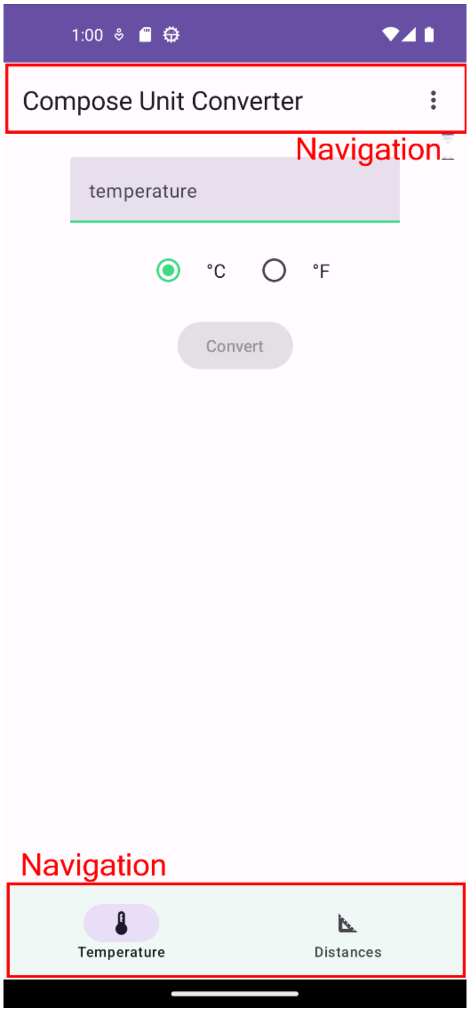


Figure 11.8 – The ComposeUnitConverter sample

There appear to be three areas: the content, bottom navigation, and the top app bar. However, **Material You** (the design language and design system used on Android) puts the latter two in one bucket, *navigation*. Therefore, inside the app window, there are only two major blocks or areas: the content (sometimes referred to as body) and the navigation. How these blocks are laid out is defined in the Material You documentation (`https://m3.material.io/foundations/layout/understanding-layout/overview`). For example, bottom navigation should be used only if the horizontal Window Size Class is `Compact`. Otherwise, display either a navigation rail or a navigation drawer.

Please note

Window Size Classes should be available in most of your composables. Please recall that you can provide them directly by passing them as a parameter or provide them using `CompositionLocal`.

Next, let's focus on the content (or body). Its layout matters the most. After all, the content is the reason why your users open the app. Which layout is best depends on the purpose of the app and the data or information to be presented, but also on the size of the app window.

Google has analyzed lots of apps and has identified three broad app layouts. They are called **Canonical Layouts**. The first one is called **Feed** (see *Figure 11.9*). You already saw it in the *WindowSizeClassDemo* sample.

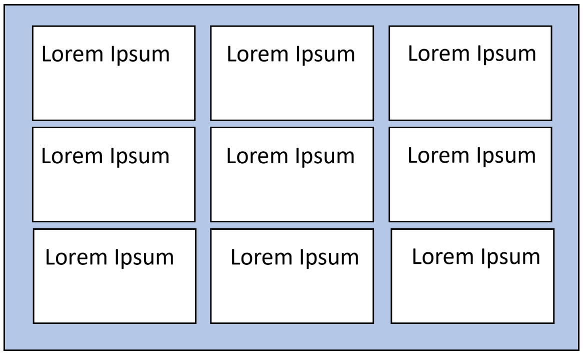


Figure 11.9 – Feed

A feed organizes cards or lists in a grid. It allows users to browse large amounts of content quickly and easily. Example use cases include pictures, business cards, news snippets, and social media entries.

The second Canonical Layout is called **List-detail** (see *Figure 11.10*).

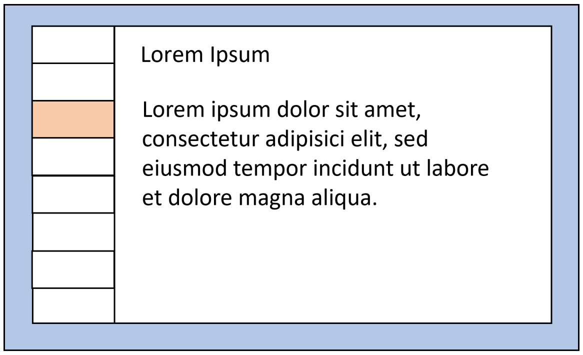

Lorem Ipsum
Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua.

Figure 11.10 – List-detail

List-detail consists of two parts, a scrollable list of items and an item detail containing supplementary information. Example use cases include e-mails, contacts lists, and to-do lists.

The third Canonical Layout is called **Supporting pane** (see *Figure 11.11*).

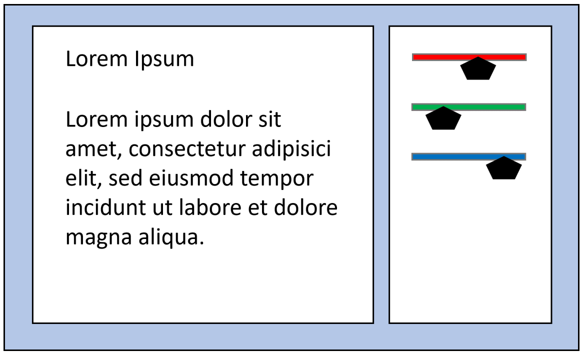

Lorem Ipsum
Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua.

Figure 11.11 – Supporting pane

It consists of a larger primary section for the main app content and a secondary section that supports the main app content. Example use cases include imaging and drawing apps, where the main section contains the picture or graphics, whereas the secondary section offers tools, sliders, or knobs to alter or modify the content.

At this point, you may be asking yourself what's special about Canonical Layouts. Aren't they just **human interaction patterns**? In a way, they are. However, they support Window Size Classes. Here's how: all three are based upon *logical panes*. Feed comprises one such pane, whereas List-detail and Supporting pane consist of two. These panes are called *logical* because they may or may not have representations while the app is running. If the app window size is big enough, both the list and detail can be shown.

If not, either the list or detail will be visible. The same applies to Supporting pane. While Feed has only one pane, it can consist of one, two, or more columns – depending on the Window Size Class.

Unfortunately, at the time of writing, there are no ready-to-use composable functions that provide Canonical Layouts. You need to implement them on your own or use additional libraries. Accompanist by Google contains the `TwoPanel()` composable. Please refer to `https://google.github.io/accompanist/adaptive/` for further information. My own library, `compose_adaptive_scaffold`, is also based on the idea of two panes. Like Accompanist, it is open source and available on GitHub. Please refer to `https://github.com/tkuenneth/compose_adaptive_scaffold` for further information.

Summary

In this chapter, you learned how to use Window Size Classes, Jetpack WindowManager, and Canonical Layouts to make sure your app looks great on smartphones, tablets, and foldable devices. First, we investigated how screen sizes, form factors, and hardware features influence app layout. Then, I explained how Window Size Classes help structure your UI, and how you can compute them during runtime.

The second main section, *Using Jetpack WindowManager*, explained why relying solely on Window Size Classes is not enough to create awesome layouts for tablets *and* foldables. You learned how to query hardware features such as hinge orientation and device posture and how this helps fine-tune your UI.

Finally, the *Organizing the screen content* section introduced a Material Design concept called Canonical Layouts. You learned which Canonical Layouts have been defined so far and in which scenarios they work best.

In the final chapter, *Bringing Your Compose UI to Different Platforms*, I will show you how to bring your Jetpack Compose knowledge to systems other than Android, for example, desktop.

Exercise

The *WindowSizeClass* sample makes a few assumptions about hinges and folds. Please modify the app so that it uses `WindowMetricsCalculator` to obtain the display size and then checks whether the areas to the left and the right of the hinge are the same size.