Chapter 4: Laying Out UI Elements

In the previous chapters, you learned how to build simple UIs. Although they consisted of just a few UI elements, they needed to arrange their buttons, text fields, and sliders in a particular order, direction, or hierarchy. **Layouts** position and size their content in a way specific to this layout, such as horizontally (`Row()`) or vertically (`Column()`). This chapter explores layouts in greater detail.

In this chapter, we will cover the following topics:

- Using predefined layouts
- Understanding the single measure pass
- Creating custom layouts

We will start by exploring the predefined layouts of `Row()`, `Column()`, and `Box()`. You will learn how to combine them to create beautiful UIs. Next, I'll introduce you to `ConstraintLayout`. It places composables that are relative to others on the screen and uses attributes to flatten the UI element hierarchy. This is an alternative to nesting `Row()`, `Column()`, and `Box()`.

The second main section will explain why the layout system in Jetpack Compose is more performant than the traditional View-based approach. We will once again go under the covers and look at some of the internals of the Compose runtime. This will prepare you for the final main section of this chapter, *Creating custom layouts*.

In this final section, you will learn how to create a custom layout and thus gain precise control over the rendering of its children. This is helpful if the predefined layouts do not offer enough flexibility for a particular use case.

Now, let's get started!

Technical requirements

This chapter showcases three sample apps:

- **`PredefinedLayoutsDemo`**
- **`ConstraintLayoutDemo`**
- **`CustomLayoutDemo`**

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, and how to get it. If you want to try the **`CheckboxWithLabel()`** composable from the *Combining basic building blocks* section, you can use the *Sandbox* app project in the top-level directory of this book's GitHub repository at https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose. Open its **`SandboxActivity`** and copy the composable functions from **`code_snippets.txt`**, which is located in the **`/chapter_04`** folder.

Using predefined layouts

When you create a UI, you must define where its elements appear and how big they are. Jetpack Compose provides a couple of basic layouts, which arrange their content along one main axis. There are three axes to consider:

- Horizontal
- Vertical
- Stacked

Each axis is represented by a layout. **`Row()`** arranges its content horizontally, while **`Column()`** does so vertically. **`Box()`** and **`BoxWithConstraints()`** stack their contents on top of each other. By combining these axis-orientated building blocks, you can create great-looking UIs easily.

Combining basic building blocks

The following **`PredefinedLayoutsDemo`** sample app shows three checkboxes that toggle a red, a green, and a blue rectangle, respectively. The boxes appear only if the corresponding checkbox is checked:

Figure 4.1 – The sample PredefinedLayoutsDemo app

Let's see how this is done. First, I will show you how to create a checkbox with an accompanying label:

```
@Composable
fun CheckboxWithLabel(label: String, state:
MutableState<Boolean>) {
  Row(
    modifier = Modifier.clickable {
      state.value = !state.value
    }, verticalAlignment = Alignment.CenterVertically
  ) {
    Checkbox(
      checked = state.value,
      onCheckedChange = {
        state.value = it
      }
    )
```

```
      Text(
        text = label,
        modifier = Modifier.padding(start = 8.dp)
      )
    }
  }
```

Jetpack Compose has a built-in **`Checkbox()`**. It receives the current state (**`checked`**) and a lambda expression (**`onCheckedChange`**), which is invoked when the checkbox is clicked. At the time of writing, you cannot pass a label. However, we can achieve something similar by putting **`Checkbox()`** and **`Text()`** inside a **`Row()`**. We need to make the row clickable because we want to change the state of the checkbox when the text is clicked too. To make the checkbox with a label more visually appealing, we can center **`Checkbox()`** and **`Text()`** vertically inside the row by setting **`verticalAlignment`** to **`Alignment.CenterVertically`**.

`CheckboxWithLabel()` receives a **`MutableState<Boolean>`** because other composables need to be recomposed when it changes the value inside **`onCheckedChange`**.

Next, let's see where the state is created:

```
  @Composable
  fun PredefinedLayoutsDemo() {
    val red = remember { mutableStateOf(true) }
    val green = remember { mutableStateOf(true) }
    val blue = remember { mutableStateOf(true) }
    Column(
      modifier = Modifier
        .fillMaxSize()
        .padding(16.dp)
    ) {
      …
```

`PredefinedLayoutsDemo()` arranges its content vertically by putting it inside a **`Column()`**. The column fills all the available space (**`fillMaxSize()`**)

and has a padding of 16 density-independent pixels on all four sides (**`padding(16.dp)`**). The three states (**`red`**, **`green`**, and **`blue`**) are passed to **`CheckboxWithLabel()`**. Here's what these invocations look like:

```
CheckboxWithLabel(
  label = stringResource(id = R.string.red),
  state = red
)
CheckboxWithLabel(
  label = stringResource(id = R.string.green),
  state = green
)
CheckboxWithLabel(
  label = stringResource(id = R.string.blue),
  state = blue
)
```

They are almost the same, differing only in the state (**`red`**, **`green`**, and **`blue`**) and the label string (**`R.string.red`**, **`R.string.green`**, or **`R.string.blue`**).

Now, let's find out how the stacked colored boxes are created:

```
Box(
  modifier = Modifier
    .fillMaxSize()
    .padding(top = 16.dp)
) {
  if (red.value) {
    Box(
      modifier = Modifier
        .fillMaxSize()
        .background(Color.Red)
    )
  }
  if (green.value) {
    Box(
```

```
        modifier = Modifier
          .fillMaxSize()
          .padding(32.dp)
          .background(Color.Green)
      )
    }
    if (blue.value) {
      Box(
        modifier = Modifier
          .fillMaxSize()
          .padding(64.dp)
          .background(Color.Blue)
      )
    }
  }
```

The three colored boxes are put inside another **`Box()`**, which fills all the available space. To create a gap between it and the last checkbox, I specified a top padding of 16 density-independent pixels.

A colored box is only added if its corresponding state is **`true`** (for example, **`if (red.value) { …)`**. All colored boxes fill the available space. As they will be stacked on top of each other, only the last (top) one will be visible. To fix this, the green and blue boxes receive paddings that differ in size: the padding for the blue box (the last one) is 64 density-independent pixels, so in the areas of the padding, the green box becomes visible. The green box has a padding of 32 density-independent pixels, so in this area, the first box (the red one) can be seen.

As you have seen, by combining basic layouts such as **`Box()`** and **`Row()`**, you can easily create great-looking UIs. In the next section, I will introduce you to an alternative approach where we will define a UI based on constraints.

Creating layouts based on constraints

Defining UIs based on constraints has been the most recent preferred approach in Android's traditional **`View`** world because older layouts such as **`RelativeLayout`** or **`LinearLayout`** could impact performance when they're used in large, multiply-nested layouts. **`ConstraintLayout`** avoids this by flattening the **`View`** hierarchy. As you will see in the *Understanding the single measure pass* section, this is no issue for Jetpack Compose.

However, for more complex layouts in a Compose app, you may still want to limit the nesting of **`Box()`**, **`Row()`**, and **`Column()`** to make your code simpler and clearer. This is where **`ConstraintLayout()`** can help.

The **`ConstraintLayoutDemo`** sample app is a reimplementation of **`PredefinedLayoutsDemo`** based on **`ConstraintLayout()`**. By comparing the two versions, you get a thorough understanding of how this composable function works. To use **`ConstraintLayout()`** in your app, you need to add a dependency to your module-level **`build.gradle`** file. Please note that the version number shown here is just an example. You can find the latest version at https://developer.android.com/jetpack/androidx/versions/all-channel:

```
implementation
"androidx.constraintlayout:constraintlayout-
compose:1.0.0-rc02"
```

So, how do we define a layout based on constraints? Let's find out by examining the reimplementation of **`CheckboxWithLabel()`**. It places text next to a checkbox:

```
@Composable
fun CheckboxWithLabel(
  label: String,
  state: MutableState<Boolean>,
  modifier: Modifier = Modifier
) {
  ConstraintLayout(modifier = modifier.clickable {
    state.value = !state.value
```

```
    }) {
      val (checkbox, text) = createRefs()
      Checkbox(
        checked = state.value,
        onCheckedChange = {
          state.value = it
        },
        modifier = Modifier.constrainAs(checkbox) {
        }
      )
      Text(
        text = label,
        modifier = Modifier.constrainAs(text) {
          start.linkTo(checkbox.end, margin = 8.dp)
          top.linkTo(checkbox.top)
          bottom.linkTo(checkbox.bottom)
        }
      )
    }
  }
```

`ConstraintLayout()` uses a **domain-specific language** (**DSL**) to define the location and size of a UI element relative to other ones. Therefore, each composable in a **`ConstraintLayout()`** must have a reference associated with it, which is created using **`createRefs()`**. Constraints are provided using the **`constrainAs()`** modifier. Its lambda expression receives a **`ConstrainScope`**. It includes properties such as **`start`**, **`top`**, and **`bottom`**. These are called **anchors** because they define a location that can be linked (using **`linkTo()`**) to the location of another composable.

Let's look at **`Text()`**. Its **`constrainAs()`** contains **`bottom.linkTo(checkbox.bottom)`**. This means that the bottom of the text is constrained to the bottom of the checkbox. As the top of the text is linked to the top of the checkbox, the height of the text is equal to the height of the checkbox. The following line means that the start of the text is con-

strained by the end of the checkbox, with an additional margin of 8 density-independent pixels:

```
start.linkTo(checkbox.end, margin = 8.dp)
```

So, in the direction of reading, the text comes after the checkbox. Next, let's look at **`ConstraintLayoutDemo()`**:

```
@Composable
fun ConstraintLayoutDemo() {
  val red = remember { mutableStateOf(true) }
  val green = remember { mutableStateOf(true) }
  val blue = remember { mutableStateOf(true) }
  ConstraintLayout(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp)
  ) {
    val (cbRed, cbGreen, cbBlue, boxRed, boxGreen,
boxBlue) =
            createRefs()
    CheckboxWithLabel(
      label = stringResource(id = R.string.red),
      state = red,
      modifier = Modifier.constrainAs(cbRed) {
        top.linkTo(parent.top)
      }
    )
    ...
```

Once we have created the references that areneeded to define constraints using **`createRefs()`**, we add our first **`CheckboxWithLabel()`**. Its **`top`** is linked to (constrained by) the **`top`** of **`parent`**, which is **`ConstraintLayout()`**. So, the first checkbox with a label is the topmost one. Here's how the second one, which toggles the green box, is invoked:

```
CheckboxWithLabel(
```

```
    label = stringResource(id = R.string.green),
    state = green,
    modifier = Modifier.constrainAs(cbGreen) {
      top.linkTo(cbRed.bottom)
    }
  )
```

Its top is constrained by the bottom of the first checkbox with a label (which toggles the red box). Finally, here's how we need to constrain the third **`CheckboxWithLabel()`**:

```
  modifier = Modifier.constrainAs(cbBlue) {
    top.linkTo(cbGreen.bottom)
  }
```

To conclude this section, let me show you how to define the colored boxes. Here's the red one:

```
  if (red.value) {
    Box(
      modifier = Modifier
        .background(Color.Red)
        .constrainAs(boxRed) {
          start.linkTo(parent.start)
          end.linkTo(parent.end)
          top.linkTo(cbBlue.bottom, margin = 16.dp)
          bottom.linkTo(parent.bottom)
          width = Dimension.fillToConstraints
          height = Dimension.fillToConstraints
        }
    )
  }
```

Both **`start`** and **`end`** are linked to the corresponding anchors of **`parent`** (which is **`ConstraintLayout()`**). **`top`** is constrained by **`bottom`** of the last checkbox, so the red box appears below it. **`bottom`** of the red box is constrained by **`bottom`** of **`parent`**. Please note that currently, we must set **`width`** and **`height`** to the value that we obtained from

`Dimension.fillToConstraints`. Otherwise, the box won't have the correct size.

Next, let's look at the constraints of the green box:

```
constrainAs(boxGreen) {
  start.linkTo(parent.start, margin = 32.dp)
  end.linkTo(parent.end, margin = 32.dp)
  top.linkTo(cbBlue.bottom, margin = (16 + 32).dp)
  bottom.linkTo(parent.bottom, margin = 32.dp)
  width = Dimension.fillToConstraints
  height = Dimension.fillToConstraints
}
```

This code is practically the same. One difference is that all the sides receive a **`margin`** of 32 density-independent pixels. This is necessary because we want the red box, which is below the green one, to be visible at the locations of the margin. As the red box already has a **`top`** margin of 16, we must add this value to the **`top`** margin. You may be wondering why I am not linking to **`boxRed`** instead. That is because the red box will not be present if its corresponding checkbox is not checked. In this case, the anchor would not be there.

Here's what the constraints for the blue box will look like:

```
constrainAs(boxBlue) {
  start.linkTo(parent.start, margin = 64.dp)
  end.linkTo(parent.end, margin = 64.dp)
  top.linkTo(cbBlue.bottom, margin = (16 + 64).dp)
  bottom.linkTo(parent.bottom, margin = 64.dp)
  width = Dimension.fillToConstraints
  height = Dimension.fillToConstraints
}
```

The only thing I needed to change is the margin on all four sides because otherwise, the box below (the green one) would not be visible.

In a nutshell, this is how **`ConstrainLayout()`** works:

- You constrain a composable by linking its anchors to other ones
- The linking is based on references. To setup these references, you must call **`createRefs()`**.

The main advantage of combining **`Box()`**, **`Row()`**, and **`Column()`** is that you flatten your UI element hierarchy. Think of it like this: in **`PredefinedLayoutsDemo`**, I needed to stack the colored boxes in a parent **`Box()`**. In **`ConstrainLayoutDemo`**, the boxes and the three **`CheckboxWithLabel()`** share the same parent (a **`ConstrainLayout()`**). This reduces the number of composables and makes the code cleaner.

In the next section, we will once again peek inside the internals of Jetpack Compose. We will learn how the layout process works and why it is more efficient than the traditional View-based approach.

Understanding the single measure pass

Laying out a UI element hierarchy means determining the sizes of all the elements and positioning them on the screen based on the layout strategy of their parent. At first, getting the size of, say, some text doesn't sound too complicated. After all, isn't it determined by the font and the text to be output? Here's an example, with two pieces of text laid out in a **`Column()`**:

```
@Composable
@Preview
fun ColumnWithTexts() {
  Column {
    Text(
      text = "Android UI development with Jetpack
Compose",
      style = MaterialTheme.typography.h3
```

```
      )
      Text(
        text = "Hello Compose",
        style = MaterialTheme
            .typography.h5.merge(TextStyle(color =
  Color.Red))
      )
    }
  }
```

If you deploy the preview, you will notice that, in portrait mode, the first text requires more space vertically than in landscape mode. The second text always fits into one line. The size that a composable takes on-screen partially depends on the conditions that have been imposed from *outside*. Here, the maximum width of the column (the parent) influences the height of the first piece of text. Such conditions are called **constraints**. You will see them in action in the *Creating custom layouts* section. Please note that they are not the same as the constraints you use in **`ConstraintLayout()`**.

Once a layout has obtained and measured the size of its content, the layout will position its children (the content). Let's see how this works by looking at the source code of **`Column()`**:

```
65  @Composable
66  inline fun Column(
67      modifier: Modifier = Modifier,
68      verticalArrangement: Arrangement.Vertical = Arrangement.Top,
69      horizontalAlignment: Alignment.Horizontal = Alignment.Start,
70      content: @Composable ColumnScope.() → Unit
71  ) {
72      val measurePolicy = columnMeasurePolicy(verticalArrangement, horizontalAlignment)
73      Layout(
74          content = { ColumnScopeInstance.content() },
75          measurePolicy = measurePolicy,
76          modifier = modifier
77      )
78  }
```

Figure 4.2 – Source code of Column()

The composable is very short. Besides assigning a value to **`measurePolicy`**, it only invokes **`Layout()`**, passing **`content`**, **`measurePolicy`**, and **`modifier`**. We

briefly looked at the source code of **`Layout()`** in the *Emitting UI elements* section of *Chapter 3*, *Exploring the Key Principles of Compose*, to understand what it means to emit UI elements. Now, we'll focus on the layout process. The **`measurePolicy`** variable references an implementation of the **`MeasurePolicy`** interface. In this case, it's the result of a call to **`columnMeasurePolicy()`**.

Defining measure policies

Depending on the values of **`verticalArrangement`** and **`horizontalAlignment`**, the call to **`columnMeasurePolicy()`** returns either **`DefaultColumnMeasurePolicy`** (a variable) or the result of **`rowColumnMeasurePolicy()`**. **`DefaultColumnMeasurePolicy`** calls **`rowColumnMeasurePolicy`**. Therefore, this function defines the measure policy for any **`Column()`**. It returns a **`MeasurePolicy`**.

TIP

Please remember that you can look at the source code of a policy by pressing the Ctrl key and clicking on a name, such as **`columnMeasurePolicy`**.

`MeasurePolicy` belongs to the **`androidx.compose.ui.layout`** package. It defines how a layout is measured and laid out, so it is the main building block for both predefined (for example, **`Box()`**, **`Row()`**, and **`Column()`**) and custom layouts. Its most important function is **`measure()`**, which is an extension function of **`MeasureScope`**. This function receives two parameters, **`List<Measurable>`** and **`Constraints`**. The elements of the list represent the children of the layout. They can be measured using **`Measurable.measure()`**. This function returns an instance of **`Placeable`**, a representation of the size a child wants to span.

`MeasureScope.measure()` returns an instance of **`MeasureResult`**. This interface defines the following components:

- The size of a layout (**`width`**, **`height`**)

- Alignment lines (**`alignmentLines`**)
- Logic to position the children (**`placeChildren()`**)

You can find an implementation of **`MeasureResult`** in the *Creating custom layouts* section.

Alignment lines define an offset line that can be used by parent layouts to align and position their children. For example, text baselines are alignment lines.

Depending on the complexity of the UI, a layout may find that its children do not fit nicely in its boundaries. The layout may want to remeasure the children, passing different measurement configurations. Remeasuring children is possible in the Android **`View`** system, but this can lead to decreased performance. Therefore, in Jetpack Compose, a layout may measure its content only once. If it tries again, an exception will be thrown.

A layout can, however, query the **intrinsic size** of its children and use it for sizing and positioning. **`MeasurePolicy`** defines four extension functions of **`IntrinsicMeasureScope`**. **`minIntrinsicWidth()`** and **`maxIntrinsicWidth()`** return the minimum or maximum width of a layout, given a particular height, so that the content of the layout can be painted completely. **`minIntrinsicHeight()`** and **`maxIntrinsicHeight()`** return the minimum or maximum height of a layout given a particular width so that the content of the layout can be painted completely. To get an idea of how they work, let's briefly look at one of them:

```
The function used to calculate IntrinsicMeasurable.minIntrinsicWidth. It represents the minimum width this
layout can take, given a specific height, such that the content of the layout can be painted correctly.
fun IntrinsicMeasureScope.minIntrinsicWidth(
    measurables: List<IntrinsicMeasurable>,
    height: Int
): Int {
    val mapped = measurables.fastMap {
        DefaultIntrinsicMeasurable(it, IntrinsicMinMax.Min, IntrinsicWidthHeight.Width)
    }
    val constraints = Constraints(maxHeight = height)
    val layoutReceiver = IntrinsicsMeasureScope( density: this, layoutDirection)
    val layoutResult = layoutReceiver.measure(mapped, constraints)
    return layoutResult.width
}
```

Figure 4.3 – Source code of minIntrinsicWidth()

IntrinsicMeasureScope.minIntrinsicWidth() receives two parameters: **height** and a list of children (**measurables**). The **IntrinsicMeasurable** interface defines four functions that obtain the minimum or maximum values for a particular element (**minIntrinsicWidth()**, **maxIntrinsicWidth()**, **minIntrinsicHeight()**, and **maxIntrinsicHeight()**). Each element of **measurables** is converted into an instance of **DefaultIntrinsicMeasurable**. As this class implements the **Measurable** interface, it provides an implementation of **measure()**. It returns **FixedSizeIntrinsicsPlaceable**, which provides the smallest possible width for a given **height**. The converted children are measured by an instance of **IntrinsicsMeasureScope**.

We'll finish looking at the internals of the Compose layout process by turning to **Constraints**. They are, for example, passed to **MeasureScope.measure()**. The class belongs to the **androidx.compose.ui.unit** package. It stores four values: **minWidth**, **minHeight**, **maxWidth**, and **maxHeight**. They define the minimum and maximum values the children of a layout must honor when measuring themselves. So, their width must be no smaller than **minWidth** and no larger than **maxWidth**. Their height must lie within **minHeight** and **maxHeight**.

The companion object defines the **Infinity** constant. It is used to signal that the constraint should be considered infinite. To create a **Constraints** instance, you can invoke the top-level **Constraints()** function.

This was a lot of information. Before moving on, let's recap what we have learned.

- The **Layout()** composable receives three parameters: the content, the measure policy, and a modifier.
- The measure policy defines how a layout is measured and laid out.
- The intrinsic size of a layout determines the minimum or maximum dimension for the corresponding input.

In the traditional View system, a parent view may call the `measure()` method more than once on its children (please refer to https://developer.android.com/guide/topics/ui/how-android-draws for details). On the other hand, Jetpack Compose requires that children must be measured *exactly once* before they are positioned. This results in a more performant measurement.

In the next section, we will make use of this knowledge by implementing a simple custom layout. It will position its children from left to right and from top to bottom. When one row is filled, the next one will be started below it.

Creating custom layouts

Sometimes, you may want to lay children out one after another in a row and start a new row when the current one has been filled. The `CustomLayoutDemo` sample app, as shown in the following screenshot, shows you how to do this. It creates 43 randomly colored boxes that vary in width and height:

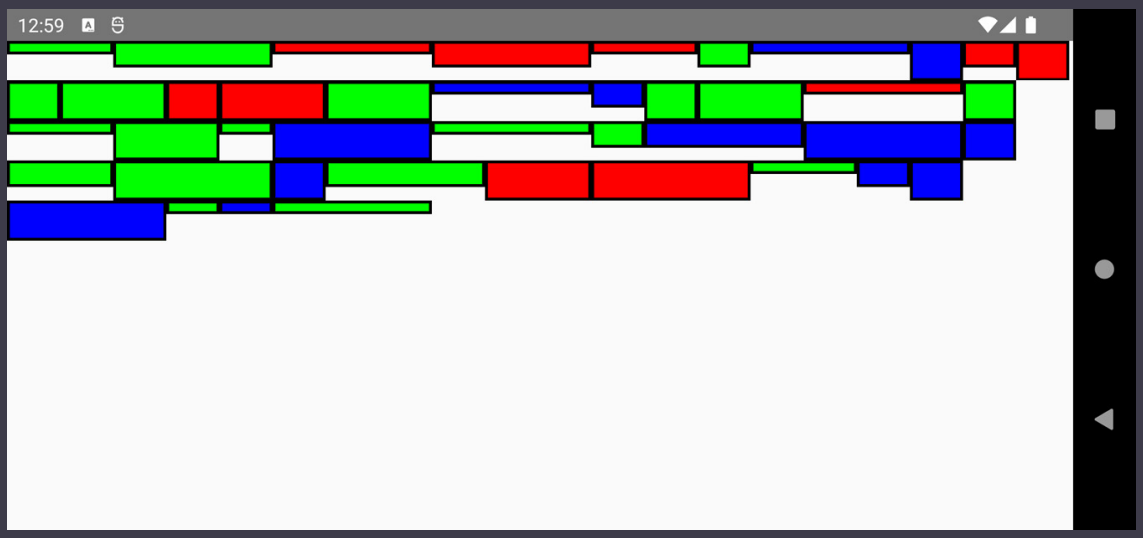

Figure 4.4 – Sample CustomLayoutDemo app

Let's start by looking at the composable function that creates colored boxes:

```
@Composable
fun ColoredBox() {
  Box(
    modifier = Modifier
      .border(
        width = 2.dp,
        color = Color.Black
      )
      .background(randomColor())
      .width((40 * randomInt123()).dp)
      .height((10 * randomInt123()).dp)
  )
}
```

A colored box consists of a **`Box()`** with a black, two density-independent pixels wide border. The **`width()`** and **`height()`** modifiers set the preferred size of the box. This means that the layout could override it. For simplicity, my example doesn't. **`randomInt123()`** randomly returns either **`1`**, **`2`**, or **`3`**:

```
private fun randomInt123() = Random.nextInt(1, 4)
```

`randomColor()` randomly returns red, green, or blue:

```
private fun randomColor() = when (randomInt123()) {
    1 -> Color.Red
    2 -> Color.Green
    else -> Color.Blue
}
```

Next, I'll show you how the colored boxes are created and set as the content of my custom layout:

```
@Composable
@Preview
fun CustomLayoutDemo() {
    SimpleFlexBox {
        for (i in 0..42) {
            ColoredBox()
```

```
            }
        }
    }
```

`SimpleFlexBox()` is our custom layout. It is used like any predefined layout. You can even provide a modifier (which has not been done here for simplicity). So, how does the custom layout work? Let's find out:

```
@Composable
fun SimpleFlexBox(
  modifier: Modifier = Modifier,
  content: @Composable () -> Unit
) {
  Layout(
    modifier = modifier,
    content = content,
    measurePolicy = simpleFlexboxMeasurePolicy()
  )
}
```

Custom layouts should receive at least two parameters – **`content`** and a **`modifier`** with a default value of **`Modifier`**. Additional parameters may influence the behavior of your custom layout. For example, you may want to make the alignment of children configurable. For simplicity, the example does not do so.

As you know from the previous section, measurement and positioning are defined through a measure policy. I will show you how to implement one in the next section.

Implementing a custom measure policy

At this point, I have shown you almost all the code for the custom layout. The only thing that's missing is the measure policy. Let's see how it works:

```
private fun simpleFlexboxMeasurePolicy():
MeasurePolicy =
```

```
    MeasurePolicy { measurables, constraints ->
      val placeables = measurables.map { measurable ->
        measurable.measure(constraints)
      }
      layout(
        constraints.maxWidth,
        constraints.maxHeight
      ) {
        var yPos = 0
        var xPos = 0
        var maxY = 0
        placeables.forEach { placeable ->
          if (xPos + placeable.width >
            constraints.maxWidth
          ) {
            xPos = 0
            yPos += maxY
            maxY = 0
          }
          placeable.placeRelative(
            x = xPos,
            y = yPos
          )
          xPos += placeable.width
          if (maxY < placeable.height) {
            maxY = placeable.height
          }
        }
      }
    }
```

`MeasurePolicy` implementations must provide implementations of **`MeasureScope.measure()`**. This function returns an instance of the **`MeasureResult`** interface. You do not need to implement this on your own. Instead, you must invoke **`layout()`**. This function belongs to **`MeasureScope`**.

We pass the measured size of the layout and a **`placementBlock`**, which is an extension function of **`Placeable.PlacementScope`**. This means that you can invoke functions such as **`placeRelative()`** to position a child in its parent's coordinate system.

A measure policy receives the content, or children, as **`List<Measurable>`**. As you know from the *Understanding the single measure pass* section, children must be measured exactly once before they are positioned. We can do this by creating a map of **`placeables`**, invoking **`measure()`** on each **`measurable`**. My example doesn't constrain child views further, instead measuring them with the given constraints.

`placementBlock` iterates over the **`placeables`** map, calculating the location of a placeable by increasing **`xPos`** and **`yPos`** along the way. Before invoking **`placeRelative()`**, the algorithm checks if a placeable completely fits into the current row. If this is not the case, **`yPos`** will be increased and **`xPos`** will be reset to **`0`**. How much **`yPos`** will be increased depends on the maximum height of all the placeables in the current row. This value is stored in **`maxY`**.

As you have seen, implementing simple custom layouts is straightforward. Advanced topics such as alignment lines (which help with/are needed for X...) are beyond the scope of this book. You can find more information about them at https://developer.android.com/jetpack/compose/layouts/alignment-lines.

Summary

This chapter explored the predefined layouts of **`Row()`**, **`Column()`**, and **`Box()`**. You learned how to combine them to create beautiful UIs. You were also introduced to **`ConstraintLayout`**, which places composables that are relative to others on the screen and flattens the UI element hierarchy.

The second main section explored why the layout system in Jetpack Compose is more performant than the traditional View-based approach.

We looked at some of the internals of the Compose runtime, which prepared us for the final main section of this chapter, *Creating custom layouts*, where you learned how to create a custom layout and thus gain precise control over the rendering of its children.

The next chapter, *Managing the State of Your Composable Functions*, will deepen your knowledge of state. We will examine the difference between stateless and stateful composable functions. Also, we will look at advanced use cases such as surviving configuration changes.