5
Beautiful Layouts with CardView and ScrollView

This is the last chapter on layouts before we spend some time focusing on Kotlin and object-oriented programming. We will formalize our learning on some of the different attributes we have already seen, and we will also introduce two more cool layouts: `ScrollView` and `CardView`. To finish the chapter off, we will run the `CardView` project on a tablet emulator.

In this chapter, we will cover the following topics:

- Compiling a quick summary of UI attributes
- Building our prettiest layout so far using `ScrollView` and `CardView`
- Switching and customizing themes
- Creating and using a tablet emulator

Let's start by recapping some attributes.

Attributes – a quick summary

In the last few chapters, we have used and discussed quite a few different attributes. I thought it would be worth a quick summary and further investigation of a few of the more common ones.

Sizing using dp

As we know, there are thousands of different Android devices. Android uses **density-independent pixels**, or **dp**, as a unit of measurement to try and have a system of measurement that works across different devices. The way this works is by first calculating the density of the pixels on the device an app is running on.

We can calculate density by dividing the horizontal resolution by the horizontal size, in inches, of the screen. This is all done on the fly on the device on which our app is running.

All we have to do is use `dp` in conjunction with a number when setting the size of the various attributes of our widgets. Using density-independent measurements, we can design layouts that scale to create a uniform appearance on as many different screens as possible.

So, problem solved then? We just use `dp` everywhere and our layouts will work everywhere? Unfortunately, density independence is only part of the solution. We will see more of how we can make our apps look great on a range of different screens throughout the rest of the book.

As an example, we can affect the height and width of a widget by adding the following code to its attributes:

```
...
android:height="50dp"
android:width="150dp"
...
```

Alternatively, we can use the attributes window and add them through the comfort of the appropriate edit boxes. Which option you use will depend on your personal preference, but sometimes one way will feel more appropriate than another in a given situation. Either way is correct and, as we go through the book making apps, I will usually point out if one way is *better* than another.

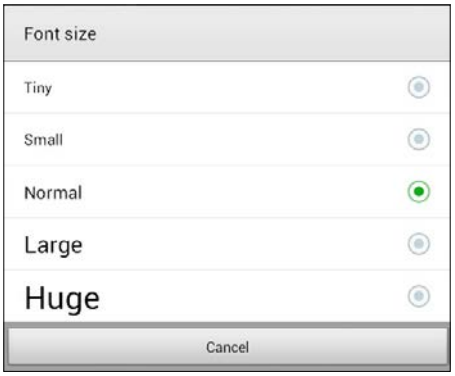
We can also use the same `dp` units to set other attributes, such as margin and padding. We will look more closely at margins and padding in a minute.

Sizing fonts using sp

Another device-dependent unit of measurement used for sizing Android fonts is **scalable pixels**, or **sp**. The `sp` unit of measurement is used for fonts, and is pixel density-dependent in the exact same way that `dp` is.

The extra calculation that an Android device will use when deciding how big your font will be based on the value of `sp` you use is the user's own font size settings. So, if you test your app on devices and emulators with normal-size fonts, then a user who has a sight impairment (or just likes big fonts) and has their font setting set to large will see something different to what you saw during testing.

If you want to try playing with your Android device's font size settings, you can do so by selecting **Settings | Display | Font size**:



As we can see in the preceding screenshot, there are quite a few settings, and if you try it on **Huge,** the difference is, well, huge!

We can set the size of fonts using `sp` in any widget that has text. This includes `Button`, `TextView`, and all the UI elements under the **Text** category in the palette, as well as some others. We do so by setting the `textSize` property as follows:

```
android:textSize="50sp"
```

As usual, we can also use the attributes window to achieve the same thing.

Determining size with wrap or match

We can also decide how the size of UI elements, and many other UI elements, behave in relation to the containing/parent element. We can do so by setting the `layoutWidth` and `layoutHeight` attributes to either `wrap_content` or `match_parent`.

For example, say we set the attributes of a lone button on a layout to the following:

```
...
android:layout_width="match_parent"
android:layout_height="match_parent"
....
```

Then, the button will expand in both height and width to **match** the **parent**. We can see that the button in the next image fills the entire screen:

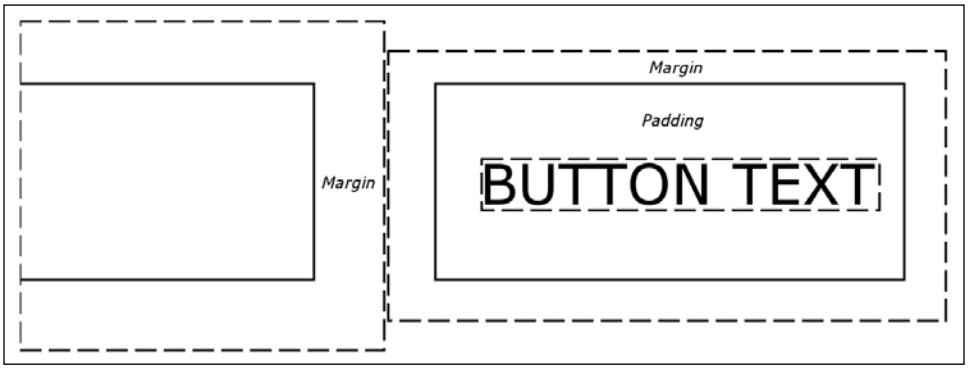
More common for a button is `wrap_content`, as shown in the following code:

```
....
android:layout_width="wrap_content"
android:layout_height="wrap_content"
....
```

This causes the button to be as big as it needs to be to **wrap** its **content** (width and height in `dp` and text in `sp`).

Using padding and margin

If you have ever done any web design, you will be very familiar with the next two attributes. **Padding** is the space from the edge of the widget to the start of the content in the widget. The **margin** is the space outside of the widget that is left between other widgets – including the margin of other widgets, should they have any. Here is a visual representation:



We can set padding and margin in a straightforward way, equally for all sides, like this:

```
...
android:layout_margin="43dp"
android:padding="10dp"
...
```

Look at the slight difference in naming convention for margin and padding. The padding value is just called `padding`, but the margin value is referred to as `layout_margin`. This reflects the fact that padding only affects the UI element itself, but margin can affect other widgets in the layout.

Or, we can specify different top, bottom, left, and right margins and padding, as follows:

```
android:layout_marginTop="43dp"
android:layout_marginBottom="43dp"
android:paddingLeft="5dp"
android:paddingRight="5dp"
```

Specifying the margin and padding values for a widget is optional, and a value of zero will be assumed if nothing is specified. We can also choose to specify some of the different side's margins and padding but not others, as in the earlier example.

It is probably becoming obvious that the way we design our layouts is extremely flexible, but also that it is going to take some practice to achieve precise results with these many options. We can even specify negative margin values to create overlapping widgets.

Let's look at a few more attributes, and then we will go ahead and play around with a stylish layout, `CardView`.

Using the layout_weight property

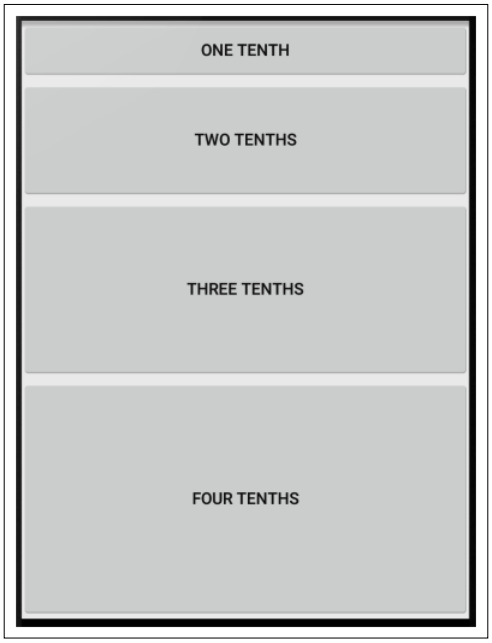
Weight refers to a relative amount compared to other UI elements. So, for `layout_weight` to be useful, we need to assign a value to the `layout_weight` property on two or more elements.

We can then assign portions that add up to 100% in total. This is especially useful for dividing up screen space between parts of the UI where we want the relative space they occupy to remain the same regardless of screen size.

Using `layout_weight` in conjunction with the `sp` and `dp` units can make for a simple and flexible layout. For example, look at this code:

```
<Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="0.10"
        android:text="one tenth" />

<Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="0.20"
        android:text="two tenths" />

<Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="0.30"
        android:text="three tenths" />

<Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="0.40"
        android:text="four tenths" />
```

Here is what this code will do:



Notice that all the `layout_height` attributes are set to `0dp`. Effectively, the `layout_weight` attribute is replacing the `layout_height` property. The context in which we use `layout_weight` is important (or it won't work), and we will see this in a real project soon. Also note that we don't have to use fractions of one; we can use whole numbers, percentages, and any other number. As long as they are relative to each other, they will probably achieve the effect you are after. Note that `layout_weight` only works in certain contexts, and we will get to see where as we build more layouts.

Using gravity

Gravity can be our friend, and can be used in so many ways in our layouts. Just like gravity in the solar system, it affects the position of items by moving them in a given direction as if they were being acted upon by gravity. The best way to see what gravity can do is to look at some example code and diagrams:

```
android:gravity="left|center_vertical"
```

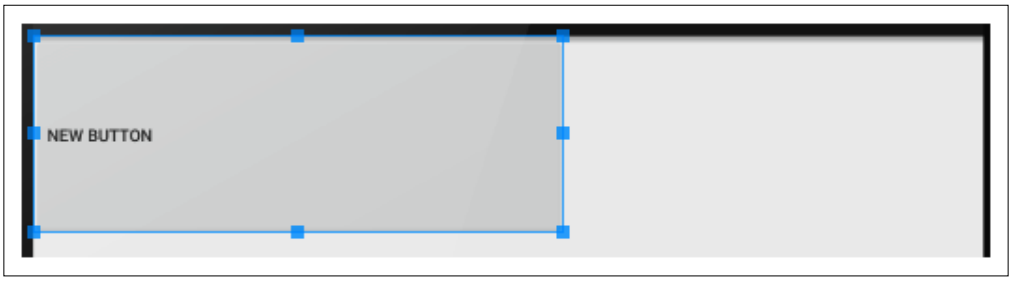
If the `gravity` property on a button (or another widget) is set to `left|center_vertical` as shown in the preceding code, it will have an effect that looks like this:

Notice that the content of the widget (in this case the button's text) is indeed aligned left and centrally vertical.

In addition, a widget can influence its own position within a layout element with the `layout_gravity` element, as follows:

```
android:layout_gravity="left"
```

This would set the widget within its layout, as expected, like this:



The previous code allows different widgets within the same layout to be affected as if the layout has multiple different gravities.

The content of all the widgets in a layout can be affected by the `gravity` property of their parent layout by using the same code as a widget:

```
android:gravity="left"
```

There are, in fact, many more attributes than those we have discussed. Many we won't need in this book, and some are quite obscure, so you might never need them in your entire Android career. But others are quite commonly used and include `background`, `textColor`, `alignment`, `typeface`, `visibility`, and `shadowColor`. Let's explore some more attributes and layouts now.

Building a UI with CardView and ScrollView

Create a new project in the usual way. Name the project `CardView Layout` and choose the **Empty Activity** project template. Leave all the rest of the settings the same as all the previous projects.

To be able to edit our theme and properly test the result, we need to generate our layout file and edit the Kotlin code to display it by calling the `setContentView` function from the `onCreate` function. We will design our `CardView` masterpiece inside a `ScrollView` layout, which, as the name suggests, allows the user to scroll through the content of the layout.

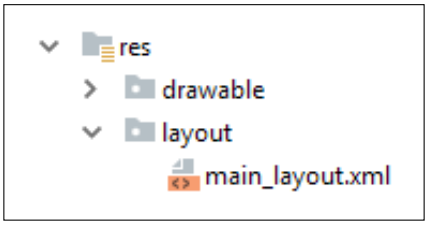
Right-click the `layout` folder and select **New**. Notice that there is an option for **Layout resource file**. Select **Layout resource file** and you will see the **New Resource File** dialog window.

In the **File name** field, enter `main_layout`. The name is arbitrary, but this layout is going to be our main layout, so the name makes that plain.

Notice that it is set to **LinearLayout** as the **Root** element option. Change it to `ScrollView`. This layout type appears to work just like `LinearLayout`, except that, when there is too much content to display on screen, it will allow the user to scroll the content by swiping with their finger.

Click the **OK** button and Android Studio will generate a new `ScrollView` layout in an XML file called `main_layout` and place it in the `layout` folder ready for us to build our `CardView`-based UI.

You can see our new file in this next screenshot:



Android Studio will also open the UI designer ready for action.

Setting the view with Kotlin code

As we have done previously, we will now load the `main_layout.xml` file as the layout for our app by calling the `setContentView` function in the `MainActivity.kt` file.

Select the `MainActivity.kt` tab. In the unlikely event the tab isn't there by default, you can find it in the project explorer under `app/java/your_package_name`, where `your_package_name` is equal to the package name that you chose when you created the project.

Amend the code in the `onCreate` function to look exactly like this next code. I have highlighted the line that you need to add:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.main_layout);
}
```

You could now run the app, but there is nothing to see except an empty `ScrollView` layout.

Adding image resources

We are going to need some images for this project. This is so we can demonstrate how to add them into the project (this section) and neatly display and format them in a `CardView` layout (next section).

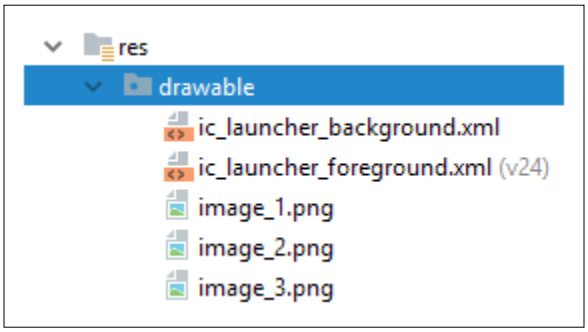
It doesn't really matter where you get your images from. It is the practical hands-on experience that is the purpose of this exercise. To avoid copyright and royalty issues, I am going to use some book images from the Packt Publishing website. This also makes it easy for me to provide you with all the resources you need to complete the project should you not want to go to the bother of acquiring your own images. Feel free to swap the images in the `Chapter05/CardViewLayout/res/drawable` folder.

There are three images: `image_1.png`, `image_2.png`, and `image_3.png`. To add them to the project, follow these steps.

1. Find the image files using your operating system's file explorer.
2. Highlight them all and press *Ctrl* + *C* to copy them.
3. In the Android Studio project explorer, select the `res/drawable` folder by left-clicking it.
4. Right-click the `drawable` folder and select **Paste.**

5. In the pop-up window that asks you to **Choose Destination Directory**, click **OK** to accept the default destination, which is the `drawable` folder.
6. Click **OK** again to **Copy Specified Files.**

You should now be able to see your images in the `drawable` folder along with a couple of other files that Android Studio placed there when the project was created, as shown in this next screenshot:



Before we move on to `CardView`, let's design what we will put inside them.

Creating the content for the cards

The next thing we need to do is create the content for our cards. It makes sense to separate the content from the layout. What we will do is create three separate layouts, called `card_contents_1`, `card_contents_2`, and `card_contents_3`. They will each contain a `LinearLayout`, which will contain the actual image and text.

Let's create three more layouts with `LinearLayout` at their root:

1. Right-click the `layout` folder and select **New layout resource file**.
2. Name the file `card_contents_1` and make sure that **LinearLayout** is selected as the **Root element**
3. Click **OK** to add the file to the `layout` folder
4. Repeat steps one through three two more times, changing the filename each time to `card_contents_2` and then `card_contents_3`

Now, select the `card_contents_1.xml` tab and make sure you are in design view. We will drag and drop some elements to the layout to get the basic structure and then we will add some `sp`, `dp`, and gravity attributes to make them look nice:

1. Drag a `TextView` widget on to the top of the layout.
2. Drag an `ImageView` widget on to the layout below `TextView` widget.

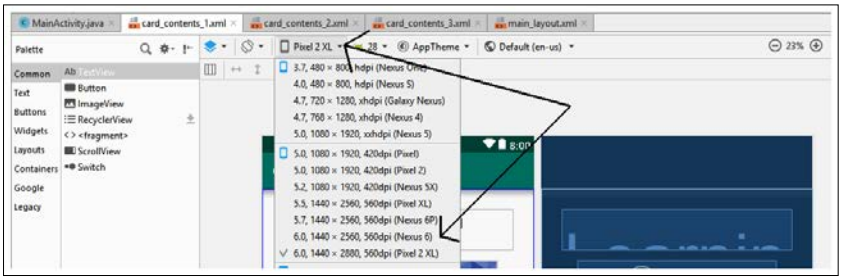
3. In the **Resources** pop-up window, select **Project | image_1** and then click **OK.**
4. Drag another two **TextView** widgets below the image.
5. This is how your layout should now appear:

Now, let's use some material design guidelines to make the layout look more appealing.

It is possible that, as you proceed through these modifications, the UI elements on the bottom of the layout might disappear from the bottom of the design view. If this happens to you, remember you can always select any UI element from the **Component Tree** window underneath the palette. Or, refer to the next tip.

Another way of minimizing the problem is to use a bigger screen, as explained in the following instructions:

I changed the default device for the design view to **Pixel 2 XL** to create the previous screenshot. I will leave this setting for the rest of the book unless I specifically mention that I am changing it. It allows a few more pixels on the layout and means this layout is easier to complete. If you want to do the same, look at the menu bar above the design view, click the device dropdown, and choose your design view device, as shown in the following screenshot:



1. Set the `textSize` attribute for the `TextView` widget at the top to `24sp`.
2. Set the **Layout_Margin | all** attribute to `16dp`.
3. Set the `text` attribute to **Learning Java by Building Android Games** (or whatever title suits your image).
4. On the `ImageView`, set `layout_width` and `layout_height` to `wrap_content`.
5. On the `ImageView`, set `layout_gravity` to `center_horizontal`.
6. On the `TextView` beneath the `ImageView`, set `textSize` to `16sp`.
7. On the same `TextView`, set **Layout_Margin | all** to `16dp`.
8. On the same `TextView`, set the `text` attribute to `Learn Java and Android from scratch by building 6 playable games` (or something that describes your image).
9. On the bottom `TextView`, change the `text` attribute to `BUY NOW`.
10. On the same `TextView`, set **Layout_Margin | all** to `16dp`.
11. On the same `TextView`, set the `textSize` attribute to `24sp`.
12. On the same `TextView`, set the `textColor` attribute to `@color/colorAccent`.
13. On the `LinearLayout` holding all the other elements, set `padding` to `15dp`. Note that it is easiest to select `LinearLayout` from the **Component Tree** window.

14. At this point, your layout will look very similar to the following screenshot:

Now, lay out the other two files (`card_contents_2` and `card_contents_3`) with the exact same dimensions and colors. When you get the **Resources** popup to choose an image, use `image_2` and `image_3` respectively. Also, change all the `text` attributes on the first two `TextView` elements so that the titles and descriptions are unique. The titles and descriptions don't really matter; it is layout and appearance that we are learning about.

Note that all the sizes and colors were derived from the material design website at `https://material.io/design/introduction`, and the Android specific UI guideline at `https://developer.android.com/guide/topics/ui/look-and-feel`. It is well worth studying alongside this book, or soon after you complete it.

We can now move on to `CardView`.

Defining dimensions for CardView

Right-click the `values` folder and select **New | Values resource file**. In the **New Resource File** pop-up window, name the file `dimens.xml` (short for dimensions) and click **OK**. We will use this file to create some common values that our `CardView` object will use by referring to them.

To achieve this, we will edit the XML directly. Edit the `dimens.xml` file to be the same as the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="card_corner_radius">16dp</dimen>
    <dimen name="card_margin">10dp</dimen>
</resources>
```

Be sure to make it exactly the same because a small omission or mistake could cause an error and prevent the project from working.

We have defined two resources, the first called `card_corner_radius`, with a value of `16dp`, and the second called `card_margin`, with a value of `10dp`.

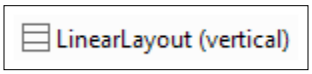
We will refer to these resources in the `main_layout` file and use them to consistently configure our three `CardView` elements.

Adding CardView to our layout

Switch to the `main_layout.xml` tab and make sure you are in the design view. You probably recall that we are now working with a `ScrollView` that will scroll the content of our app, rather like a web browser scrolls the content of a web page that doesn't fit on one screen.

`ScrollView` has a limitation – it can only have one direct child layout. We want it to contain three `CardView` elements.

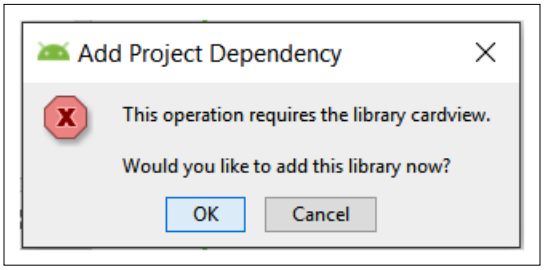
To overcome this problem, drag a `LinearLayout` from the `Layouts` category of the palette. Be sure to pick **LinearLayout (vertical),** as represented by this icon in the palette:



We will add our three `CardView` objects inside `LinearLayout` and then the whole thing will scroll nice and smoothly without any errors.

`CardView` can be found in the **Containers** category of the palette, so switch to that and locate `CardView`.

Drag a `CardView` object onto the `LinearLayout` on the design and you will get a pop-up message in Android Studio. This is the message pictured here:



Click the **OK** button, and Android Studio will do some work behind the scenes and add the necessary parts to the project. Android Studio has added some more classes to the project, specifically, classes that provide `CardView` features to older versions of Android that wouldn't otherwise have them.

You should now have a `CardView` object on the design. Until there is some content in it, the `CardView` object is only easily visible in the **Component Tree** window.

Select the `CardView` object via the **Component Tree** window and configure the following attributes:

- Set `layout_width` to `wrap_content`
- Set `layout_gravity` to `center`
- Set **Layout_Margin** | **all** to `@dimens/card_margin`
- Set `cardCornerRadius` to `@dimens/card_corner_radius`
- Set `cardEleveation` to `2dp`

Now, switch to the **Text** tab and you will find you have something very similar to this next code:

```
<androidx.cardview.widget.CardView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center"
   android:layout_margin="@dimen/card_margin"
   app:cardCornerRadius="@dimen/card_corner_radius"
   app:cardElevation="2dp" />
```

The previous code listing only shows the code for the `CardView` object.

The current problem is that our `CardView` object is empty. Let's fix that by adding the content of `card_contents_1.xml`. Here is how to do it.

Including layout files inside another layout

We need to edit the code very slightly, and here is why. We need to add an `include` element to the code. The `include` element is the code that will insert the content from the `card_contents_1.xml` layout. The problem is that, to add this code, we need to slightly alter the format of the `CardView` XML. The current format starts and concludes the `CardView` object with one single tag, as follows:

```
<androidx.cardview.widget.CardView
…
…/>
```

We need to change the format to a separate opening and closing tag like this (don't change anything just yet):

```
<androidx.cardview.widget.CardView
…
…
</androidx.cardview.widget.CardView>
```

This change in format will enable us to add the `include…` code, and our first `CardView` object will be complete. With this in mind, edit the code of `CardView` to be exactly the same as the following code. I have highlighted the two new lines of code, but also note that the forward slash that was after the `cardElevation` attribute has also been removed:

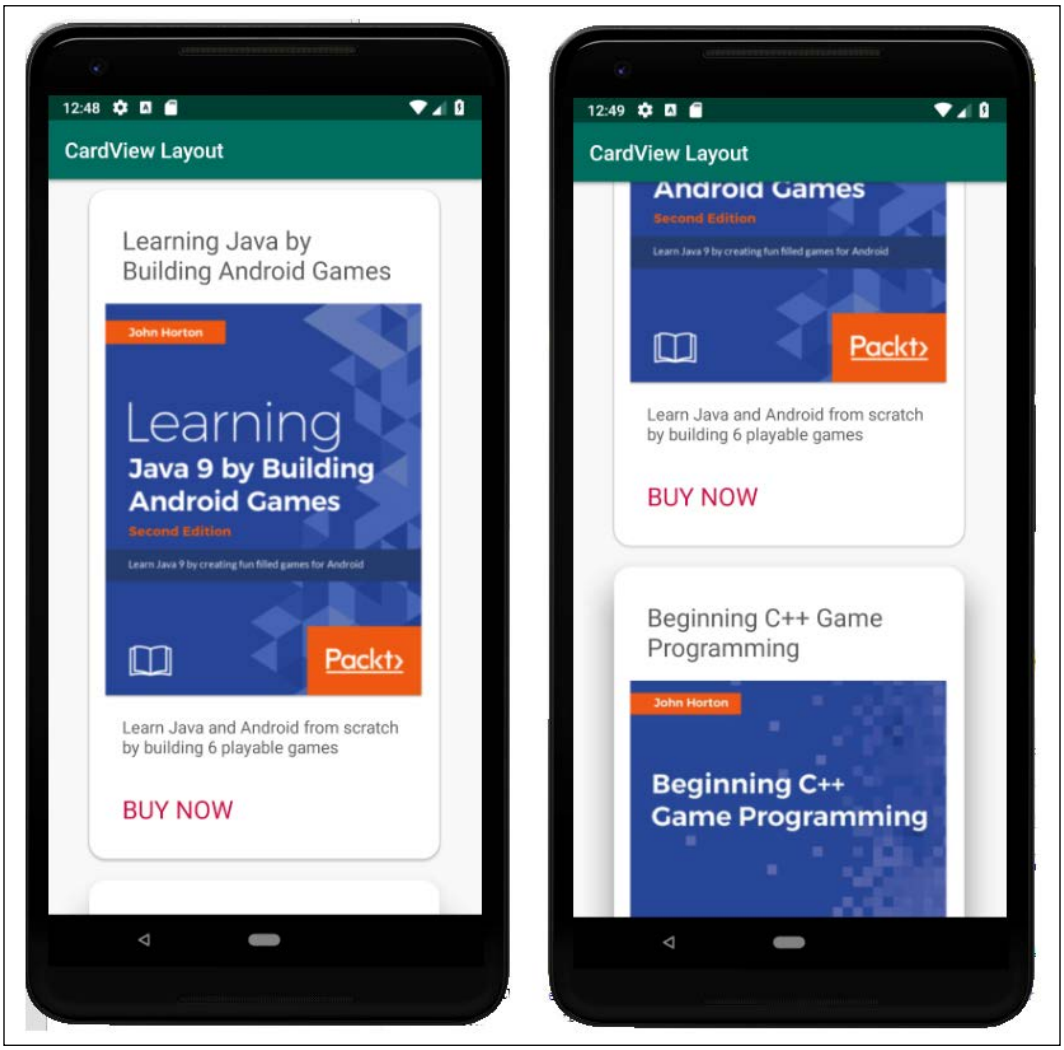
```
<androidx.cardview.widget.CardView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_gravity="center"
   android:layout_margin="@dimen/card_margin"
   app:cardCornerRadius="@dimen/card_corner_radius"
   app:cardElevation="2dp" >

   <include layout="@layout/card_contents_1" />

</androidx.cardview.widget.CardView>
```

You can now view the `main_layout` file in the visual designer and see the layout inside the `CardView` object. The visual designer does not reveal the real aesthetics of `CardView`. We will see all the `CardView` widgets scrolling nicely in the completed app shortly. Here is a screenshot of where we are up to so far:

Add two more `CardView` widgets to the layout and configure them the same as the first, with one exception. On the second `CardView` object, set `cardElevation` to `22dp` and, on the third `CardView` object, set `cardElevation` to `42dp`. Also, change the `include` code to reference `card_contents_2` and `card_contents_3` respectively.

You could do this very quickly by copying and pasting the `CardView` XML and simply amending the elevation and the `include` code, as mentioned in the previous paragraph.

Now we can run the app and see our three beautiful, elevated `CardView` widgets in action. In this next screenshot, I have photoshopped two screenshots to be side by side, so you can see one full `CardView` layout in action (on the left) and, in the image on the right, the effect the elevation setting has, which creates a very pleasing depth with a shadow effect:



The image will likely be slightly unclear in the black and white printed version of this book. Be sure to build and run the app for yourself to see this cool effect.

Now we can play around with editing the theme of the app.

Themes and material design

Creating a new theme, technically speaking, is very easy, and we will see how to do it in a minute. From an artistic point of view, however, it is more difficult. Choosing which colors work well together, let alone suit your app and the imagery, is much more difficult. Fortunately, we can turn to material design for help.

Material design has guidelines for every aspect of UI design and all the guidelines are very well documented. Even the sizes for text and padding that we used for the `CardView` project were all taken from material design guidelines.

Not only does material design make it possible for you to design your very own color schemes, but it also provides palettes of ready-made color schemes.

This book is not about design, although it is about implementing design. To get you started, the goal of our designs might be to make our UI unique and to stand out at the exact same time as making it comfortable for, even familiar to, the user.

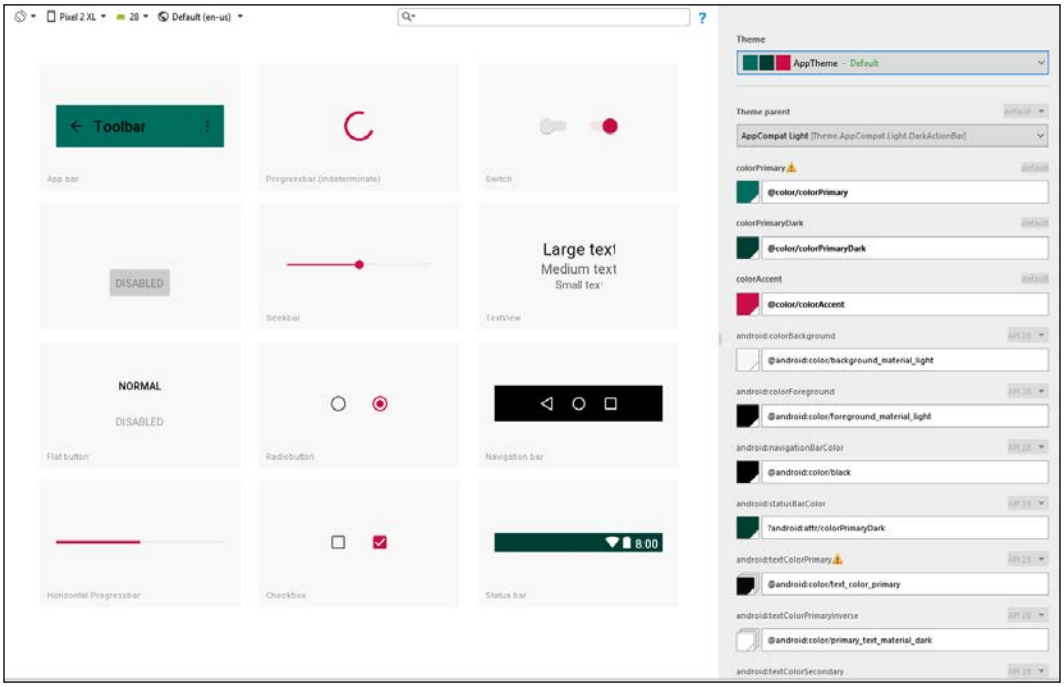
Themes are constructed from XML `style` items. We saw the `styles.xml` file in *Chapter 3*, *Exploring Android Studio and the Project Structure*. Each item in the `styles` file defined the appearance and gave it a name such as `colorPrimary` or `colorAccent`.

The questions that remain are, how do we choose our colors and how do we implement them in our theme? The answer to the first question has two possible options. The first answer is to enroll on a design course and spend the next few years studying UI design. The more useful answer is to use one of the built-in themes and make customizations based on the material design guidelines, discussed in depth for every UI element at `https://developer.android.com/guide/topics/ui/look-and-feel/`.

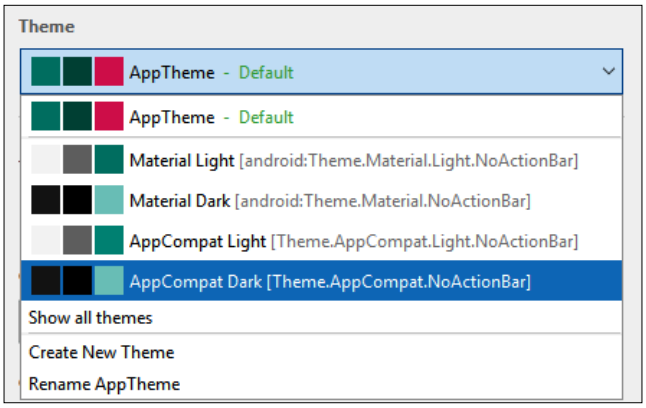
We will do the latter now.

Using the Android Studio theme designer

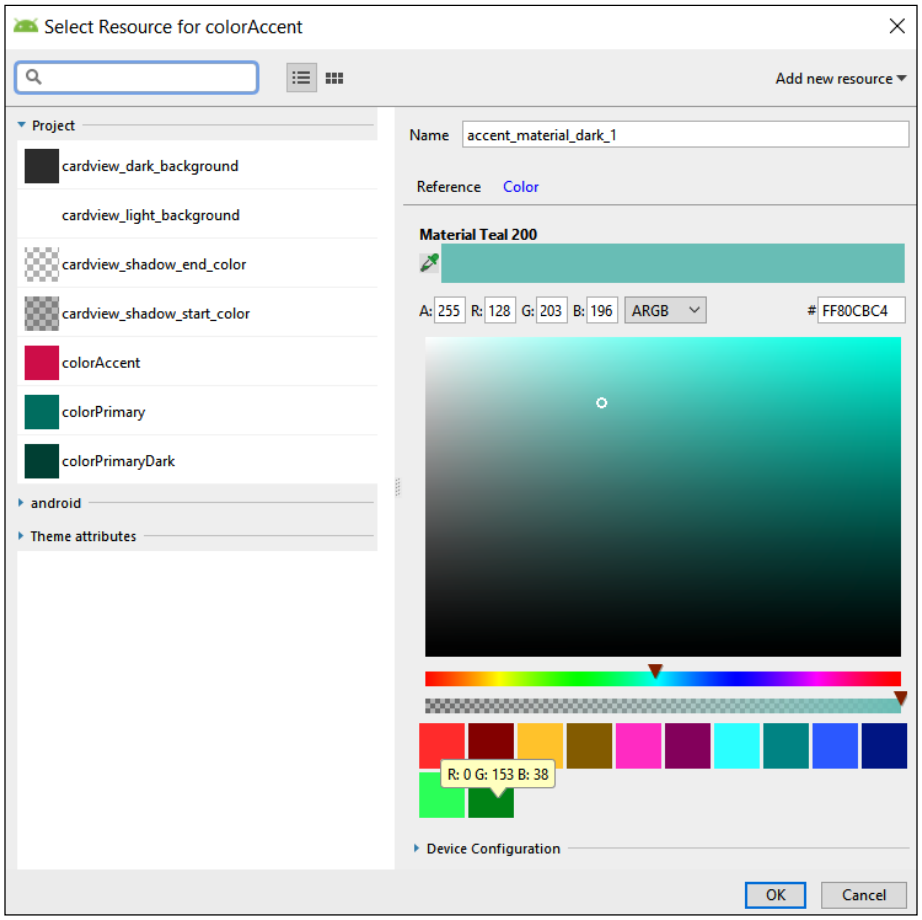
From the Android Studio main menu, select **Tools** | **Theme Editor**. On the left-hand side, notice the UI examples that show what the theme will look like, and on the right are the controls to edit aspects of the theme:



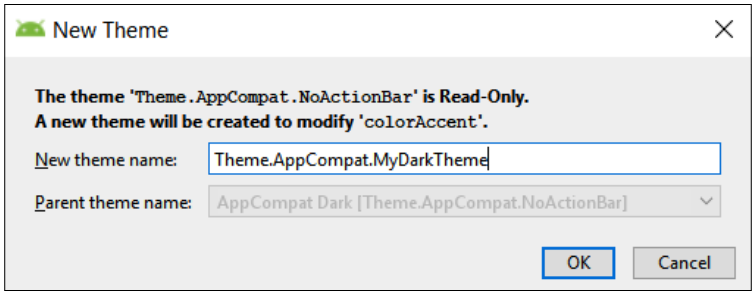
As mentioned, the easiest way to create your own theme is to start with, and then edit, an existing theme. In the **Theme** dropdown, select a theme you like the look of. I chose **AppCompat Dark**:



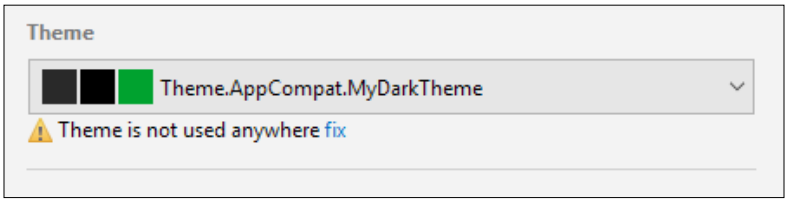
Select any items on the right-hand side that you want to change the color of, and choose a color in the screen that follows:



You will be prompted to choose a name for your new theme. I called mine `Theme.AppCompat.MyDarkTheme`:



Now, click the **fix** text to apply your theme to the current app, as indicated in the following screenshot:

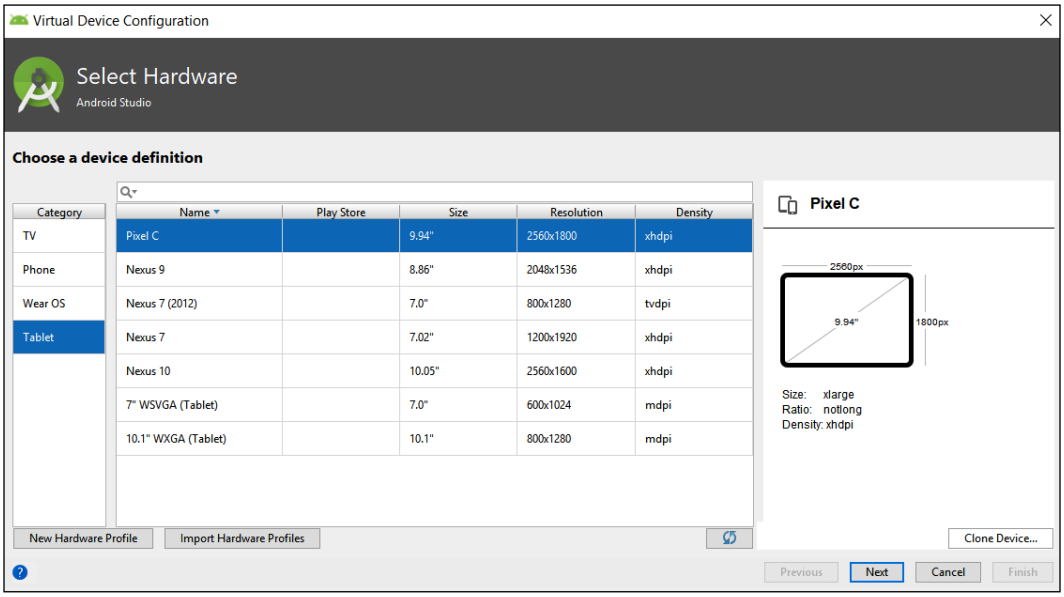


You can then run your app on the emulator to see the theme in action:

So far, all our apps have been run on a phone. Obviously, a huge part of the Android device ecosystem is tablets. Let's see how we can test our apps on a tablet emulator, as well as get an advanced look at some of the problems this diverse ecosystem is going to cause us, and then we can begin to learn to overcome these problems.

Creating a tablet emulator

Select **Tools** | **AVD Manager** and then click the **Create Virtual Device...** button on the **Your Virtual Devices** window. You will see the **Select Hardware** window in the following screenshot:

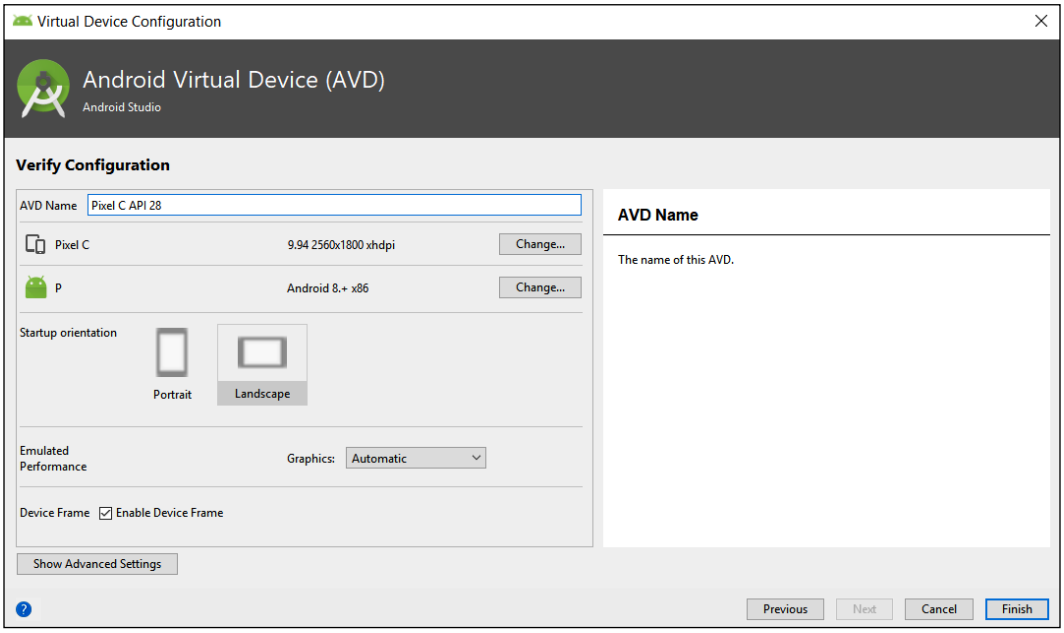


Select the **Tablet** option from the **Category** list and then highlight the **Pixel C** tablet from the choice of available tablets. These choices are highlighted in the previous screenshot.

If you are reading this sometime in the future, the Pixel C option might have been updated. The choice of tablet is less important than practicing this process of creating a tablet emulator and then testing your apps.

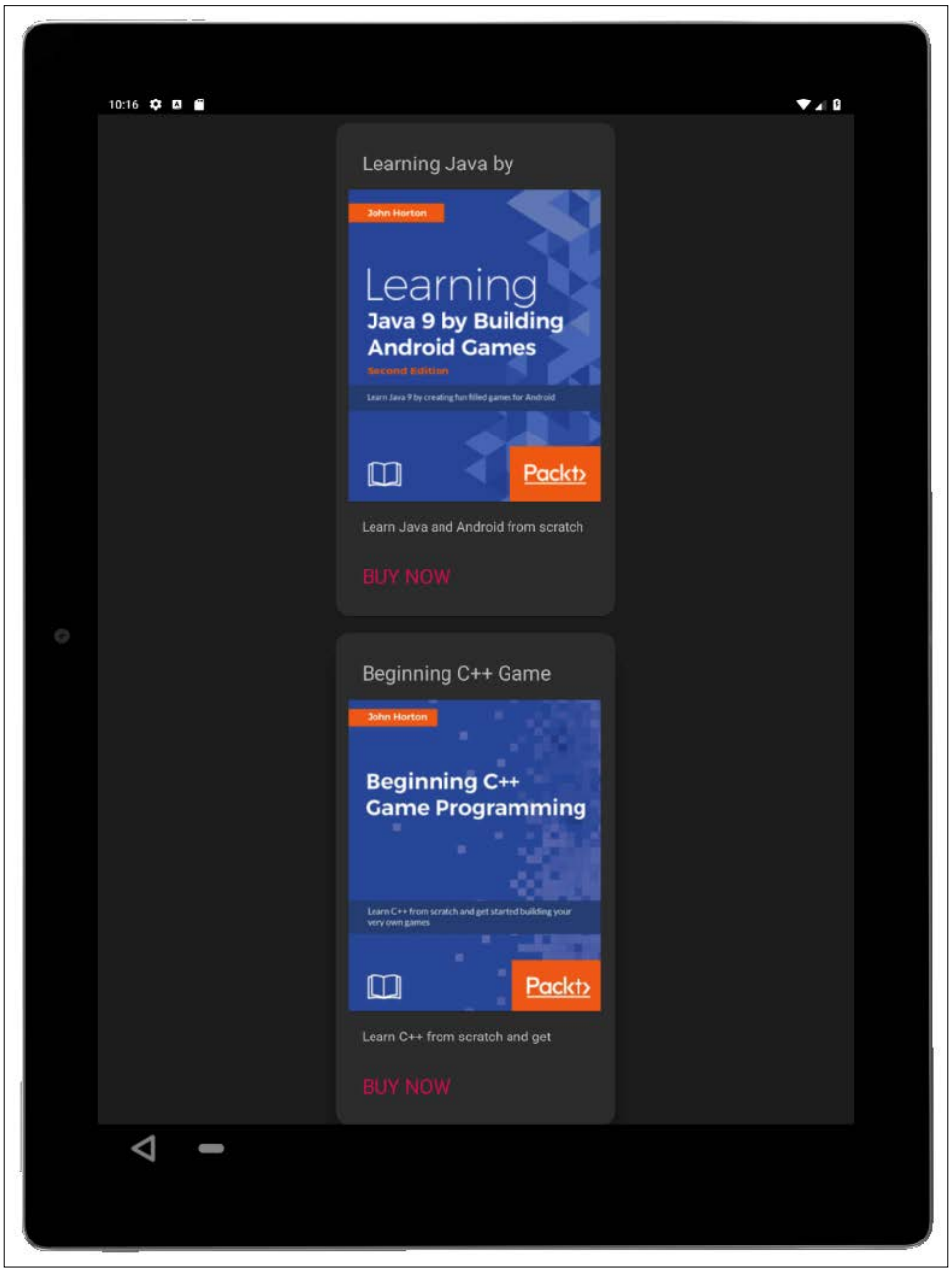
Click the **Next** button. On the **System Image** window that follows, just click **Next**, because this will select the default system image. It is possible that choosing your own image will cause the emulator not to work properly.

Finally, on the **Android Virtual Device** screen, you can leave all the default options as they are. Feel free to change the **AVD Name** for your emulator or the **Startup Orientation** (portrait or landscape) if you want to:



Click the **Finish** button when you are ready.

Now, whenever you run one of your apps from Android Studio, you will be given the option to choose **Pixel C** (or whatever tablet you created). Here is a screenshot of my Pixel C emulator running the `CardView` app:



Not too bad, but there is quite a large amount of wasted space and it looks a bit sparse. Let's try it in landscape mode. If you try running the app with the tablet in landscape mode, the results are worse. What we can learn from this is that we are going to have to design our layouts for different size screens and for different orientations. Sometimes, these will be clever designs that scale to suit different sizes or orientations, but often they will be completely different designs.

Frequently asked question

Q) Do I need to master all this stuff about material design?

A) No. Unless you want to be a professional designer. If you just want to make your own apps and sell them or give them away on the Play store, then knowing just the basics is good enough.

Summary

In this chapter, we built aesthetically pleasing `CardView` layouts and put them in a `ScrollView` layout so that the user can swipe through the content of the layout a bit like browsing a web page. To conclude the chapter, we launched a tablet emulator and saw that we are going to need to get smart with how we design our layouts if we want to cater for different device sizes and orientations. In *Chapter 24*, *Design Patterns, Multiple Layouts, and Fragments*, we will begin to take our layouts to the next level and learn how to cope with such a diverse array of devices by using Android Fragments.

Before we do so, however, it will serve us well to learn more about Kotlin and how we can use it to control our UI and interact with the user. This will be the focus of the next seven chapters.

Of course, the elephant in the room at this point is that, despite learning lots about layouts, project structure, the connection between Kotlin and XML, and much more besides, our UIs, no matter how pretty, don't actually do anything! We need to seriously upgrade our Kotlin skills while also learning more about how to apply them in an Android context.

In the next chapter, we will do exactly that. We will look at how we can add Kotlin code that executes at exactly the moment we need it to by working with the **Android Activity lifecycle**.