Community Experience Distilled

Android 5 Programming by Example

Turn your ideas into elegant and powerful mobile applications using the latest Android Studio for the Android Lollipop platform

Kyle Mew

[PACKT] open source*
PUBLISHING
community experience distilled

Android 5 Programming by Example

Turn your ideas into elegant and powerful mobile applications using the latest Android Studio for the Android Lollipop platform

Kyle Mew

[PACKT] PUBLISHING open source* community experience distilled

BIRMINGHAM - MUMBAI

Android 5 Programming by Example

First published: July 2015

Production reference: 1210715

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-844-9

www.packtpub.com

Credits

Author
Kyle Mew

Reviewers
Mustafa Gezen
Olivier Goutay
Ten Wong

Commissioning Editor
Neil Alexander

Acquisition Editor
Vivek Anantharaman

Content Development Editor
Nikhil Potdukhe

Technical Editors
Ruchi Desai
Chinmay S. Puranik

Copy Editors
Ulka Manjrekar
Swati Priya

Project Coordinator
Vijay Kushlani

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Abhinash Sahu

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Author

Kyle Mew has been programming since the early 80s and has written for several technology websites. He has also written three radio plays and another book on Android development, *Android 3.0 Application Development Cookbook*, published by Packt Publishing.

I would like to thank Rhonda for the photographs.

About the Reviewers

Mustafa Gezen is a Norwegian developer who develops applications for both Android and iOS devices. His early experiences with programming have helped him develop amazing stuff. He likes to take things apart and make them function differently. He has experience in Python, Objective-C, Java, PHP and some other languages. His main field of interest is runtime code modification.

I would like to thank my amazing family for supporting me in what I do, always believing in me, and putting their trust in me.

Olivier Goutay is an expert in Android application development. With more than 3 years of professional experience, he currently takes part in healthcare and open source Android developments in San Francisco. He currently works for Omada Health, the lead start-up in chronic disease prevention in the U.S.

Before this, he was involved with major French and European Bank mobile services and developed tomorrow's technologies, which are now used by many people on their mobile devices.

His specialties are Android development, software security, software architecture, and iOS.

Ten Wong is an embedded software engineer at Seeed Studio, an open source hardware company. He has 3 years of experience in Android driver and framework. He also has experience in TV Android app development and Arduino open source hardware shield.

I would like to thank my wife for supporting me in reviewing this book.

Table of Contents

Preface

Welcome to *Android 5 Programming by Example*, a step-by-step guide to developing Android mobile apps for the Lollipop platform. This book will take you through the installation and configuration of the development environment with the design, coding and testing processes, and on to publishing and monetizing your mobile apps.

Along the way, we will develop a series of small, working apps, designed to introduce all the familiar features of mobile apps, such as maps and touchscreen events. We will cover, in detail, the features that are new to Android 5 and higher, particularly the new design language, Material Design.

To achieve all this, you will use the Android Studio **Integrated Development Environment** (**IDE**) and the latest **Software Development Kit** (**SDK**). To test and debug apps, you will be able to use real devices connected to your computer and create virtual devices that emulate Android devices. All this software is open source and free to download and use.

The Android platform is no longer restricted to phones and tablets, and, later in the book, you will be taken on a brief tour of Android Wear, TV, and Auto, learning how to adapt and extend the existing apps and create new apps designed specifically for these form factors.

Logically, the book concludes with a detailed explanation of how to register as a Google Developer and publish your apps. To reach the widest possible number of users, you will see how to make Android 5 apps backwards compatible and include the Lollipop features on an older version. Finally, how to include in-app purchasing and advertising is covered, and at that point, you will be fully equipped to develop and distribute apps of your own invention.

What this book covers

Chapter 1, *Setting Up the Development Environment*, takes you through the installation and configuration of the SDK, Android Studio, and device emulators.

Chapter 2, *Building a UI*, helps you apply the Material theme to the portrait and landscape UIs.

Chapter 3, *Activities and Fragments*, takes you through the creation of Activities and layouts that work together.

Chapter 4, *Managing RecyclerViews and Their Data*, helps you use the Material Design CardView and RecyclerView to display data.

Chapter 5, *Detecting Touchscreen Gestures*, lets you apply touch listeners and gesture listeners to detect touchscreen events.

Chapter 6, *Notifications and the Action Bar*, lets you issue push notifications using Android's latest lock screen features.

Chapter 7, *Maps, Locations, and Google Services*, covers maps and helps you detect device location.

Chapter 8, *Apps for TVs, Cars, and Wearables*, helps you develop apps for all the three new form factors with purpose-built APIs.

Chapter 9, *Camera, Video, and Multimedia*, lets you capture, store, and play images, video, and sound using native applications and media recording widgets and classes.

Chapter 10, *Publishing and Marketing*, covers the procedure of publishing apps on the Google Play Store and monetizing them with in-app purchases and advertisements.

What you need for this book

All you need for this book is a PC with a minimum of 4 GB of RAM (although 8 GB is preferable), and an Internet connection. All the software required to follow the book is open source and can be downloaded for free.

Who this book is for

If you have a great idea for a mobile app and a little knowledge of procedural programming languages such as Java, this book is for you. No actual experience of developing apps or even working with IDEs is required.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "From a practical point of view, the SDK provides us with two versions of material theme (light and dark) and two widgets, `CardView` for simple content and `RecyclerView` for lists."

A block of code is set as follows:

```
ArrayList<MainDataDef> mainData = new ArrayList<MainDataDef>();
for (int i = 0; i < MainData.nameArray.length; i++) {
    mainData.add(new MainDataDef(
            MainData.imageArray[i],
            MainData.infoArray[i]
    ));
})
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<activity
    android:name=".DetailActivity"
    android:label="@string/title_activity_detail">
</activity>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select **Blank Activity** and leave everything else as is, or choose your own values."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Setting Up the Development Environment

Android 5 is the most significant update available at the present time, since its platform was created in 2009. It introduced a completely reworked user interface and thousands of new APIs, including the all new camera APIs. Android 5 also incorporates exciting, new, power-saving technologies, and improved app performance.

In this chapter, we will install and configure all the development tools required for building an Android app. These will include the Android Studio and Android SDK, various platform specific tools and system images, and an Android Virtual Device. Once our environment is properly set up, we will create a simple "Hello World" project and test it on a mobile device, as well as on an emulator. This exercise will give us a good opportunity to become acquainted with some of the most widely used elements of our development environment, as well as providing a quick, but useful, demonstration of how an Android Studio project is put together.

In this chapter, we will:

- Understand what is new and different about Android 5
- Download and install the Android Studio and SDK
- Install the latest SDK tools, platform-tools, and build-tools
- Install the Lollipop platform and system images
- Create a basic "Hello World" project
- Run the app on a handset
- Configure an Android Virtual Device (AVD)
- Run the app on the AVD emulator

Before we start, it's worth while taking a closer look at Android 5 itself, and see what sets it apart from other Android versions, and what it has to offer to us, as developers.

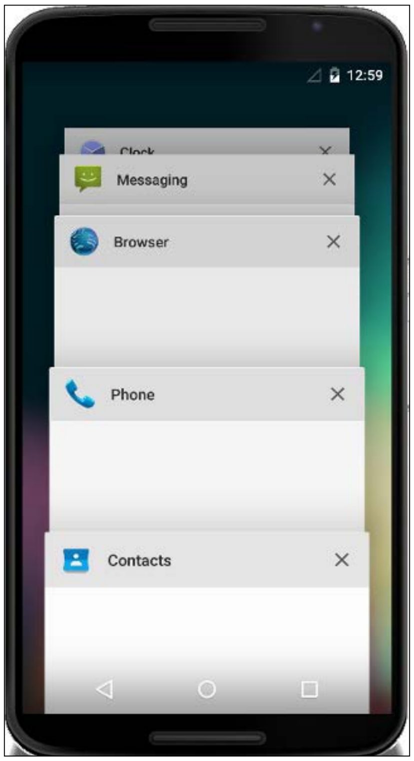
What is Android 5?

Android 5, or Lollipop, represents the most revolutionary upgrade to the Android operating system to date. It introduces many exciting, new features for users, plus a host of new APIs and access to cutting edge technologies for developers. The most significant, and obvious, changes have to be the new Material Design UI and the ability to deploy Lollipop on wearables, TVs, and in our cars.

It's not a bad idea to have a quick look at how Android 5 appears to the user, before exploring what it means for us as developers.

Lollipop from a user's perspective

The first thing any Android 5 user will will be aware of, other than the expanded and improved notifications bar and a more functional lock screen, is the new visual language-Material Design. They will notice how almost everything they touch or interact with responds with an animation. These simple onscreen behaviors are intended to provide the user with a clear, and intuitive, visual feedback. Another interesting change is the new Overview feature which replaces Recent apps, allowing individual documents and entire apps to be available.



Perhaps the most interesting departure, from the user's point of view, that Lollipop makes from previous versions is how they will now encounter it on their television sets, in their cars, and on wearable devices, such as watches and glasses. Those with these wearables will undoubtedly want these apps that take advantage of the two new sensors that Lollipop introduces, such as the heart rate and tilt sensors.

Lollipop from a developer's perspective

From a developer's point of view, Android 5 provides far more exciting prospects than a prettier UI, improved battery life, and a better lock screen. For us, with over 5,000 new APIs, a whole new design language and dozens of new features and technologies, Android 5 gives us the most powerful set of tools yet. Not only is Android now more powerful, it is also easier to program than before. With a truly helpful IDE, and APIs that are designed for ease of use, developing an app has never been simpler or less daunting. If you want to turn your ideas into a reality, then Android 5 is the way. It is truer now than it has ever been that we are limited only by the power of our imagination.

Material Design

The Material Design UI paradigm is far more than an attractive and easier to understand interface. It is a serious design language, with some important points to make about how we interact with our devices. Inspired by ideas of how materials of the future might behave, such material can be thought of as a dynamic and responsive piece of paper, which can move, change shape and size, split apart, join together, and exist in three dimensions. It is this added dimension, with real-time, programmable shadows that gives Material Design its sense of depth. The way content is displayed on material is also dynamic and Google suggests that we think of it as "smart ink". There a few design rules that need to be considered when building apps using Material Design, and we will cover these when we return to the subject in later chapters. From a practical point of view, the SDK provides us with two versions of material theme (light and dark) and two widgets: `CardView` for simple content and `RecyclerView` for lists. We can also define and customize the shadows, animations, and drawables our apps use.

Other devices

One of the most exciting opportunities that Lollipop offers us as developers is the ability to create apps for devices other than phones and tablets. Android 5 makes it possible to write apps for screens as small as wrist watches or as large as home cinemas, including anything in between.

TV

Android 5 makes coding for TV sets very similar to coding for handsets. The major differences are size, viewing distances and the way that the TV apps are generally navigated with a remote control and D-pad. The Android 5 SDK comes equipped with purpose-built themes and layouts, which make it simple for us to deploy an app built for a tablet, onto a TV, or vice versa.

Wear

When it comes to designing apps for wearables, issues such as power consumption and restricted screen size become some of the more important deciding factors. For this reason, Android 5 imposes a strict time-out policy on wearable apps. As all Android wearable apps need to be installed on a handset first, we have the opportunity to present the content on either the wearable or the parent device. Despite these restrictions, and the fact that not all features are available on wearables (such as web browsing), the addition of an API for the new heart-rate sensor, provides the developers interested in creating health and fitness based apps with new and exciting opportunities.

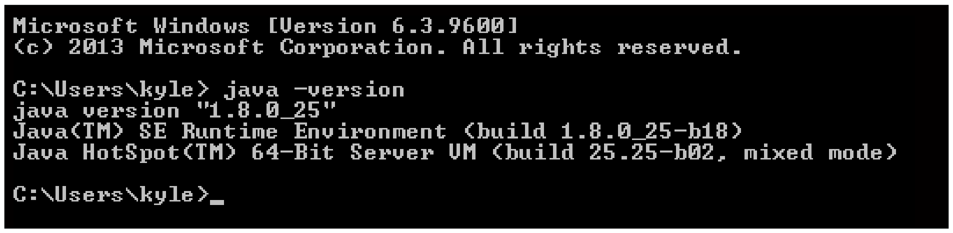
Auto

Android in our cars offers another new field introduced by Lollipop. The emphasis here rests entirely on safety: only messaging and audio features are allowed to run on in-car apps. This means that when developing apps for cars, we need to take into account which features will be disabled for safety reasons.

This is all the theory and background we will need for now. It's time to get to work and set up our work space.

Installing and configuring the development environment

Before we can even begin to work with the Android SDK, we need to make sure that we have the latest Java Development Kit (JDK) installed. It is more than likely that you have this already, but if you are unsure, type `java -version` at the command line, and hopefully you will see something like this:



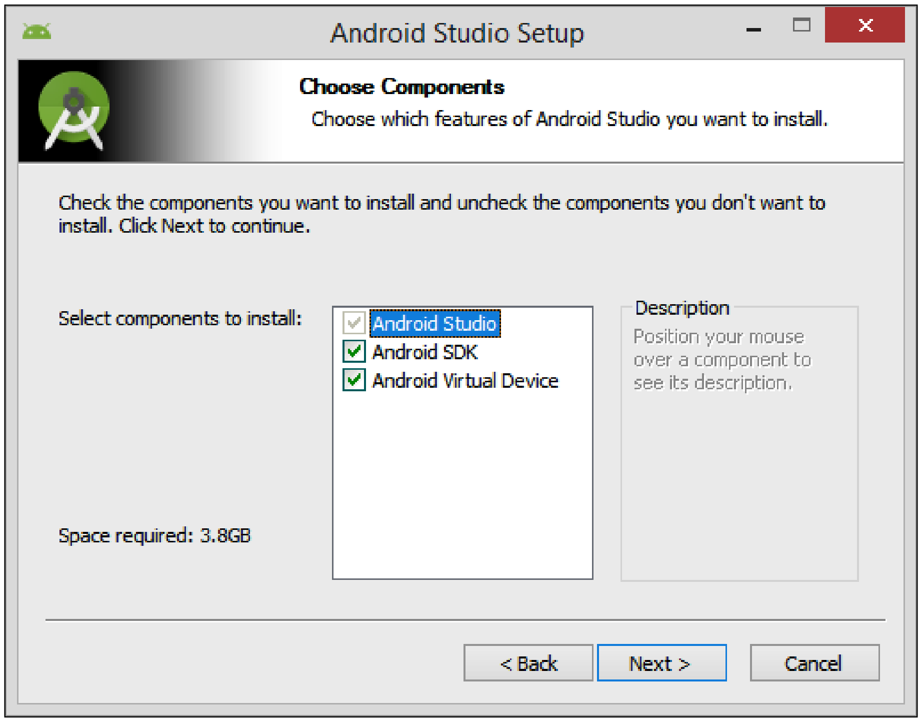
```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\kyle> java -version
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)

C:\Users\kyle>_
```

[Note that Android 5 requires Java 1.7 or higher.]

Installing the SDK

If you only have the Java Runtime Environment (JRE), you can download the JDK from `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`. You will also need to take a note of where the JDK is installed on your computer. It should be something like `C:\Program Files\Java\jdk1.8.0_25`. Let us get started with the installation, by performing the following steps:

1. Download and install the Android Studio and SDK. They can be found at `http://developer.android.com/sdk/index.html` and should come bundled together as a single executable.
2. Run the executable and follow the wizard, making sure to install all the components, as seen in the following screenshot:



Choose Components

Choose which features of Android Studio you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

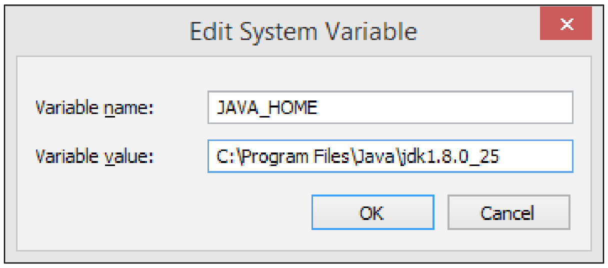
- [✓] Android Studio
- [✓] Android SDK
- [✓] Android Virtual Device

Description

Position your mouse over a component to see its description.

Space required: 3.8GB

3. Before running the Android Studio we need to set up an environment variable to point to our JDK. From your control panel's **System Properties** window, select the **Advanced** tab and then the **Environment Variables...** button. Add a new user variable called `JAVA_HOME` and paste it in the path to your JDK, as noted earlier:


Edit System Variable

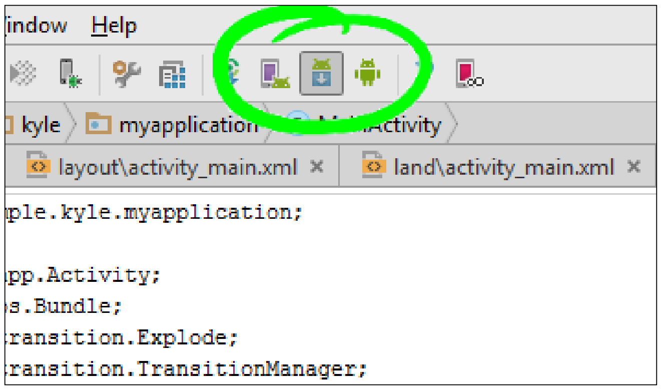
Variable name: JAVA_HOME

Variable value: C:\Program Files\Java\jdk1.8.0_25

Managing the SDK tools

There are still one or two tools we need before we can start to build and test any apps. The SDK separates tools, platforms, and device system images, allowing us to download only the packages we need or those that are specific to our project.

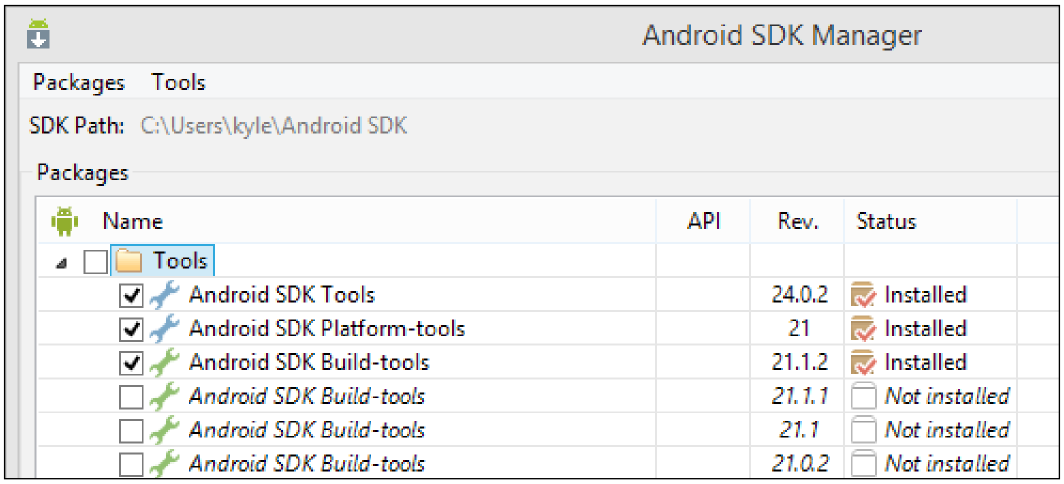
The Android SDK Manager is the program we use to do this. It can be run from the Android Studio, or as a stand-alone application from the SDK's root directory, by executing the `SDK Manager.exe` file. For now, we will run it from within the Android Studio environment. This can be done from within an open project, by clicking on the SDK Manager icon in the toolbar, or from the quick start screen from the Configure page.



```
ple.kyle.myapplication;

pp.Activity;
s.Bundle;
ransition.Explode;
ransition.TransitionManager;
```

1. Open the SDK Manager; we can see that there are three sections: a `Tools` folder, a list of API platform folders, dating right back to Android 1.5 and an `Extras` folder.
2. Open the `Tools` folder. At the top, you will see **Android SDK Tools** and **Android SDK Platform-tools**. These will represent the latest tools available and both of them have to be installed.

   Beneath this, you will see a list of build tools, and as this book is focused on Lollipop, we need to only install the latest version.



Android SDK Manager

Packages Tools

SDK Path: C:\Users\kyle\Android SDK

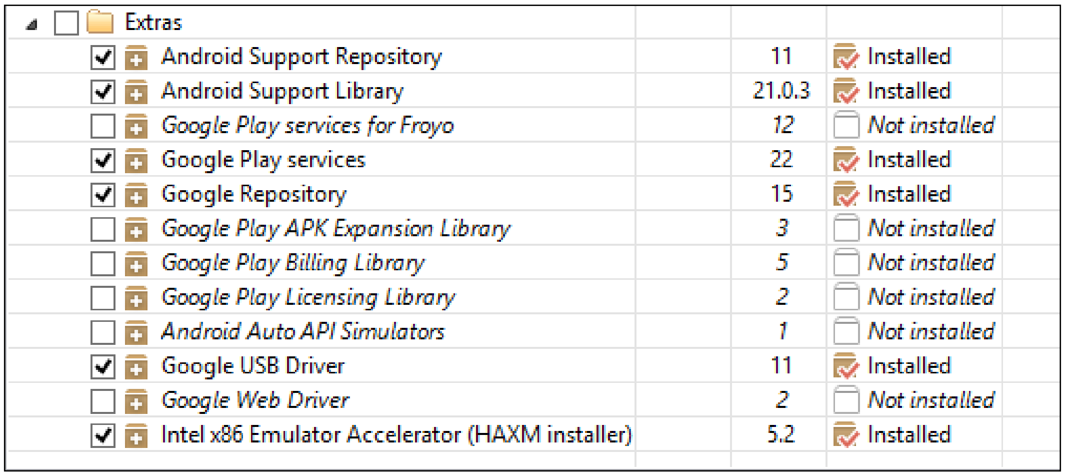
Packages

Name	API	Rev.	Status
☐ Tools			
☑ Android SDK Tools		24.0.2	Installed
☑ Android SDK Platform-tools		21	Installed
☑ Android SDK Build-tools		21.1.2	Installed
☐ *Android SDK Build-tools*		*21.1.1*	*Not installed*
☐ *Android SDK Build-tools*		*21.1*	*Not installed*
☐ *Android SDK Build-tools*		*21.0.2*	*Not installed*

3. Moving on to the list of platforms, again, all we need to do is select those that apply to us as Lollipop developers, and that means anything with an API level of 21 or higher.

   At some point, you will want to test your apps on earlier platforms to reach the widest possible market, but as this book is focused on Android 5, it is only necessary to install the most recent platform. Select all items in this folder.

There are several handy tools in the **Extras** folder. In particular, the **Android Support Library** and **Google Play services**, which offer a wide range of extra APIs. If you are planning on testing your apps on a real device connected to a PC, then you will need the **Google USB Driver** and if you have a recent Intel processor, you will want the hardware accelerator for running the emulators. Select the packages shown below, and install them.



☐ Extras			
☑ Android Support Repository	11	Installed	
☑ Android Support Library	21.0.3	Installed	
☐ *Google Play services for Froyo*	*12*	*Not installed*	
☑ Google Play services	22	Installed	
☑ Google Repository	15	Installed	
☐ *Google Play APK Expansion Library*	*3*	*Not installed*	
☐ *Google Play Billing Library*	*5*	*Not installed*	
☐ *Google Play Licensing Library*	*2*	*Not installed*	
☐ *Android Auto API Simulators*	*1*	*Not installed*	
☑ Google USB Driver	11	Installed	
☐ *Google Web Driver*	*2*	*Not installed*	
☑ Intel x86 Emulator Accelerator (HAXM installer)	5.2	Installed	

We are now ready to create our first app, but first, a note on hardware acceleration. The Emulator Accelerator needs to be executed manually and can be found in the SDK folder under `extras\intel\Hardware_Accelerated_Execution_Manager\intelhaxm-android.exe`. Depending on your system, you may also have to enable virtualization in your BIOS.

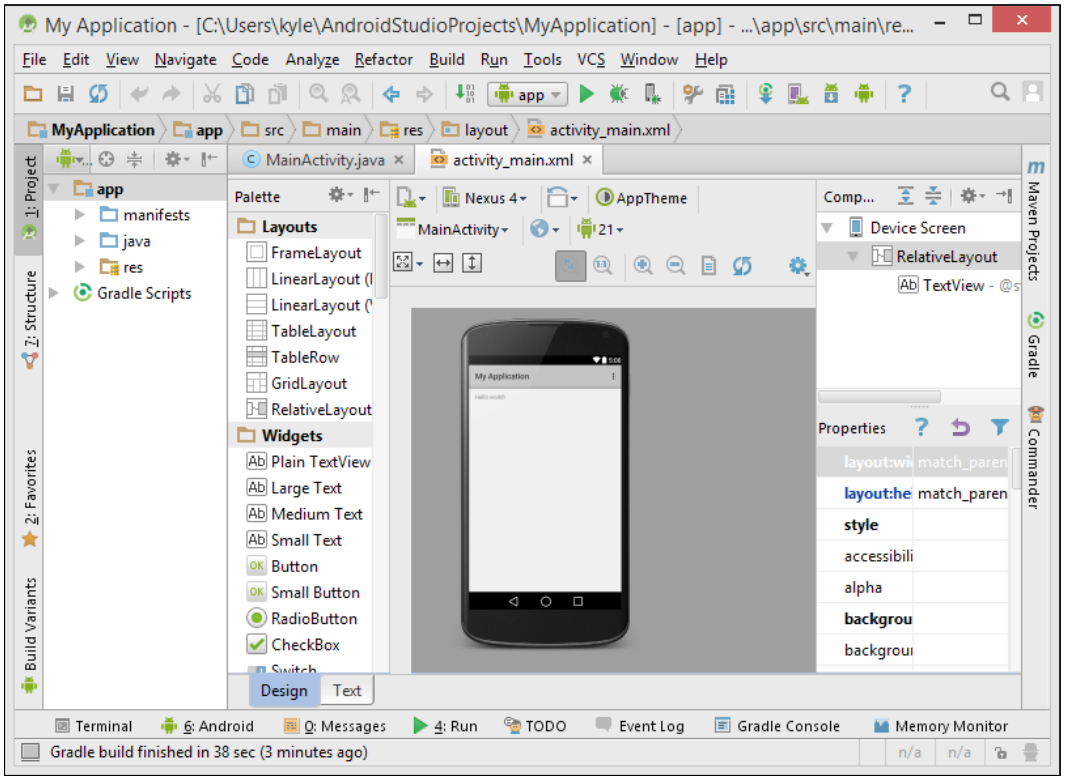
Android software is regularly updated, and it is well worth while checking back with the SDK Manager from time to time for updates, especially before starting a new project.

> Unfortunately, as Android is under constant development, installation is not always as straightforward as it appears here. Google do their best to address issues as they arrive, and in case of any problems, the web page `http://tools.android.com/knownissues` can be very useful.

Creating a "Hello World" app

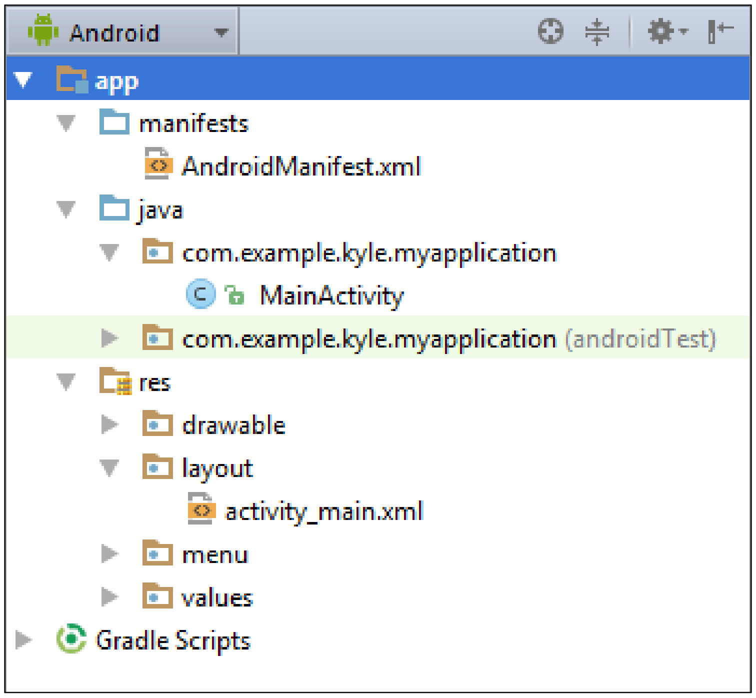
Finally, it is time to build our first app. It will do next to nothing but will give us a good look at how apps are put together by the Android Studio. We will see which files and code are automatically generated for us and get to grips with the directory structure of our project, by performing the following steps:

1. Start the Android Studio and from the start screen, select **Start a new Android Studio project**.
2. Follow the wizard, and accept the suggested values, making sure that you select the **Phone and Tablet** form factor and a minimum SDK level, no lower than API 21.
3. Select the **Blank Activity** template and accept the field values suggested by the wizard on the final screen and click on **Finish**. After a brief pause, the IDE will open up.



Gradle build finished in 38 sec (3 minutes ago)

At first sight the IDE can appear daunting, but if we approach it one element at a time it is actually very straightforward. Take a look at the panel on the left – this displays the directory structure of our project. There are two main branches, **app** and **Gradle Scripts**. We will return to Gradle, when it comes to packaging our apps, but for now, expand the **app** branch.



There are three main sections here, `manifests`, `java`, and `res` (which is short for resources). Along with this directory structure, the Android Studio also generates several code files for us automatically, so when we first create a project below including the following:

- The `manifest` file declares many of our app's broader properties, such as the permissions required by the user. It can be opened, viewed, and edited by double-clicking on the `AndroidManifest.xml` node.
- Android projects keep the layout data and procedural code separate. The layout information for the single activity we created here can be found in the XML file that was named `activity_main.xml` for us. There are two ways to view this file: the design view, which displays a list of widgets and an image of a phone and a text view, which can be accessed via the **Text** tab at the bottom of the pane. Any drawables, strings, and menu definitions we might need are also filed in the `res` directory.

- When we created the project, the IDE also created a Java class called `MainActivity`. To start with, all it contains are empty callback routines that are called when the Activity is first created, when the menu is created and when a menu item is selected.

Both Android Studio and the emulator generate a lot of temporary files during operation. This can lead to performance issues with your anti-virus software. If you are comfortable doing so, setting exclusions for Android directories can help to speed up the software considerably.

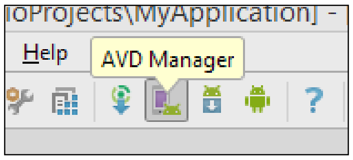
Testing the app on a physical device

It is now time to see how our app looks on an actual device and also on an emulator. To run the project on an actual phone or tablet, you will first have had to download the Google USB driver from the SDK Manager and enabled USB debugging in your handset's settings. Connect your handset and click on the **Run** button on the IDE's main toolbar, or select **Run** from the menu. After confirming the device you are using, the app will be installed and launched.

Setting up a virtual device

It is unlikely that we will always have access to all the hardware we want our apps to run on, but with Android Virtual Devices, we can create almost any form factor we can imagine and test out our apps on those. Let's set up a virtual device by performing the following steps:

1. Open the AVD Manager from the main toolbar:



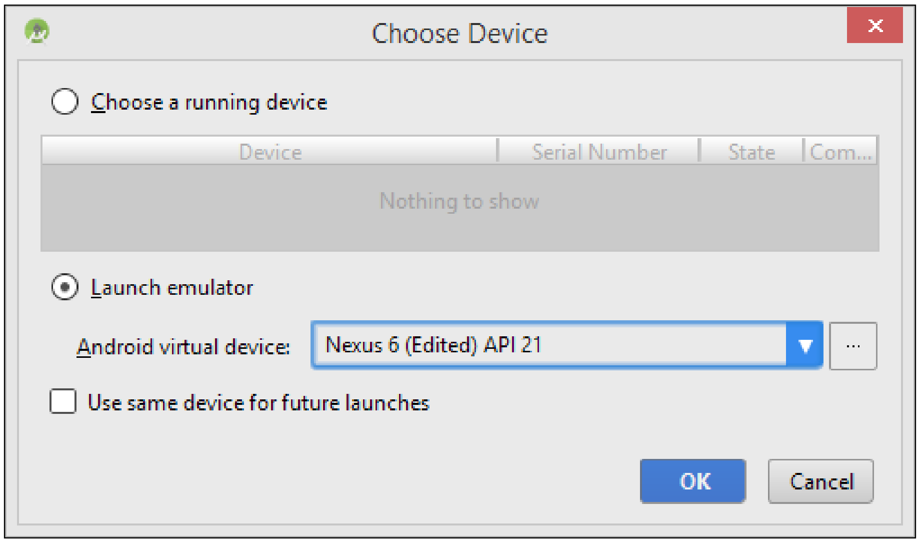
2. We could simply select one of the preconfigured devices, but to get a better understanding of how AVDs work, hit the **Create Virtual Device...** button.
3. Select **Phone** from the **Category** list.
4. Select one of the phones presented. I chose **Nexus 6**, but this is not important.
5. Click on the **Clone Device...** button, so as to maintain the original device.
6. The following page allows us to edit the device's hardware profile. Here, we get to choose features such as screen size, memory capacity, and sensors. Leave everything as it is but do have a quick look at the options available, for later.

7. Hit **Finish**.
8. You will be returned to the hardware selection screen. As before, select **Phone** from the **Category** list, and you will now find the AVD we just cloned. Select it and click on **Next**.
9. Select a system image with an API level of 21 or higher and a target of 5 or greater and click on **Next**.

> If you have the Intel hardware acceleration enabled, then choose **x86** or **x86_64** as the ABI. Otherwise, you will have to make do with ARM. For the purposes of this demonstration, I will assume that you are using an ARM ABI.

10. The following page is mainly used just to verify our settings, but it also gives us the opportunity to adjust the scale and improve the performance of the emulator. Whether or not you adjust the scale will depend on the screen you are working on and the resolution of the emulated device. Similarly, the choice between accelerating the emulator's performance by harnessing your computer's own GPU or by speeding up start times with a snapshot will depend on your hardware and your project's purpose.
11. Hit **Finish**.

Running an app on a virtual device is no different from running one on an actual phone, the only difference being which device you choose.

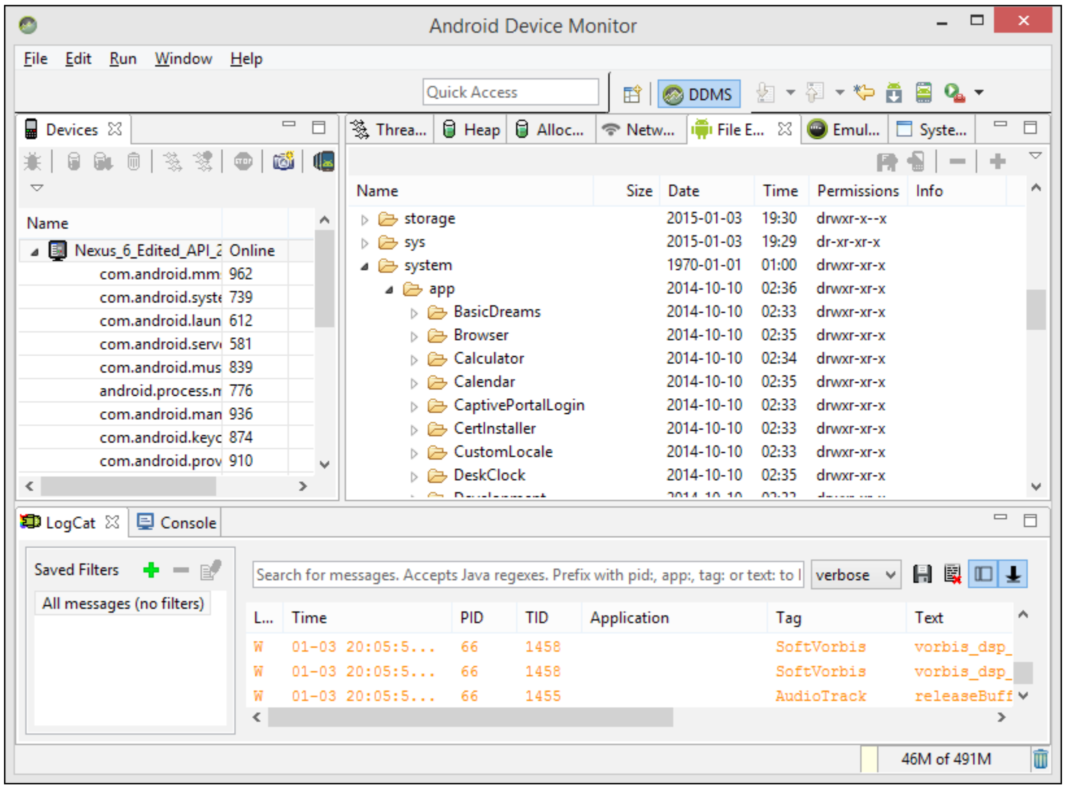


As one would imagine, the touch screen of the AVD can be operated with the mouse. The device can be rotated by pressing Left *Ctrl* + *F12*. Also, the power button can be pressed with *F7* and the volume can be turned up and down with *Ctrl* + *F5* and *Ctrl* + *F6*.

This pretty much concludes our whistle-stop tour of Lollipop, Android Studio, and the SDK. Before we get started with some actual programming, there is one last tool that it is worth taking a look at, the **Android Device Monitor**.

Monitoring devices

Run the app again, on either a device or an emulator, but first open the Android Device Monitor. It can be found on the main toolbar, to the right of the SDK Manager. The device monitor provides some very useful views on what is actually going on inside our devices and some fun ways of manipulating them. A paired down version of the Device Monitor is available directly from the IDE, by selecting Android from the icon in the lower left corner of the editor.


Android Device Monitor
File Edit Run Window Help
Quick Access
DDMS
Devices
Threa... Heap Alloc... Netw... File E... Emul... Syste...
LogCat Console
Saved Filters
All messages (no filters)
Search for messages. Accepts Java regexes. Prefix with pid:, app:, tag: or text: to l
verbose
46M of 491M

Name	
Nexus_6_Edited_API_2	Online
com.android.mm	962
com.android.syste	739
com.android.laun	612
com.android.serv	581
com.android.mus	839
android.process.n	776
com.android.man	936
com.android.keyc	874
com.android.prov	910

Name	Size	Date	Time	Permissions	Info
storage		2015-01-03	19:30	drwxr-x--x	
sys		2015-01-03	19:29	dr-xr-xr-x	
system		1970-01-01	01:00	drwxr-xr-x	
app		2014-10-10	02:36	drwxr-xr-x	
BasicDreams		2014-10-10	02:33	drwxr-xr-x	
Browser		2014-10-10	02:35	drwxr-xr-x	
Calculator		2014-10-10	02:34	drwxr-xr-x	
Calendar		2014-10-10	02:35	drwxr-xr-x	
CaptivePortalLogin		2014-10-10	02:33	drwxr-xr-x	
CertInstaller		2014-10-10	02:33	drwxr-xr-x	
CustomLocale		2014-10-10	02:33	drwxr-xr-x	
DeskClock		2014-10-10	02:35	drwxr-xr-x	

```
L...  Time              PID   TID   Application   Tag          Text
W     01-03 20:05:5...  66    1458                SoftVorbis   vorbis_dsp_
W     01-03 20:05:5...  66    1458                SoftVorbis   vorbis_dsp_
W     01-03 20:05:5...  66    1455                AudioTrack   releaseBuff
```

Along with a stream of useful debug information, the device monitor gives us direct access to its data, via a file explorer and tools that allow us to send it calls, SMS messages and set it to fake locations.

The camera icon in the **Devices** toolbar can be used to take full-sized screenshots, even when the onscreen device is scaled.

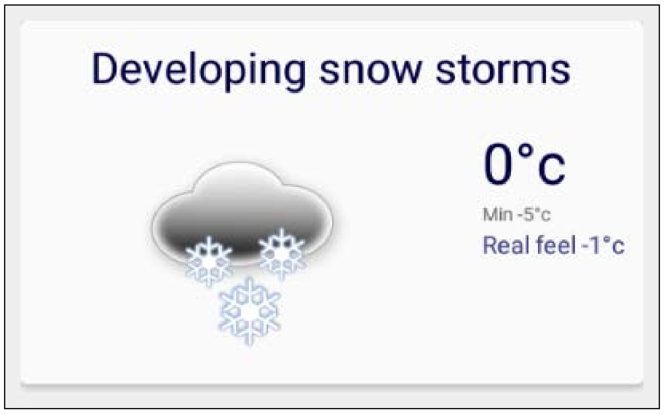
Summary

We have covered quite a lot for an introductory chapter, but we have encountered most of the essential tools that we will need later. By now, you will have a fully functional and up-to-date IDE, and know how to create and test apps on an AVD. You have seen some of the more important files and resources that are generated when an Android project is first created and should now understand where these files can be found within the project.

Having had a look around the most prominent features of our development environment, we are now in a position to dig a little deeper, and see how we can build sophisticated and dynamic layouts, looking at how a Java code can bring these designs to life.

2
Building a UI

The **Material Design** UI paradigm has brought a whole new look and feel to the Android platform. This new approach aims to give Android apps a clean and simple appearance with intuitive controls and animations. Google talks of virtual paper and virtual ink, and this concept can be seen most clearly in the new screen component (or widget), the Card (or `CardView`), which unlike previous Android widgets casts a shadow and has rounded corners.



Even before we have placed our first `CardView` widget into our layout, we can start to utilize Material Design by applying and customizing one of the material themes. These themes allow us to define a few base colors and properties which are then automatically applied throughout our app, giving it a brand identity that helps our app to be easily recognized by the user.

Having created our layout, we can then see how Java is used to provide functionality. Here, we will use a button to launch a simple Material Design animation, which we will then adapt to our layout to handle screen rotations and provide textual context to our images for users with visual impairments.

In this chapter, we will cover the following topics:

- Apply a material theme to our app
- Apply your brand colors
- Understand Material Design color guidelines
- Add new widgets to a relative layout
- Write some Java code to detect button clicks
- Write code to produce an animation
- Observe the build process with the Gradle console
- Apply accessibility options for images
- Create layouts for alternative screen orientations

In this chapter, we will continue to develop the `Hello World` app that we started in the previous chapter and use it to demonstrate a simple animation. The code can be downloaded from the Packt Publishing website and is called `Hello World - Chapter 2`.

Applying a Material Design theme

Android themes govern the general appearance of our app, controlling things like default background colors and text colors and sizes. Prior to Android 5, the Holo theme was the most widely used built-in theme, and you can preview it by clicking on the **App Theme** button at the top of the `activity_main.xml` file, when viewed in the **Design** tab.

Note that previewing a theme will have no effect on the app when it is run on a handset or an emulator, as this has to be achieved within the code.

All Android themes are highly configurable and none more so than the material theme, which allows us, with just a few lines of code, to set a color scheme that is applied across the app, and unlike its predecessors, to also change the color of the toolbar and the navigation bar. The following exercise details how such branding can be applied to the project we set up in the last chapter:

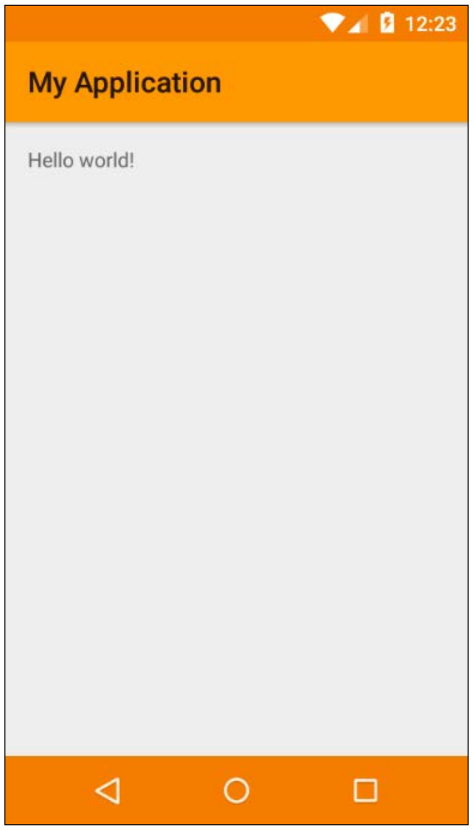
1. Open the `Hello World` project from the last chapter.
2. If it is not open already, open the **Project** tool window from the menu with **View | Tool Windows | Project**.
3. Locate the `res/values` folder and right-click on it.
4. Select **New | XML | Values XML File** from the menu and call the file `colors`.
5. Fill out the `colors.xml` file as follows here:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#FF9800</color>
  <color name="primary_dark">#F57C00</color>
  <color name="accent">#03A9F4</color>
  <color name="text_primary">#DF000000</color>
  <color name="text_secondary">#8A000000</color>>
</resources>
```

6. Open the `res/values/styles/styles.xml (v21)` file and complete it as below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

  <style name="AppTheme" parent="android:Theme.Material.Light">
    <item name="android:colorPrimary">@color/primary</item>
    <item name="android:colorPrimaryDark">
      @color/primary_dark</item>
    <item name="android:colorAccent">@color/accent</item>
    <item name="android:textColorPrimary">
      @color/text_primary</item>
    <item name="android:textColor">
      @color/text_secondary</item>
    <item name="android:navigationBarColor">
      @color/primary_dark</item>
  </style>
</resources>
```

7. You can now run the app on a handset or an emulator, to see how our brand colors have been applied:



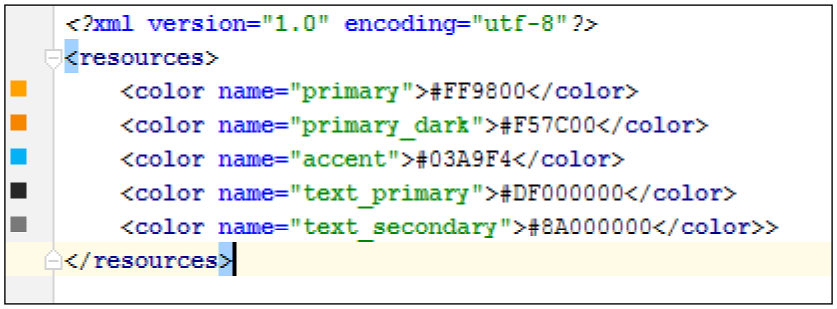
Note that not all the material theme colors will show up on the standard Android AVDs, in particular, the status and navigation bars. To view the changes, you will need a real device or one of the third-party emulators.

Being able to apply our own color scheme to the previously un-editable UI elements, such as the status and navigation bars is a huge bonus. Not only does it give us control over how the entire screen looks, but it gives our apps an identifiable and individual feel.

Android provides fields such as `colorPrimaryDark` and `navigationBarColor` as convenient ways to apply our color schemes throughout the app. It is generally recommended that the navigation bar be left black, and was colored here simply by way of demonstration. We did not use all the color attributes that we could have; had we wanted to, we could have set the window background color with `windowBackground` and `statusBarColor`, which will override it being set by default as `colorPrimaryDark`.

Note that `colorAccent` is not visible in this demonstration. It is used for switches, sliders and editable text views, among other things. It is included here as we will be using this theme (or one with colors of your choice) throughout the book and the inclusion of `colorAccent` will become evident as we progress.

Selecting colors for our theme is made remarkably easy with the help of an IDE, as you can see that the colors we have defined are displayed in the gutter:



```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="primary">#FF9800</color>
    <color name="primary_dark">#F57C00</color>
    <color name="accent">#03A9F4</color>
    <color name="text_primary">#DF000000</color>
    <color name="text_secondary">#8A000000</color>>
</resources>
```

From the `colors.xml` file, these colors can be clicked on to produce a dynamic color wheel for us to select from. Although we are free to use any colors we like for our theme, Google design guidelines suggest that colors should be picked from the recommended hues, a full list of which can be found at `http://www.google.com/design/spec/style/color.html`. Furthermore, Google also recommends that the primary color should have a value of 500 and the darker version should be 700.

In the `colors.xml` file, it can be seen that the text colors are defined with alpha channels.

Google recommends that we use transparency to produce various shades of text. In particular, they suggest around 87% opacity for our primary text and 54% for the secondary. When dealing with white text on a dark background, opacity values of 100% and 70% should be used. Edit text hints should be around 28% for either background.

You will have noticed that there are two `styles.xml` files, the `v21` version that we used and another with the same name. This other styles file is used for providing alternative themes for when we need to make our apps backward compatible. We will come to this in due course, but for now the other styles file can be safely ignored.

Material Design guidelines do not have to be followed rigidly, especially if you are designing a full-screen app such as a game. They are there to help developers build apps that provide a consistent experience across the platform, and how strictly you adhere to the guidelines is entirely up to you.

Having seen how easy it is to apply a personalized theme to our app, we can now start to add more visual components to our layout and take a look at how they can then be controlled programmatically with Java. We will continue with our `Hello World` project and to make it a little more interesting, we will add in some simple Android 5 animations.

Adding animated widgets

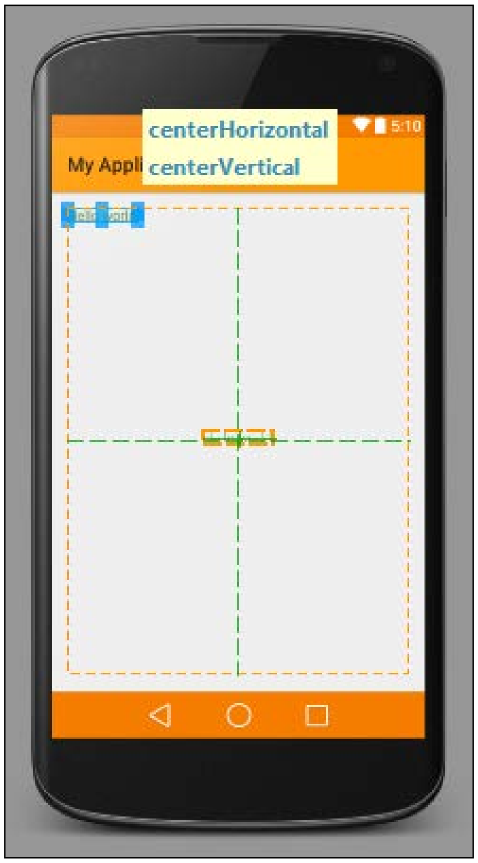
As with many programming languages, design and functionality are dealt with more or less separately. We use XML to design our layouts and Java to provide them with functionality. Here, we will see how both of these are done and will deal separately with each.

Designing an XML layout

We will be using the graphical design view to construct this UI, but it is worth checking the text view from the tab at the bottom after each step, to see how the changes that we make are applied in XML.

1. Open the `Hello World` project and then the `activity_main.xml` file.
2. Click on the **Design** tab at the bottom to view the device preview.

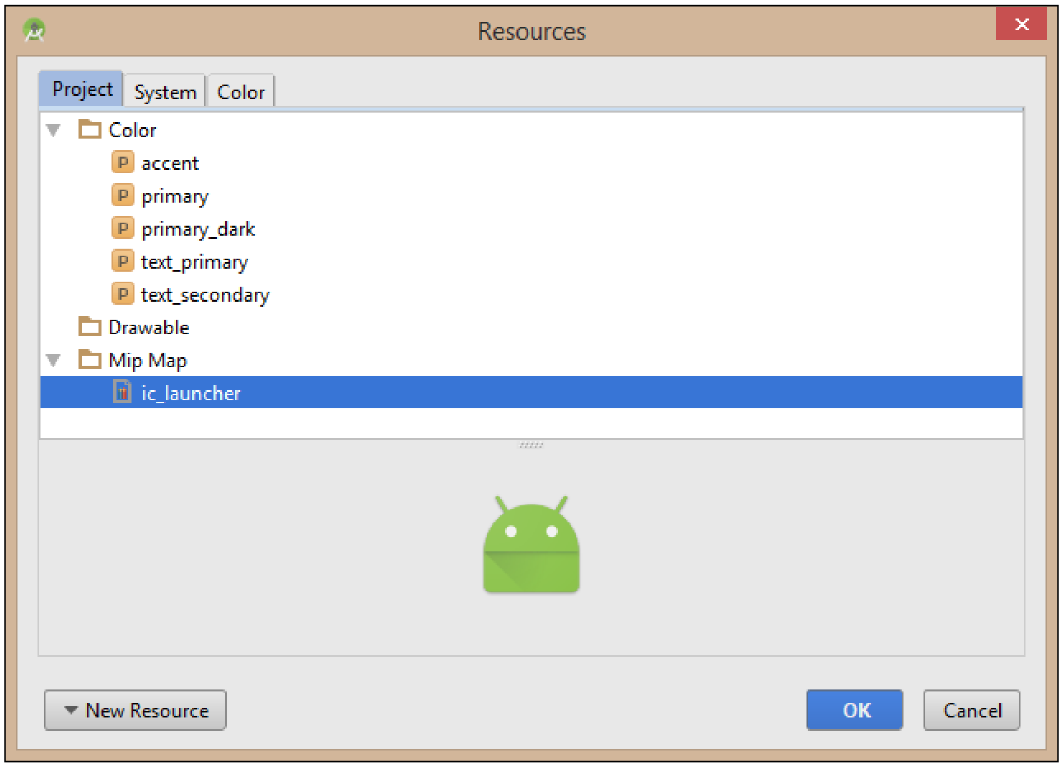
3. Drag a `TextView` control into the middle of the screen, like so:



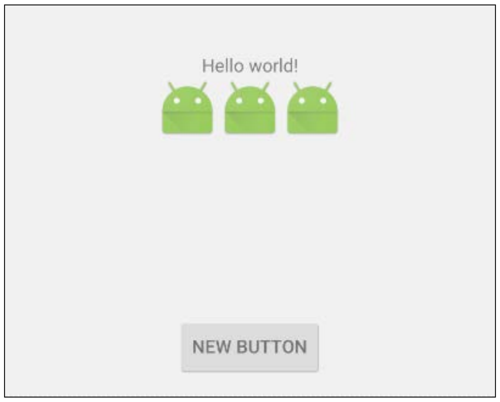
4. From the **Palette** on the left, drag and drop a **Button** widget to the bottom-center of the screen.
5. Drag an **ImageView** control from the **Palette** just under the **TextView**. The tip at the top should read:

   ```
   centreHorizontal
   below=<generated>
   ```

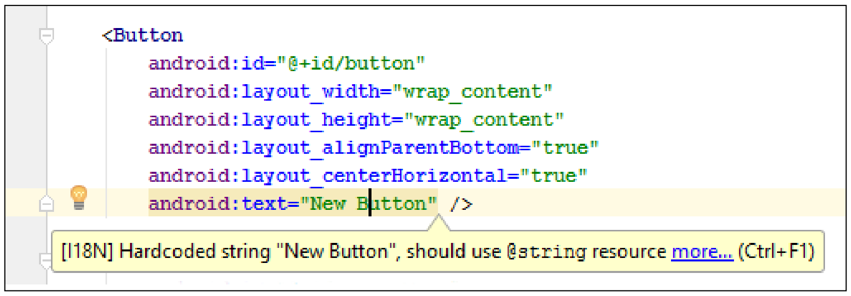
6. With the **ImageView** still selected, or by selecting it in the **Component Tree** pane, locate `src` in the **Properties** pane beneath this and click on the **....** button to bring up this dialog box:



7. Select **Mip Map** | **ic_launcher** and click on **OK**.
8. Selecting the **ImageView** in the preview window and press *Ctrl* + *C* followed by *Ctrl* + *V*.
9. Place the **ImageView** copy to the right of the one we just created.
10. Repeat this process, placing a third **ImageView** to the left, so that the lower half of the layout looks like this:



11. Now, open the **Text** view of our layout with the tab at the bottom.
12. Locate the **Button** node and click on the line that reads `android:text="New Button"`. An amber quick fix bulb will appear along with a warning about hardcoding strings.



```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_centerHorizontal="true"
    android:text="New Button" />
```

[I18N] Hardcoded string "New Button", should use @string resource more... (Ctrl+F1)

13. Click on the **quick fix** drop-down and select **Extract string resource**.
14. In the resultant dialog, provide the **Resource name** as `button_text` and click on **OK**.
15. Open the file in **Text** view and add this line to the `RelativeLayout` element:

    ```
    android:id="@+id/view_group"
    ```

16. Then, add this line to the **TextView**:

    ```
    android:textAppearance="?android:attr/textAppearanceMedium"
    ```

    That's it, as far as designing our layout is concerned!

You can use *Ctrl* + *Alt* + *L* to automatically format any code. *Ctrl* + *Alt* + *Shift* + *L* will bring up the reformatting dialog.

All Android layout designs have at their root a container object called a **ViewGroup**, in to which all other graphical objects are placed. This includes other ViewGroups, although complex layouts with lots of nested ViewGroups can have a negative effect on performance.

The ViewGroup widget container we used here was the **RelativeLayout**. There are several other types of layout, each being suited to a particular purpose, and we will encounter these throughout the book. The RelativeLayout widget container we used here allows us to define widget positions relative to other widgets. For example:

```
android:layout_below="@+id/textView
```

This is very handy when it comes to designing layouts that will run on screens of varying sizes and ratios.

There are three tools at our disposal when it comes to generating layout files. We have the (almost) WYSIWYG device preview window, that allows us to position and size the widgets, the **Properties** pane which lets us set particular values and, perhaps most powerfully, the **Text** edit window which provides control over every aspect of our design.

We set the size of our text with `android:attr/textAppearanceMedium`. We could have set the size exactly with something like `android:textSize="42sp"`, but using `textAppearanceMedium` or `textAppearanceLarge` or `textAppearanceSmall` takes into consideration the text settings that the user has configured on their phone.

By far the most important aspect of what we have just done is the fact that each widget has an identifier in the form of `android:id="@+id/some_unique_identifier"`. These IDs are how we refer to and control widgets during runtime from our Java code.

We used the built-in application icon for our **ImageView** controls as a matter of convenience, but we could have supplied our own imagery, stored it in the `res(ources)/drawable` folder and used its filename (without the extension) as its ID. We will be doing a lot of this later, so it is not necessary to worry about it here. If you take a look inside the `mipmap` folder, you will see that there are four `ic_launcher` icons, for varying screen densities. To achieve high quality icons for all available screen densities, you will need to provide all four icons.

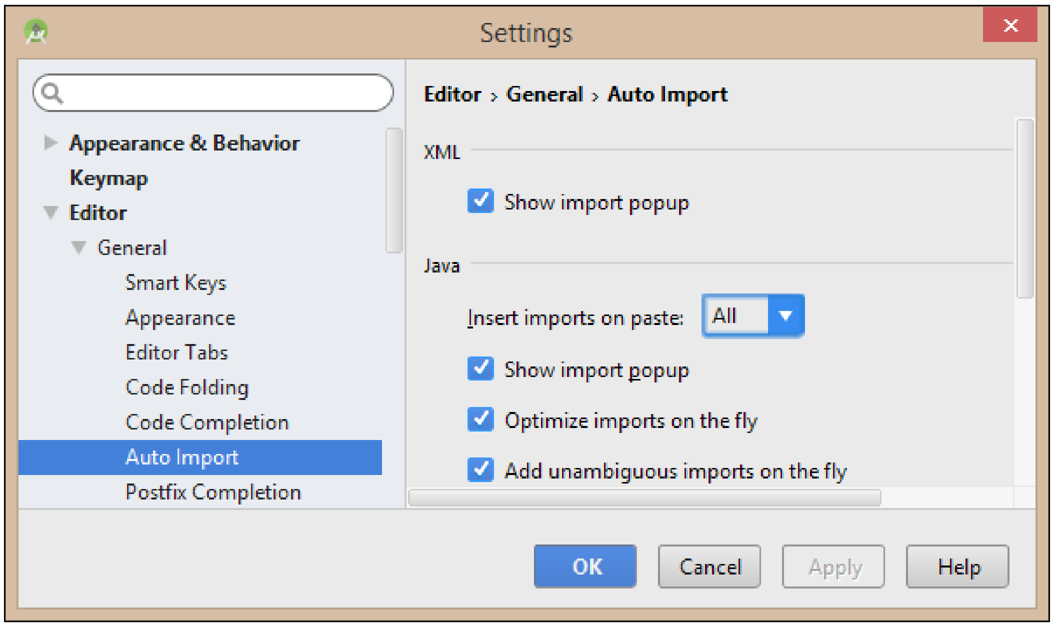
When we created the button on our screen, IDE provided the text **New Button** for us. Although a hardcoded string like this will work perfectly well, it is not recommended, for the reason that you will not be able to provide translations into other languages.

With our layout in place, we can now get on with the business of making it do something. Here, we will apply some of the new Android 5 animations which work when the button is clicked on.

Controlling the widget behavior with Java

Android 5 introduces a new and simpler way of animating screen elements. These animations are of most use when transitioning between one screen to another, and can be used to intuitively display to the user what the app is doing. This app has only one screen (`Activity`), so we will just animate our widgets to fly off the screen and then return. Before we start though, we need to configure the IDE to automatically import the Java libraries our app will use. Follow these steps to see how both tasks are done:

1. From the **File Menu** select **Settings** | **Editor** | **General** | **Auto Import** and check all boxes as below:



Editor › General › Auto Import

XML

- [x] Show import popup

Java

Insert imports on paste: All

- [x] Show import popup
- [x] Optimize imports on the fly
- [x] Add unambiguous imports on the fly

2. Open the `MainActivity.java` file.
3. At the top of the class, add these fields:

```
public class MainActivity extends Activity {
  private ViewGroup viewGroup;
  private TextView textView;
  private ImageView imageView, imageView2, imageView3;
  private Button button;
```

4. In the `onCreate()` method underneath the line `setContentView(R.layout.activity_main);` add this code:

   ```
   viewGroup = (ViewGroup) findViewById(R.id.view_group);

   textView = (TextView) findViewById(R.id.textView);
   textView.setText("Animation demo");

   imageView1 = (ImageView) findViewById(R.id.imageView);
   imageView2 = (ImageView) findViewById(R.id.imageView2);
   imageView3 = (ImageView) findViewById(R.id.imageView3);
   ```

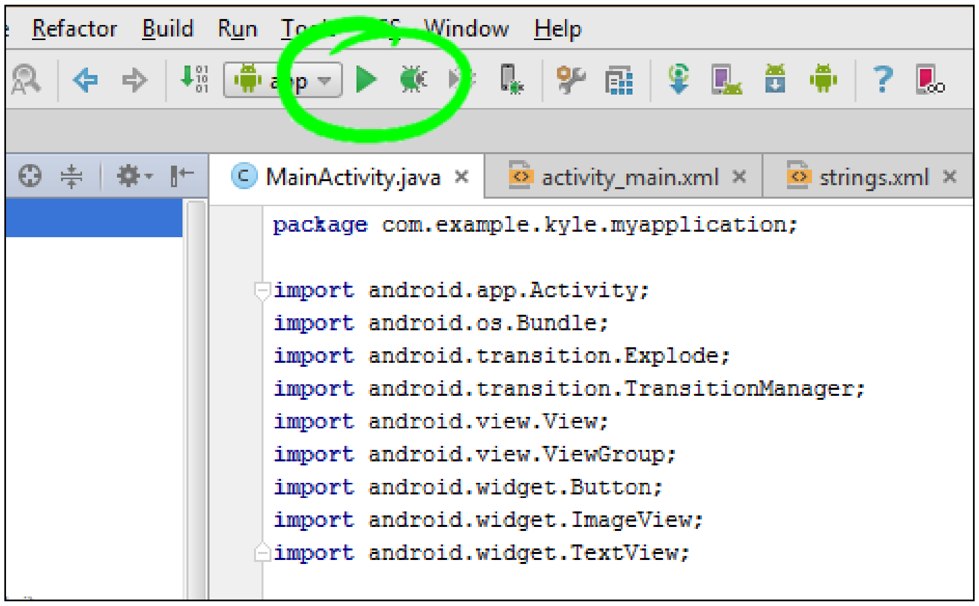
5. Beneath this, add the code for the `Button` control as follows:

   ```
   button = (Button) findViewById(R.id.button);
   button.setText("OK");
   button.setOnClickListener(new View.OnClickListener() {

     @Override
     public void onClick(View v) {
       TransitionManager.beginDelayedTransition(viewGroup,
         new Explode());
       toggle(textView, imageView, imageView2, imageView3);
     }
   });
   ```

6. Create a new method called `toggle()` and complete it like this:

   ```
   private static void toggle(View... views) {
     for (View v : views) {
       boolean isVisible = v.getVisibility() == View.VISIBLE;
       v.setVisibility(isVisible ? View.INVISIBLE :
         View.VISIBLE);
     }
   }
   ```

7. The app can now be tested on an emulator or a connected handset. Click on the run icon on the IDE toolbar:



```
package com.example.kyle.myapplication;

import android.app.Activity;
import android.os.Bundle;
import android.transition.Explode;
import android.transition.TransitionManager;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.ImageView;
import android.widget.TextView;
```

Although simple to follow, the code here covers some very important points. Firstly, there is the `onCreate()` method. This method is called as soon as the Activity is launched, usually with the application icon, and will form the start point of almost every Android app you will ever create. The first three lines, that were created for us, inflate our layout. We then used `findViewById()`, to associate our layout widgets with our Java instances.

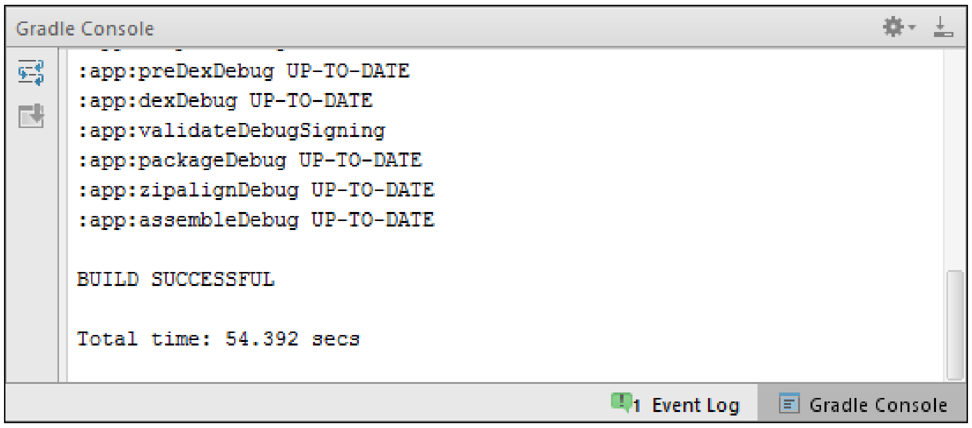
> Note that, for convenience, we used the names that the editor suggested here. In future, we will use the form `textView` to declare the Java instances and `text_view` for the XML counterparts.

We also changed the text inside two of our widgets using the `setText()` method. We could have done this from within the XML, but it is very useful to know how to do this dynamically with Java.

The `OnClickListener()` interface we attached to our button provides us with the `onClick()` method, giving us control over what actions are performed when our widgets are clicked on. There was only one button in this Activity, so we created an `OnClickListener()` specifically for it. Often, our apps will have more than one button or clickable control and, as we will see in the next chapter, we can have the Activity itself implement a click listener and then have one `onClick()` method to handle all our buttons.

The animation itself is configured and triggered with the `TransitionManager` class and we will return to this later in the book. For now, it is worth changing the term `Explode()` to `Fade()` or `Slide()`. The effect these changes have will not surprise you, but it is useful to know that they are available. Most of the time, when we apply animations to our apps, they are for the purpose of demonstrating transitions from one Activity to another, rather than being merely for decoration, as we did here.

Once you click on the run icon, the build process can be quite slow, particularly on older machines. There are, however, one or two handy tools that allow us to observe this process. Hover over the small icon in the lower left corner of the IDE and select **Gradle Console**.



```
:app:preDexDebug UP-TO-DATE
:app:dexDebug UP-TO-DATE
:app:validateDebugSigning
:app:packageDebug UP-TO-DATE
:app:zipalignDebug UP-TO-DATE
:app:assembleDebug UP-TO-DATE

BUILD SUCCESSFUL

Total time: 54.392 secs
```

It is not necessary to understand the output of the Gradle console, but it is reassuring to see that the process has not ground to a halt on longer builds. Two other very useful windows that can be accessed in the same manner are the **Android** and **Run** windows. These can also be opened from the keyboard with *Alt* + *4* and *Alt* + *6* respectively.

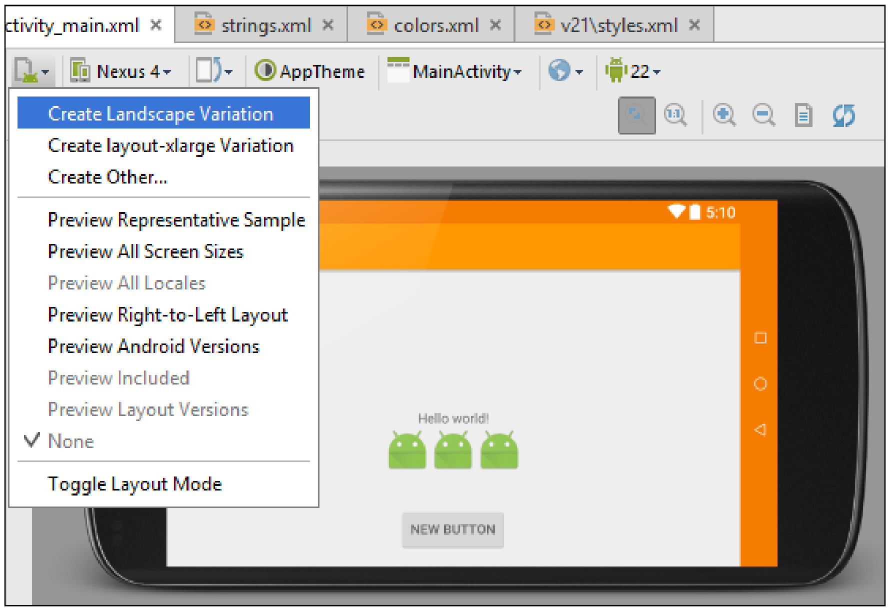
If you have spent any time experimenting with the above app, you will have noticed that when the emulator or the device is rotated through 90 degrees, one or two things do not work as we might hope. Firstly, the animation resets whenever the device is rotated. This is because this, along with any other Activity, is reloaded whenever the the orientation changes and the `onCreate()` method is called afresh. There are, however, several other callback methods that allow us to intercept this process. We will be taking a closer look at the Activity life-cycle later on, but for now we will explore the second issue, which is the way the system positions our text and images in the landscape view.

> An AVD can be rotated through 90 degrees by pressing *Ctrl* + *F12*. Pressing the home key returns the AVD to its home screen and *Esc* is the same as pressing the device's back button.

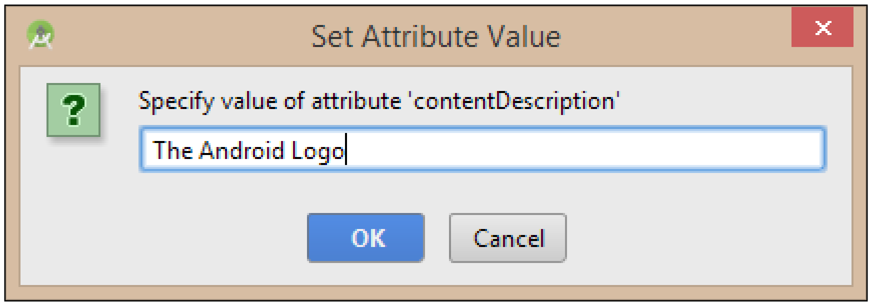
Creating alternative layouts

When a device running one of our apps is rotated into landscape orientation, it refers to the same XML file as it does in portrait mode. Often this works perfectly well, but it is incredibly simple to set up an alternative layout that better suits the shape of a landscape screen. Follow these steps to create an alternative layout file for landscape viewing.

1. Open the `activity_main.xml` file in **Design** view.
2. Click on the icon in the top-left and select **Create Landscape Variation**.



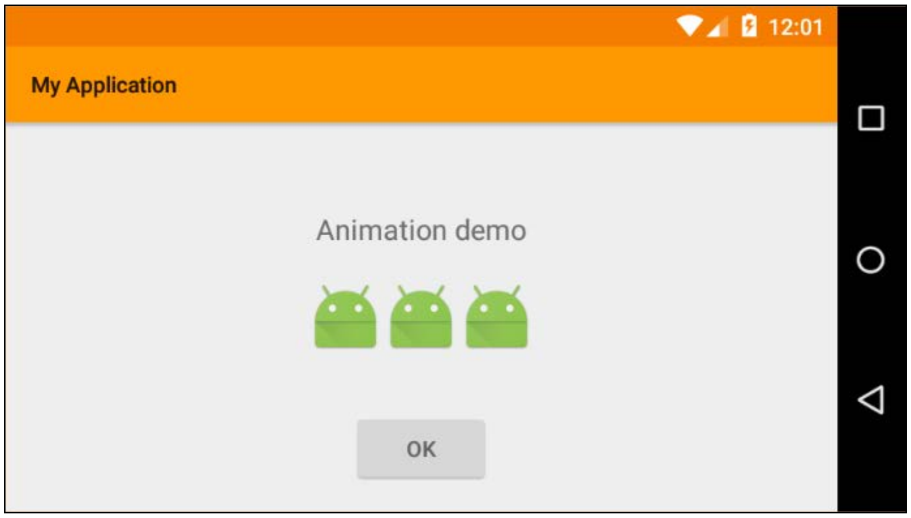
3. Drag and rearrange the onscreen widgets to form a more pleasing use of space.
4. Select one of the `ImageView` controls. Then click on the amber quick-fix icon. It will inform you that the image is missing a `contentDescription`. Click on this message and complete the resultant dialog as below:


Set Attribute Value

Specify value of attribute 'contentDescription'

The Android Logo

5. Finally, run the app on a device or an emulator and check its behavior when rotated.



Including a layout file for landscape orientation is as simple as placing a file with the same name and widget IDs in the `res/layout-land` directory instead of the `res/layout` directory. This file can then be edited in any way we like and will be automatically inflated whenever the screen is rotated to a landscape orientation.

As creating alternative layout files is such a quick and simple task, we also looked at how to provide alternative output for users with visual impairments in the form of content description for images. When a user with a visual impairment sets accessibility options and we have provided an appropriate text alternative, this description will be read out to the user. For the sake of brevity, we will not be adding this description for every exercise in this book, but it is recommended that such attributes are included in any apps intended for release.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Summary

This concludes our introduction to the relationship between XML layouts and Java code. We have seen how to produce layouts for a wide variety of screen sizes and orientations and how to connect these XML definitions to a dynamic Java code that controls our app's behavior at runtime. Significantly, we have seen how the `onCreate()` method is used to set up our app and how it is called whenever a device is rotated. We took advantage of this fact by creating an alternative layout, designed specifically for a rotated screen.

In the next chapter, we will look at how to implement the two newest Android widgets, the `CardView`, which is a convenient and stylish container for whatever information we wish to display, and the `RecyclerView`, which manages the lists of `CardView` controls, or other views, in a memory efficient manner.

3
Activities and Fragments

There are very few useful apps that run on just a single screen, or just use a single Activity; and we need a way to switch from one Activity to another and to pass information from one to another. Generally speaking, each new Activity will require its own layout file, but this not always the case; there are times when we want the same layout but with different data and resources. Later in this book, we will be building an app that acts as a tourist guide for some of the world's most famous and visited sites. We will start this journey by building a simple example for just one site that will include an introduction to the `CardView` widget introduced in Android 5, and learn how to start one Activity from another.

We will then take a look at Fragments which allow us to construct layouts in a modular fashion. Fragments behave a little like mini Activities and can be added to Activities dynamically at runtime or can be defined in a layout file like other ViewGroups. We will build a small timekeeping app that uses Fragments to switch between a digital and an analog clock face.

Next, we will include an Options menu in our app to allow the user to change time-and locale-related settings on their device. We will add action icons to the menu, so that it can be displayed on what was, prior to Lollipop, called the Action Bar. Finally, we will take advantage of the Action Bar's replacement, the Toolbar, customizing it so that it can be placed anywhere on a screen and contain far more functionality than its predecessor.

In this chapter, we will:

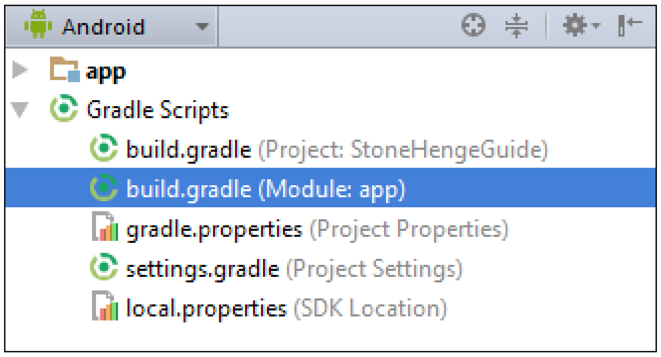
- Add a `CardView`
- Give the `CardView` a layout
- Add an image
- Create a second Activity and Layout
- Use XML to define `onClick` behavior
- Program the two activities to work together
- Use Fragments to dynamically change layouts
- Explore the Translations Editor
- Add static fragments with XML
- Include an Options menu
- Access user settings with intents
- Add menu icons to the action bar
- Replace the action bar with a custom toolbar

Adding a CardView widget

Unlike the views we have already met, `CardView` does not come included in the standard SDK libraries but rather as part of the (Lollipop specific) V7 Support Libraries and not available from the graphic layout design mode; therefore, it requires a little more work to apply one in a layout.

1. Start a new Android project.
2. Set the **Application name:** to `Stonehenge Guide`, although you can call it anything you like.
3. Select the **Phone and Tablet** form factor and **Blank Activity** from the next page.
4. Leave the other options as they are and wait for the project to build.
5. Open the `activity_main.xml` file in design view and delete the `"Hello World!"` text view that was automatically generated when we created the project.
6. For completeness, also delete the `"Hello World!"` string resource in the `res/value/strings.xml` file.
7. Edit the `styles.xml` (v21) file and create a `colors.xml` file to implement a Material Design color scheme, as we did in the last chapter.

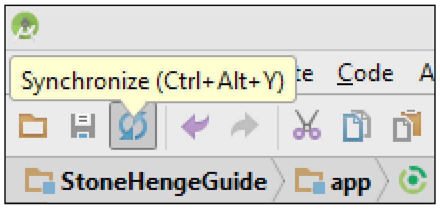
8. Open the `Gradle Scripts/build.gradle (Module: app)` file from the project explorer:


Android
app
Gradle Scripts
build.gradle (Project: StoneHengeGuide)
build.gradle (Module: app)
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)

9. Edit the `dependencies` section to match the following snippet:

```
dependencies {
  compile fileTree(dir: 'libs', include: ['*.jar'])
  compile 'com.android.support:cardview-v7:22.0.+'
}
```

10. Synchronize the project via the toolbar icon shown here:



11. Open `app/res/values/dimens.xml` and add the following three new dimension resources:

```
<dimen name="card_height">200dp</dimen>
<dimen name="card_corner_radius">4dp</dimen>
<dimen name="card_elevation">3dp</dimen>
```

12. Open the `activity_main.xml` file from `app/res/layout` and add the following `CardView` code, so that the finished layout looks like this:

```
<RelativeLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
```

```
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">

    <android.support.v7.widget.CardView
      xmlns:card_view=
      "http://schemas.android.com/apk/res-auto"
      android:layout_width="match_parent"
      android:layout_height="@dimen/card_height"
      android:layout_gravity="center"
      card_view:cardCornerRadius="@dimen/card_corner_radius"
      card_view:cardElevation="@dimen/card_elevation">
    </android.support.v7.widget.CardView>

</RelativeLayout>
```

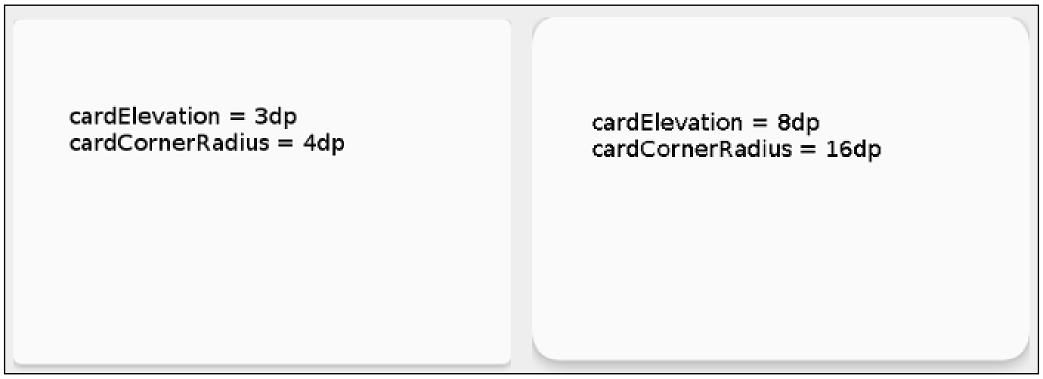
We created this project in very much the same way we did the last, and it is very useful to have initial activity and layout files created for us; this can save us a lot of time, when setting up most new projects.

As already mentioned, `CardView` is not a part of the standard libraries, which is why we had to include it in a build file, and we will have to do the same thing when we apply the other new Lollipop widget, `RecyclerView`. It is because of this that we had to **Synchronize** the project again, so that the build engine knows what libraries to load, in very much the same way that one might import a Java library. In this case, we could have just rebuilt the project from the **Build | Rebuild Project** menu item, but this won't always be the case, and it is a good idea to get into the habit of a full synchronization, as this not only rebuilds our project but also checks for other possible errors such as missing resource definitions.

Note that, when we added values to the `dimens.xml` file, there is also a `dimens.xml (w820dp)` file. This is used when designing layouts for tablets and devices wider than 820 pixels, where the margins and padding we set for smaller devices might look wrong.

It will be immediately evident from examining the XML code for the `CardView`, that it is implemented in quite a different way from the widgets we have dealt with so far. Including elements from an external library like this is very straightforward, and although we will not be covering it in this book, it is useful to know that there are a number of third-party libraries available that include many features otherwise unavailable through the standard SDK.

The CardView comes with two properties exclusive to Lollipop that we have not yet come across: cardCornerRadius and cardElevation. The purpose of these properties is obvious, but it is worth noting that the effect of changing them does not show in the preview pane and that increasing elevation only affects the widget's shadow but not its size.



Having created a CardView as a container, it's now time to provide it with some content.

Adding images and text to the layout

We will now use our CardView to display some basic information, namely a photo, a title and a short piece of text. To do this, you will need to locate where the studio stores your project files. This directory will be called AndroidStudioProjects and will more than likely be located in your Home directory or your specified save location.

1. Locate your AndroidStudioProjects directory, and open the \StonehengeGuide\app\src\main\res\drawable folder. This can be done by right-clicking on the drawable in the project explorer and selecting **Show in Explorer** from the menu.
2. Find an image and save it in the drawable directory. Any image will do; the one I use here is called stonehenge.png and is roughly 640 x 480 pixels.
3. Open your res/values/strings.xml file and add the following strings:

```
<string name="title_text">Stonehenge</string>
<string name="detail_text">One of the most famous sites in
  the world, Stonehenge is a prehistoric monument located
  in Wiltshire, England, about 2 miles west of Amesbury and
  8 miles north of Salisbury.</string>
```

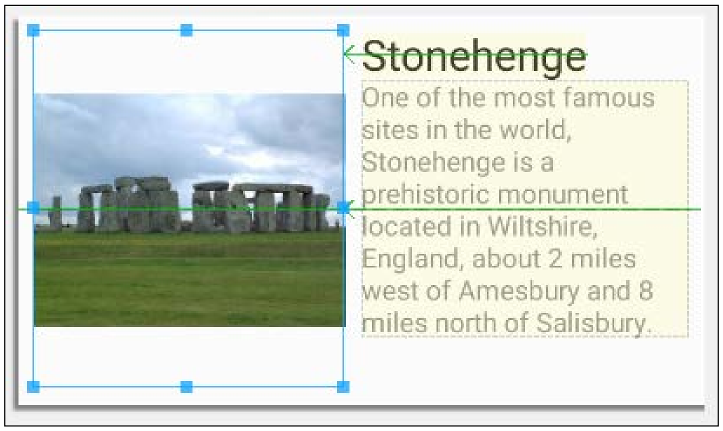
4. Open the `res/values/dimens.xml` file and add the following dimensions:

```
<dimen name="frame_width">160dp</dimen>
<dimen name="card_padding">8dp</dimen>
```

5. We need to place a `RelativeLayout` inside our `CardView`. This is not possible from graphic design mode, so drag a `RelativeLayout` anywhere on the screen and then edit the XML code so that the card looks like this:

```
<android.support.v7.widget.CardView
  xmlns:card_view="http://schemas.android.com/apk/res-auto"
  android:id="@+id/main_card_view"
  android:layout_width="match_parent"
  android:layout_height="200dp"
  android:layout_gravity="center"
  card_view:cardCornerRadius="3dp"
  card_view:cardElevation="4dp">

  <RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="@dimen/card_padding">
  </RelativeLayout>

</android.support.v7.widget.CardView>
```

6. Next, we will populate this `RelativeLayout` with a `FrameLayout` containing an `ImageView` and two TextViews, so that it looks like this:



Stonehenge

One of the most famous sites in the world, Stonehenge is a prehistoric monument located in Wiltshire, England, about 2 miles west of Amesbury and 8 miles north of Salisbury.

The best way to demonstrate the other settings and properties is with the full code for the `RelativeLayout` inside the `CardView`:

```
<RelativeLayout
  android:layout_height="match_parent"
  android:layout_width="match_parent"
  android:padding="@dimen/card_padding">

  <FrameLayout
    android:id="@+id/frameLayout"
    android:layout_alignParentStart="true"
    android:layout_centerVertical="true"
    android:layout_height="match_parent"
    android:layout_width="@dimen/frame_width">

    <ImageView
      android:clickable="false"
      android:id="@+id/imageView"
      android:layout_gravity="left|center_vertical"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:src="@drawable/stonehenge" />
  </FrameLayout>

  <TextView
    android:id="@+id/title_text_view"
    android:layout_alignParentTop="true"
    android:layout_height="wrap_content"
    android:layout_marginLeft="@dimen/card_padding"
    android:layout_toEndOf="@+id/frameLayout"
    android:layout_width="wrap_content"
    android:text="@string/title_text"
    android:textAppearance="?android:attr/textAppearanceLarge" />

  <TextView
    android:id="@+id/my_text_view"
    android:layout_below="@+id/title_text_view"
    android:layout_height="wrap_content"
    android:layout_marginLeft="@dimen/card_padding"
    android:layout_toEndOf="@+id/frameLayout"
    android:layout_width="wrap_content"
    android:text="@string/detail_text"
    android:textAppearance="?android:attr/textAppearanceSmall" />

</RelativeLayout>
```

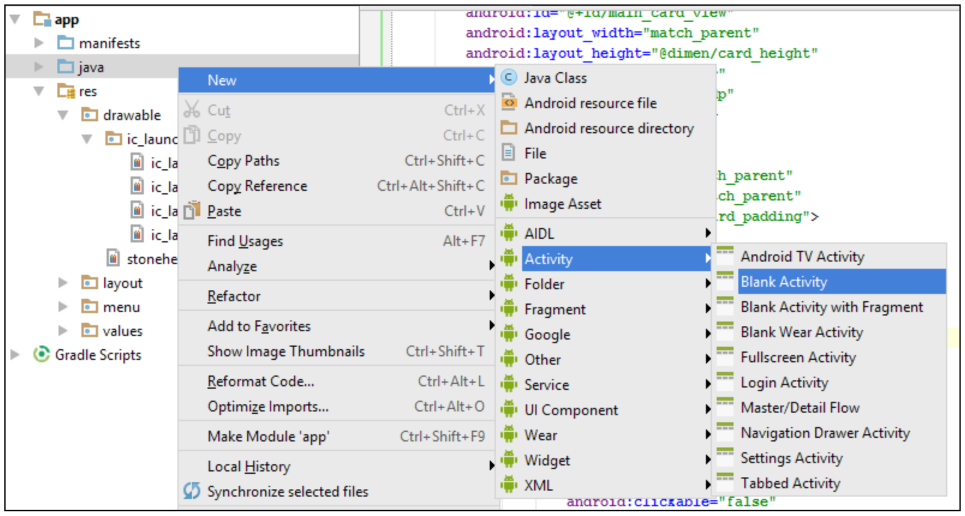
As we saw, any image file placed within a `drawable` directory in our project, becomes accessible to us in the same way that other resources do. Images placed in this folder will be available to our apps, regardless of the device they are running on. You will have noticed that there are four other drawable directories, such as the `drawable-xxhdpi` folder. Theses are particularly useful when it comes to building apps to run on a wide variety of screen densities for two reasons. Firstly, they allow us to include high-quality images for users whose devices support such screens, and secondly, they can save memory on devices that have lower screen densities as Android only loads images that can be supported by each particular physical screen.

Most of the layout features we encountered here we already met in the previous chapters and there is not much to explain, other than perhaps the `ImageView`. It is worth noting that, as well as using `android:src` to associate our photo with the `ImageView`, we could also have used `android:background`, which performs a very similar function, although it does not respect the original aspect ratio of the image. Having created our Layout, we can now move on to adding another Activity.

Creating a second Activity

So far, our app does nothing other than display information. So, next we will add some functionality by making it so that, when the user taps the image, a larger version of the picture will be shown in an another Activity. As you will see, creating new Activities with the Android Studio is very simple.

1. Right-click the Java node of the project explorer and select **New** | **Activity** | **Blank Activity**.



```
android:layout_width="match_parent"
android:layout_height="@dimen/card_height"
android:clickable="false"
```

2. In the resultant wizard, enter `ImageActivity` as the **Activity Name:**, `activity_image` as the **Layout Name:**, `image_menu` as the **Menu Resource Name**, and leave **Title:** as it is.
3. Open the `activity_image.xml` file and place a single `ImageView` inside the layout, as below:

   ```
   <ImageView
     android:id="@+id/large_image_view"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_centerHorizontal="true"
     android:layout_centerVertical="true"
     android:src="@drawable/stonehenge"
     android:onClick="returnToMainActivity"/>
   ```

4. Add the following string resource:

   ```
   <string name="title_activity_image">Stonehenge</string>
   ```

5. Open the `MainActivity` Java file and add the following code under the line `setContentView(R.layout.activity_main);` in the `onCreate()` method:

   ```
   ImageView mainImageView;
   mainImageView = (ImageView) findViewById(R.id.imageView);

   mainImageView.setOnClickListener(new View.OnClickListener() {

     @Override
     public void onClick(View v) {
       startActivity(new Intent(getApplicationContext(),
         ImageActivity.class));
     }

   });
   ```

6. Open the `ImageActivity` Java file and add the following public method to the class:

   ```
   public void returnToMainActivity(View v) {
     startActivity(new Intent(getApplicationContext(),
       MainActivity.class));
   }
   ```

7. Run the app on a device or an emulator.

The new Activity wizard conveniently created both a Java Activity and a Layout XML file for us, but it is not always necessary to have a `Layout` file associated with every Activity. Often we can use the same layout for many Activities, providing each Java Activity has a way of selecting which data to access and display.

Without us realizing it, when we created our new Activity, the wizard also modified the manifest file to include the new Activity. It is worth taking a look at, because there will be times when you will not use the wizard to create an Activity and in such cases it will be necessary to modify the `AndroidManifest.xml` file by hand.

```
<activity
  android:name=".MainActivity"
  android:label="@string/app_name" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
<activity
  android:name=".ImageActivity"
  android:label="@string/title_activity_image" >
</activity>
```

Our second Activity's Layout file contains just a single view, but there is a significant difference from the views we've explored so far, and this is the use of the `android:onClick` property. Previously we have used `View.OnClickListener()` to control how a widget behaves when clicked on. Declaring this in XML offers us an alternative way of doing this and although it lacks some of the flexibility of the Java version, it is simple and quick to use. All we had to do was declare the method called when the widget is clicked on and then add that method in Java, which here we called `returnToMainActivity()`.

In the previous chapters, we had the user click on a `Button` widget and, although this seems like an obvious choice, it is well worth noting that almost any view or widget can respond to click events.

The `StartActivity()` method takes an `Intent` as its argument. This is a vital object in any Android app and is worth taking a quick look at, as it is not only essential when working with Activities but also services and broadcasts, which form the other two main components of most apps.

[**Services** are similar to threads and run in the background and **Broadcasts** are system-wide messages that can potentially be received and acted upon by any app.]

Intents are basically descriptions of the operations we want our app to perform. They are formed of two parts, and action and the data to be acted upon. There are several `Intent()` constructors and here, we used `Intent(String action, Uri data)`. Being able to start one Activity from another is useful in many situations and here we set it in motion with a method, called by a click. There is of course, another familiar input feature that is found in nearly every mobile app, the menu, which is what we will look at next.

Applying Fragments

Using two or more Activities to create separate screens is a straightforward way to include multiple pages in our apps. However, it is not the only method, and the system also provides the **Fragment** classes. Fragments are similar to ViewGroups; in that they exist as part of an Activity, but the way that they are created and destroyed makes them behave more like mini Activities. Unlike Activities, we can also have more than one Fragment to a screen.

There are two ways to deploy Fragments in an Android app. Firstly they can be added directly to our layout XML files with the `<fragment>` tag and they can also be added and removed dynamically at run time. Although we will now look at both these techniques, it is the second, dynamic method that makes Fragments so flexible and useful.

Along with the usual main Activity layout and code, each Fragment also has an XML and a Java component, making the coding a little more complex than when working with Activities alone. In the following exercise, we will create a simple app that lets us add and replace fragments during runtime.

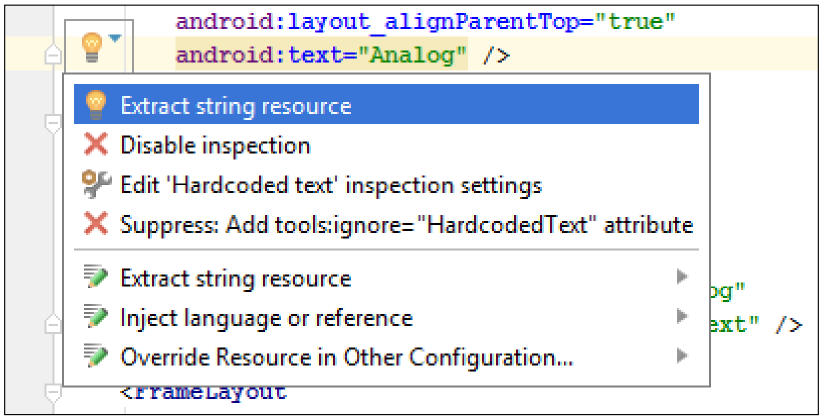
1. Start a new Android Studio project.
2. Select the **Blank Activity** template (not **Blank Activity with Fragment**), call the project `Fragment Example`, or something like that, and open the `activity_main.xml` file.
3. Change from a `RelativeLayout` to a `Linear` one by editing the code directly. The editor should change the closing tag to match as you type.
4. Set a vertical orientation for the layout with this line:

   ```
   android:orientation="vertical"
   ```

5. Replace the `TextView` with these two buttons:

```
<Button
  android:id="@+id/button_analog"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@string/button_analog_text" />

<Button
  android:id="@+id/button_digital"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@string/button_digital_text" />
```

6. Click on the line `android:text="Analog";` an amber quick fix will appear in the gutter, like below:



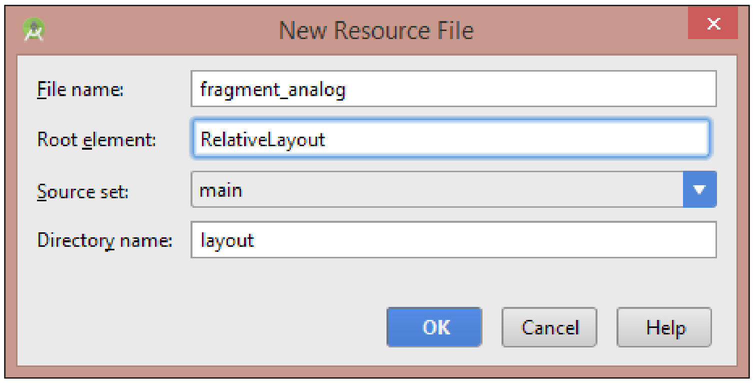
```
android:layout_alignParentTop="true"
android:text="Analog" />
```

Extract string resource
Disable inspection
Edit 'Hardcoded text' inspection settings
Suppress: Add tools:ignore="HardcodedText" attribute
Extract string resource
Inject language or reference
Override Resource in Other Configuration...

7. Click on it select **Extract string resource**, and in the resultant dialog name the string `button_analog_text`.
8. Do the same with the other button, calling it `button_digital_text`, and beneath these buttons add this `FrameLayout`:

```
<FrameLayout
  android:id="@+id/fragment_container"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

9. Right-click on the `layout` folder in the project explorer and select **New** | **New resource file**. Call it `fragment_analog` and give it a `RelativeLayout` root element, as follows:



File name: fragment_analog
Root element: RelativeLayout
Source set: main
Directory name: layout

10. Open the file and insert this `AnalogClock` inside the root element:

```
<AnalogClock
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_centerHorizontal="true"
  android:layout_centerVertical="true" />
```

11. Select `fragment_analog.xml` in the project explorer and create a copy with *Ctrl* + *C* and *Ctrl* + *V*, naming the copy `fragment_digital.xml`.
12. In this new file, replace the `AnalogClock` with this `TextClock`:

```
<TextClock
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_centerHorizontal="true"
  android:layout_centerVertical="true"
  android:textSize="48sp" />
```

13. Locate and select the folder that contains your `MainActivity.java` file in the project explorer. It will have the same name as your package.
14. From its context menu, select **New** | **Java Class** and call it `FragmentAnalog`. Fill out the class like so:

```
public class FragmentAnalog extends Fragment {

  @Override
  public View onCreateView(LayoutInflater inflater,
    ViewGroup container,
    Bundle savedInstanceState) {
    return inflater.inflate(R.layout.fragment_analog,
     container, false);
  }
}
```

15. Make a copy of this file and call it `FragmentDigital.java`.
16. Change only the layout reference in the return statement:

```
return inflater.inflate(R.layout.fragment_digital,
  container, false);
```

17. Open the `MainActivity.java` file and change the class declaration so that it implements a click listener, like below:

```
public class MainActivity extends Activity implements
  View.OnClickListener
```

18. This will generate an error and a red quick fix. Select **Implement methods** to add the `onClick()` method to the class.
19. Add these `Button` fields to the class:

```
private Button analogButton, digitalButton;
```

20. Include these four lines at the end of the `onCreate()` method:

```
analogButton = (Button) findViewById(R.id.button_analog);
analogButton.setOnClickListener(this);
digitalButton = (Button) findViewById(R.id.button_digital);
digitalButton.setOnClickListener(this)
```

21. Complete the `onClick()` method like this:

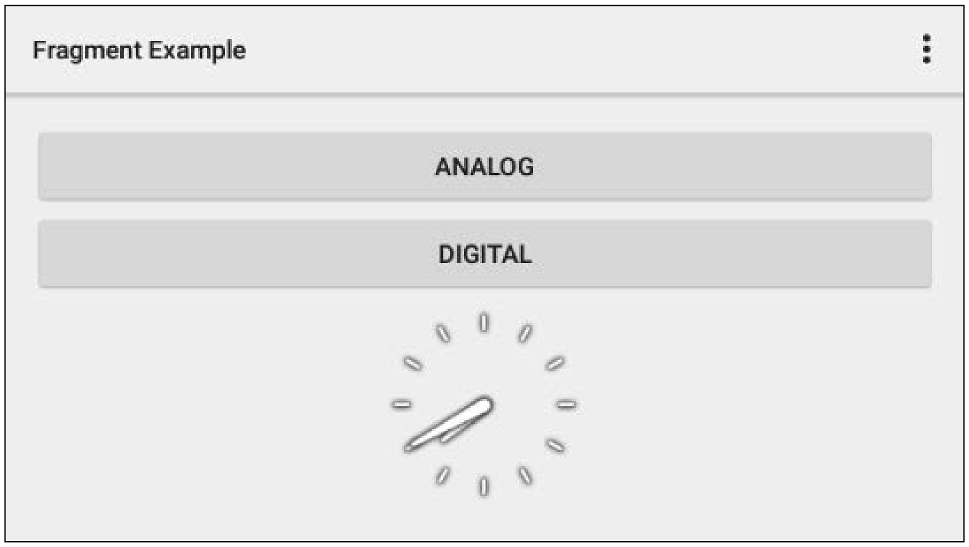
```
@Override
public void onClick(View v) {
  Fragment fragment;
  if (v == analogButton) {
    fragment = new AnalogFragment();
  } else {
    fragment = new DigitalFragment();
  }
  replaceFragment(fragment);
}
```

22. Implement the `replaceFragment()` method like below:

```
public void replaceFragment(Fragment fragment) {
  FragmentManager manager = getFragmentManager();
  FragmentTransaction transaction =
    manager.beginTransaction();
```

```
        transaction.setTransition(
          FragmentTransaction.TRANSIT_FRAGMENT_FADE);
        transaction.replace(R.id.fragment_container,fragment);
        transaction.addToBackStack(null);
        transaction.commit();
    }
```

23. You can now run the app on a device or an emulator.



We started this exercise by creating a simple layout, but we also used a handy shortcut to avoid having **hardcoded** strings in our layouts. Doing this makes creating translated versions of our apps very simple and saves a lot of work when creating alternative layouts. It is not necessary for exercises such as those in this book to follow this practice, and to save time we will not concern ourselves with it further.

The Translations Editor can be opened by right-clicking on the `res/values/strings.xml` file in the project explorer and selecting **Open Translation Editor (Preview)**. This editor makes translating Android apps very straightforward.

We also added an empty **FrameLayout** to serve as the container for our fragments we could have used any ViewGroup, but the FrameLayout is the simplest. Fragments are like Activities in that they have both an XML and a Java component, and here we created two very simple fragments just to see how they can be included and replaced dynamically. Fragments can of course contain many widgets and views, all of which can be interacted with in the usual ways and controlled with code in their respective Java files. Fragments, like Activities, have a life cycle and associated callbacks like the `onCreate()` method, and here we used `onCreateView()`, which is called when an attempt to inflate the Fragment is made. It is important to note that, although Fragments can, and usually do, contain all kinds of code, they should not communicate directly with each other. This should be done from the Activity containing them.

You will have noticed, that the way we implemented our **OnClickListener** here differed from the way we did in the previous chapter, where we implemented it directly on the view to be clicked on. Here, the OnClickListener is part of the whole class. This method is generally preferred and despite the small amount of extra work required to calculate which widget was clicked on, it is usually a far tidier solution, especially for very complex and interactive layouts.

FragmentManager and **FragmentTransaction** are the tools we use to directly manipulate our Fragments. The first two lines, where these are defined, set the transaction in process, although no action is taken until `commit()` is called. We called `replace(layout, fragment)` to switch between our displays but we could also have used `add()` with the same arguments or `remove()` with only the Fragment.

The use of `addToBackStack()` is very important as, without it, a user pressing the back button on their device will be taken back to the previous Activity and not the previous Fragment, which, most of the time, will be what we want. Another interesting note is that the manager and transaction commands can be chained, and if we wanted we could replace all six lines with just this one:

```
getFragmentManager().beginTransaction()
    .transaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_
FADE)
    .replace(R.id.fragment_container, fragment)
    .addToBackStack(null)
    .commit();
```

The fade transition was not strictly necessary, as the system usually handles transitions between Activities and Fragments intelligently, but here there would have been none. There is a lot more that can be done with the FragmentManager and FragmentTransaction and the full documentation can be found at `developer.android.com/reference/android/app/FragmentManager` and `developer.android.com/reference/android/app/FragmentTransaction`. Next, we need to take a look at another way to apply Fragments in our apps.

Adding static Fragments

We cannot complete this introduction to Fragments without looking quickly at the other way that they can be implemented, as static layouts defined as `<fragment>` tags in XML files. Although these Fragments lack the flexibility of the dynamic sort we just encountered, they are nevertheless extremely useful, in particular when it comes to complex multi-pane apps where different Fragments perform very different functions. Not only does this help keep our code organized, it is also far less resource-hungry than a complicated network of nested ViewGroups.

To best see how this is done, start a project using the **Blank Activity with Fragment** template and take a look at the `activity_main.xml` file.

```
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/fragment"
  android:name="com.example.kyle.staticfragmentexample.
MainActivityFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:layout="@layout/fragment_main" />
```

This demonstrates how a static Fragment can be included in a standard layout file. It shows which layout is to be inflated inside it with `tools:layout` and which Java class controls it with `android:name`. Several Fragments can be combined this way, along with ViewGroups and Views to create complex Activities.

It is worth taking a look at both Java classes and the other XML file in this template, to see how the other components work. You will be familiar now with most of it, due to the work we have just completed. Do not close the project just for now, as we are going to use it to see how to add a menu.

Adding menus and toolbars

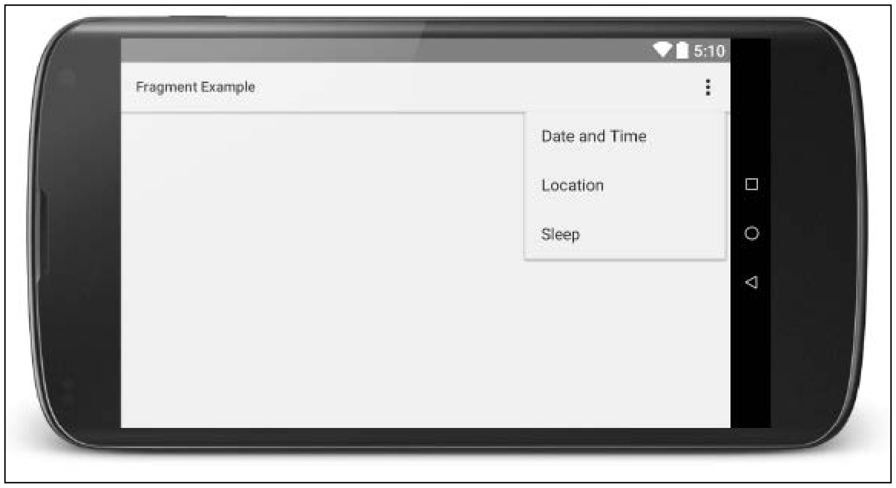
Nearly all mobile apps contain some form of global menu that provides access to functions required throughout the app. Menus can be opened in several ways on Android apps, but most commonly used is the **Options menu** which is accessed from the toolbar or action bar. Options menu items can also appear on the toolbar, as text or graphically.

First, we will add a basic drop-down toolbar menu to implement the functions currently provided by the two buttons. Open the project from the **Fragments** section and follow these steps:

1. Open the `res/menu/menu_main.xml` file.
2. Replace the existing `<item>` tag with these three:

```
<item
  android:id="@+id/menu_date"
  android:orderInCategory="100"
  android:showAsAction="never"
 android:title="Date and Time" />

<item
  android:id="@+id/menu_location"
  android:orderInCategory="100"
  android:showAsAction="never"
  android:title="Location" />

<item
  android:id="@+id/menu_sleep"
  android:orderInCategory="100"
  android:showAsAction="never"
  android:title="Sleep" />
```

3. Menus can be previewed in the same way as the layouts, by opening the preview pane:



4. Open the `MainActivity` file and locate the `onOptionsItemSelected()` method.
5. Rewrite it to look like this:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  Fragment fragment;

  int id = item.getItemId();
  switch (id) {
    case R.id.menu_date:
      startActivity(new Intent(android.provider.Settings.ACTION_
DATE_SETTINGS));
      break;
    case R.id.menu_location:
      startActivity(new Intent(android.provider.Settings.ACTION_
LOCALE_SETTINGS));
      break;
    case R.id.menu_sleep:
      startActivity(new Intent(android.provider.Settings.ACTION_
SOUND_SETTINGS));
      break;
  }

  return super.onOptionsItemSelected(item);
}
```

6. Run the app on a device and use the menu to open date, locale, and volume settings.

Adding menu items to the menu XML files

Each item requires a title and an ID. We can change the order the items appear in with the `orderInCategory` attribute with ascending integer values ordering items from top to bottom (and left to right on the toolbar). Menu items can be categorized and ordered separately by nesting them inside `<group>` tags.

Sub-menus can be created by embedding a `<menu>` tag, with its own items inside an `<item>` tag.

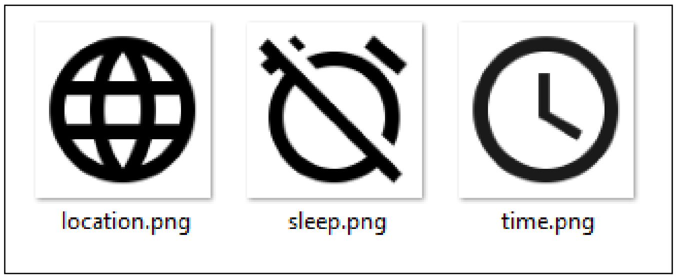
As we will see shortly, menu items can be displayed on the toolbar in the same way they were on the Action bar in versions of Android older that API 21. This can be done with the `showAsAction` attribute; if you still have the project open, it is worth running the app again with this value set to `always` and `ifRoom`, to see the effect this has. Doing this simply moves our menu options to the bar, but we can also use icons to represent our options; this, along with the new features of the Android 5 toolbar, is what we will explore next.

Configuring the toolbar

As already mentioned, Android 5 introduced a replacement for the Action bar at the top of many app screens: the toolbar. The toolbar does performs all the same functions as its predecessor, such as displaying menu options and other frequently performed actions, but is far more customizable. Most interestingly, the toolbar can now be placed anywhere on the screen.

In this next exercise, we will add our menus to the toolbar as icons; include navigation, logo, and titles on the toolbar; and then place it at the bottom of our screen.

Toolbar icons need to comply with some specific material design guidelines, a complete guide to which can be found at `http://www.google.com/design/spec/style/icons.html`. Basically, they need to be simple, single-color symbols on a transparent background. To begin with, you can download purpose-built system icons for specific screen densities from `http://www.google.com/design/icons/`. Below are the three that were download for this project, along with the names used in the next exercise.

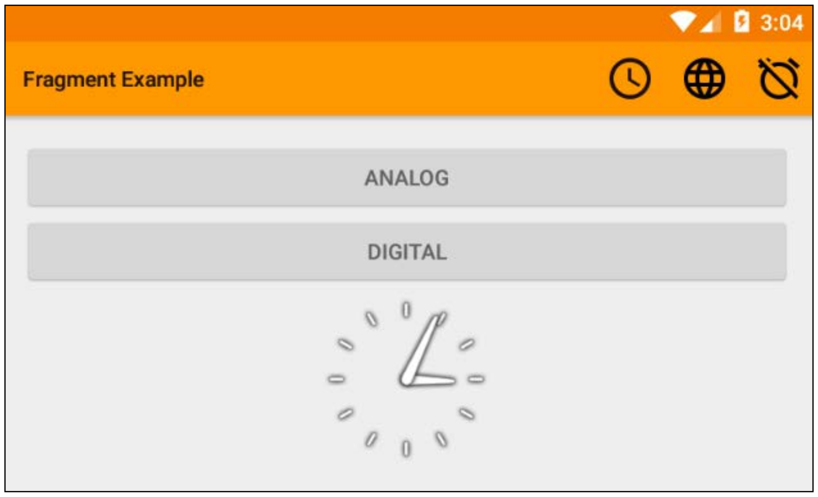


Download or find something similar of a pixel density that is suitable for the devices you are developing for. Then carry out the following steps:

1. Open the project we were just working on and copy your icons into the `drawable` folder.
2. Open the `menu_main.xml` file and add the following line into the `menu_date` item:

   ```
   android:icon="@drawable/time"
   ```

3. Do the same for the location and sleep icons in their respective item tags.
4. Change all three items' `android:showAsAction` attribute from `"never"` to `"ifRoom"`.
5. If you wish to see how your brand colors will appear against your action icons, apply a material theme as we have before, and run the app to see how your icons look.



6. Now open the `main_activity.xml` file.
7. Insert this `Toolbar` above the two buttons:

```
<Toolbar xmlns:android="http://schemas.android.com/apk/res/
android"
  android:id="@+id/toolbar"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:background="?android:attr/colorPrimary"
  android:elevation="4dp"
  android:minHeight="?android:attr/actionBarSize" />
```

8. Open the `res/values/styles/styles.xml (v21)` file and edit the `AppTheme` definition like this:

```
name="AppTheme"
  parent="android:Theme.Material.Light.NoActionBar"
```

9. You will need another image for the logo here. This does not have to be plain and simple but should not contain text, and it should be around 96 x 96 px, depending on your device's pixel density. If you want to save time, use one of the other images in the `setLogo()` command in step 11.

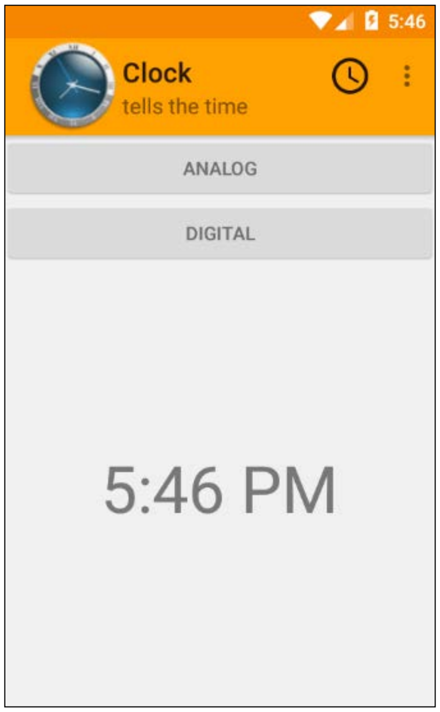
10. Open the `MainActivity` file and add this field:

```
Private Toolbar toolbar;
```

11. Then, add these lines to the `onCreate()` method:

```
toolbar = (Toolbar) findViewById(R.id.toolbar);
setActionBar(toolbar);
getActionBar().setTitle("Clock");
toolbar.setSubtitle("tells the time");
toolbar.setLogo(R.drawable.clock_logo);
```

12. In the `activity_main.xml` file, remove the padding from the root layout.
13. That's it. You can now run the app, with an output something like this:



Adding option menu items to the action bar is done simply by setting `showAsAction` to `ifRoom` or `always`, and we could have included both text and icon with `withText`. This provides a handy way to present options to the user, but it is transformed into a far more powerful tool once we can define it in a layout and refer to it from Java. This means we can place it anywhere on our screen and place anything inside it. We could add an `ImageView` or `Button`, simply by adding that element inside the `Toolbar` element in XML. Once we have a reference in Java, we can add click listeners or any other method, just as we can with other components.

We needed to remove the original action bar by changing the theme to `Material.Light.NoActionBar`, although we could have kept it and added a toolbar and of course, we can have two or more toolbars, perhaps even contained in a `Fragment` and replacing each other to create a more dynamic interface. We also had to remove padding from the parent layout, so as to have it plush to the edges like a traditional action bar and we set its elevation so that it looks like the action bar we are used to.

Summary

We have covered a lot in this chapter, starting with an introduction to the `CardView` widget and the new features incorporated in Android 5, such as the ability to elevate it and other views so that they appear to float above the screen. We saw how to add image files to a project and how we can optimize such images so as to efficiently match the screen density of the user's device. We saw how to include new Activities and how to use XML (as well as Java) to control how widgets and views behave when clicked on. The chapter also covered one of the many ways that Activities can communicate with one another.

We went on to explore another way to add flexibility to our apps with the `Fragment` classes which have much of the functionality of Activities but can be combined in one layout and treated like ViewGroups.

Finally, we investigated the relationship between **Options menu** and the action bar and saw how the toolbar introduced in Android 5 can be thought of as a part of our activity rather than a fixed widget stuck to the top of it.

Having grasped some of the more important fundamentals of Android programming, we can now go on to consider how to build more sophisticated apps that incorporate larger data sets, and how to utilize more of the technologies found in today's mobile devices.

4
Managing RecyclerViews and Their Data

In the previous chapter, we saw how to work with more than one Activity, but the data displayed in both was static. To progress, we need a way to apply a choice of data to a predefined layout and ideally present this as part of a list. Android 5 has introduced the RecyclerView-a more efficient and flexible version of the previously used ListView. To implement a RecyclerView, we will need a LayoutManager, an Adapter and some data to work with. In this chapter, we will start a new app that creates a list of CardViews, each displaying the relevant data and serving as a link to more detailed information.

In this chapter, we will cover the following topics:

- Creating a RecyclerView
- Designing a CardView Layout
- Including a LayoutManager
- Creating Data and an Adapter
- Adding a ViewHolder
- Responding to the RecyclerView selections
- Connecting a View to a web page

Creating a RecyclerView

The main Activity of our new project will consist entirely of a RecyclerView nested inside the root Layout, and is very quick and simple to construct. However, similar to the CardView, it is new to Android 5 and therefore, a part of the V7 Support Libraries and also, one of the Gradle build files will need to be modified, for it to work. The following steps demonstrate how to create a RecyclerView:

1. Start a new project in the Android studio. Give it an **Application name:** of `Ancient Britain`; check the Phone and Tablet checkbox and select **Blank Activity**.
2. Open the Gradle script file, `build.gradle` (Module app) and add the following two dependencies:

   ```
   dependencies {
     compile fileTree(dir: 'libs', include: ['*.jar'])
     compile 'com.android.support:cardview-v7:21.0.+'
     compile 'com.android.support:recyclerview-v7:21.0.+'
   }
   ```

3. Open the `activity_main.xml` file in `res/layout` and replace the `TextView` tag with the following code:

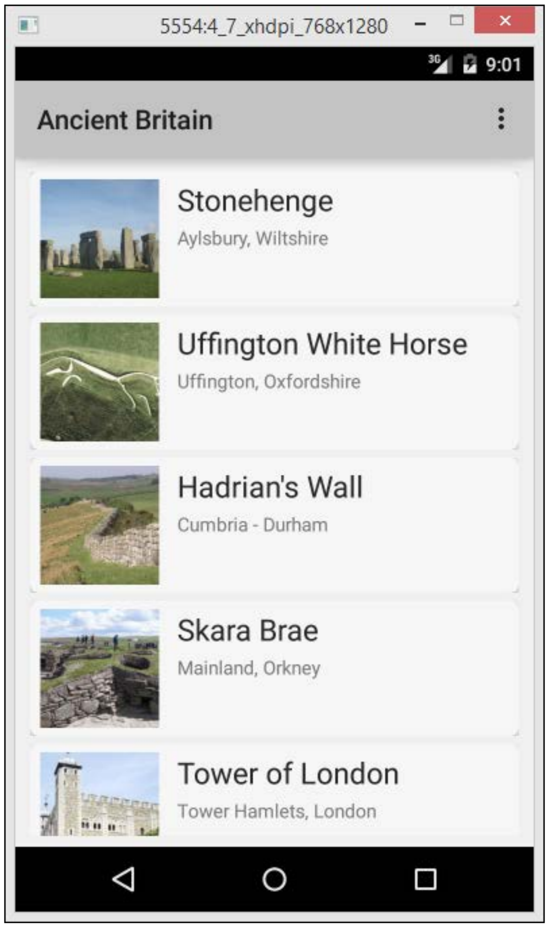
   ```
   <android.support.v7.widget.RecyclerView
     android:id="@+id/main_recycler_view"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     />
   ```

4. Create a color theme in the same way as we did in previous chapters.
5. Rebuild the project.

That's all there is to it! Our RecyclerView now fills the screen, apart from the margins, which are set automatically to Google's recommended design standards. Adding dependencies to the Gradle build scripts is something we are already familiar with and needs no explanation. The `text_secondary` color name will set smaller text to a dark gray.

Adding a CardView with layout

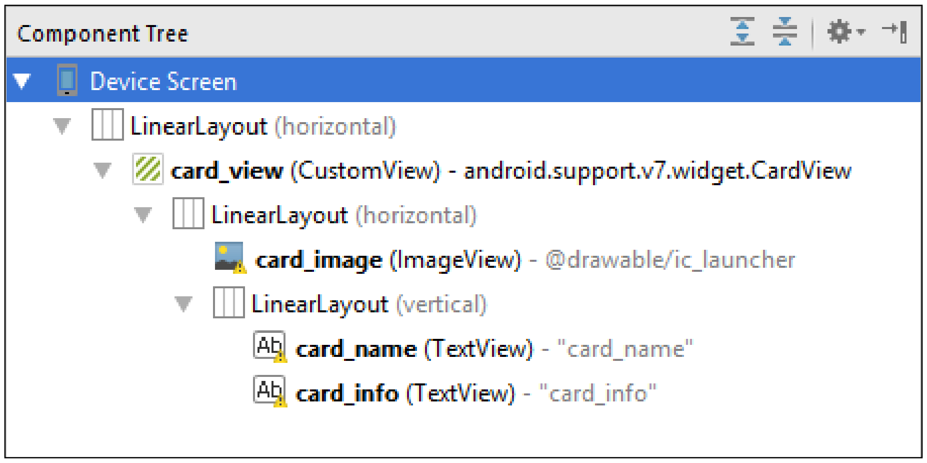
Our Recycler list will be comprised of CardViews, each conforming to the layout that we will design next. By the end of this section, our main Activity will look like this:



The first thing we need to do is design our CardView.

1. Right-click on **layout** in the project explorer and select **New** | **Layout resource file** from the menu.
2. Call it `card_main` and make sure its root element is a horizontal `LinearLayout`.

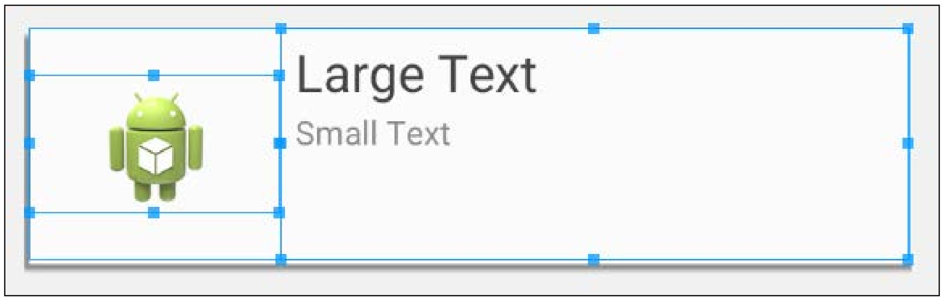
3. Design a card layout similar to those in the image above, with the following component structure and IDs.



```
Component Tree
▼ Device Screen
  ▼ LinearLayout (horizontal)
    ▼ card_view (CustomView) - android.support.v7.widget.CardView
      ▼ LinearLayout (horizontal)
          card_image (ImageView) - @drawable/ic_launcher
        ▼ LinearLayout (vertical)
            card_name (TextView) - "card_name"
            card_info (TextView) - "card_info"
```

The layout components can be moved and reordered directly from the Component Tree by dragging and dropping them.

4. Position the two inner `LinearLayout` and `ImageView` instances to match the following:



5. For the rest of this section, set the margins and/or padding to your own liking. Where `layout_width` and `layout_height` are not mentioned, use `wrap_content`.

6. Set the corner radius and elevation as you like, but make sure that the following properties are included in the `CardView` element:

```
android:id="@+id/card_view"
android:layout_width="match_parent"
android:layout_height="100dp"
```

7. Ensure that the `ImageView` has the following properties:

```
android:id="@+id/card_image"
android:layout_width="0dp"
android:layout_weight="2"
android:src="@drawable/ic_launcher
```

8. Set these properties in the inner (vertical) `LinearLayout`:

```
android:layout_width="0dp"
android:layout_height="match_parent"
android:layout_weight="5"
android:orientation="vertical"
```

9. Include these settings in the upper of the two `TextView` instances:

```
android:id="@+id/card_name"
android:textAppearance="?android:attr/textAppearanceLarge"
```

10. Finally, set these properties in the lower `TextView`:

```
android:id="@+id/card_info"
android:textAppearance="?android:attr/textAppearanceSmall"
```

Most of the properties set here are the ones we are familiar with: most importantly the `id` property, which allows us access through Java. However, one or two things may need explaining.

`LinearLayout` allows us to allocate proportional amounts of screen space to individual views, according to their `layout_weight` property. Instead of telling a view to take up as much space as it can (`match_parent`), or as little (`wrap_content`), using `layout_weight` allows us to allocate a certain proportion of the parent view group. Two views each with a weight of 1 will both take up 50% of the space, weights of 1 and 4 would take up 20% and 80% respectively. Here, we used weights of 2 and 5, meaning that 2/7 of the available space is allocated to the image and 5/7 to the vertical layout. We set the layout widths to `0dp` so as not to interfere with the weighting.

> The drawable `ic_launcher.png` was simply used as a placeholder to facilitate designing the layout; it will be replaced in the code later.

We used `textAppearanceSmall` and `textAppearanceLarge` to set our text sizes. This is often preferable to using specific dp amounts, as these settings automatically adjust to suit the size of the user's screen.

> Despite the obvious advantages of using weights in our layouts, it does need pointing out that this can have an adverse effect on performance, as the system has to recalculate its/the position more often.

Now that we have a `RecyclerView` and a `CardView` layout, we can go ahead and bring the two together with a `LayoutManager`.

Adding a LayoutManager

Views are positioned in the `RecyclerView` by a `RecyclerView.LayoutManager`, which in turn communicates with a `RecyclerView.Adapter`, and this binds our data to our views. First, let's set up a `LayoutManager`.

1. Open the `MainActivity.Java` file.
2. At the top of the class, declare the following fields:

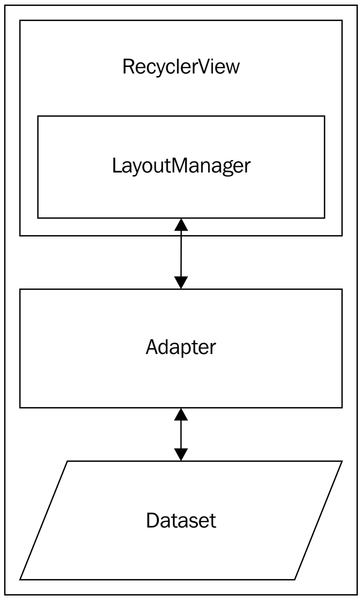
   ```
   private RecyclerView recyclerView;
   public static int currentItem;
   ```

3. Inside the `onCreate()` method, add these lines:

   ```
   recyclerView = (RecyclerView) findViewById(R.id.main_recycler_
   view);
   recyclerView.setHasFixedSize(true);

   RecyclerView.LayoutManager layoutManager = new
   LinearLayoutManager(this);
   recyclerView.setLayoutManager(layoutManager);
   ```

By now, the use of `findViewById()` should be familiar to us. The use of the `setFixedSize()` method is very useful. If you know that your list will remain the same length during runtime, then setting it to true will improve the performance of your app, as it cleverly recycles items when they are out of view, hence the name `RecyclerView`.

Giving the RecyclerView a LayoutManager is as simple as declaring a new `LinearLayoutManager`, passing it the current context and using `RecyclerView.setLayoutManager()` to set the connection. The `LayoutManager` can be thought of as belonging to the RecyclerView and it is the view's way of communicating with the data adapter, which in turn accesses a dataset, as can be seen in this diagram:



Before we move on to create our `Adapter` class, we will set up some data for it to work with.

Adding a dataset

The Android system utilizes SQLite for use with large and complex datasets, and we will return to this later in the book. For the purposes of this project, we will create the data arrays we need with Java. here's how this is done:

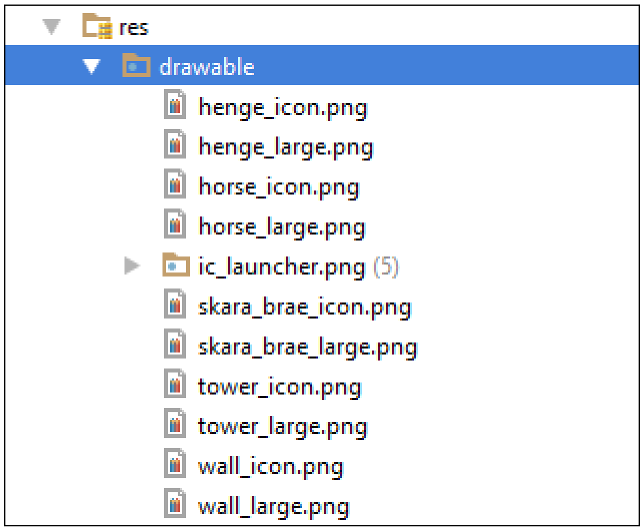
1. Create a new Java class by right-clicking on **MainActivity** on the project explorer and selecting **New | Java Class** from the menu. Call it `MainDataDef`.
2. Fill it out as follows:

```
public class MainDataDef {
  int image;
  String name;
  String info;

  public MainDataDef(int image, String name, String info) {
        this.image = image;
        this.name = name;
        this.info = info;
    }

    public int getImage() { return image; }

    public String getName() { return name; }

    public String getInfo() { return info; }
}
```

3. If you have downloaded the project files from the Packt website, then the dataset will contain ten records. However, the app will work just fine with any number and just the first five are included here. Create another Java class called `MainData` and complete it as follows:

```
public class MainData {

  static Integer[] imageArray = {R.drawable.henge_icon,
    R.drawable.horse_icon, R.drawable.wall_icon,
    R.drawable.skara_brae_icon, R.drawable.tower_icon};

  static String[] nameArray = {"Stonehenge",
    "Uffington White Horse", "Hadrian's Wall",
    "Skara Brae", "Tower of London"};

  static String[] infoArray = {"Aylsbury, Wiltshire",
    "Uffington, Oxfordshire", "Cumbria - Durham",
    "Mainland, Orkney", "Tower Hamlets, London"};

  static Integer[] detailImageArray =
    {R.drawable.henge_large, R.drawable.horse_large,
    R.drawable.wall_large, R.drawable.skara_brae_large,
    R.drawable.tower_large};

  static Integer[] detailTextArray =
    {R.string.detail_text_henge,
    R.string.detail_text_horse, R.string.detail_text_wall,
    R.string.detail_text_skara,
    R.string.detail_text_tower};

  static String[] detailWebLink =
    {"https://en.wikipedia.org/wiki/Stonehenge",
    "https://en.wikipedia.org/wiki/Uffington_White_Horse",
    "https://en.wikipedia.org/wiki/Hadrian%27s_Wall",
    "https://en.wikipedia.org/wiki/Skara_Brae",
    "https://en.wikipedia.org/wiki/Tower_of_London"};
}
```

4. If you have downloaded the project files, you will see that the `drawable` directory contains images: if not, you will need to add your own. If you are using the dataset seen above, then you will need ten images, with the names shown below. Make sure the `*_icon.png` files are roughly 160 x 160 px and the `*_large.png` files are about 640 x 480 px.



```
res
  drawable
    henge_icon.png
    henge_large.png
    horse_icon.png
    horse_large.png
    ic_launcher.png (5)
    skara_brae_icon.png
    skara_brae_large.png
    tower_icon.png
    tower_large.png
    wall_icon.png
    wall_large.png
```

5. The final part of the data consists of rather long strings. If you have downloaded the project files these can be found in the `strings.xml` resource file, if not, then you will need five strings of about 100 words a piece, with the following names:

```
<string name="detail_text_henge">
<string name="detail_text_horse">
<string name="detail_text_wall">
<string name="detail_text_skara">
<string name="detail_text_tower">
```

Putting our data arrays together was quite straightforward, but it is interesting to note how we used integers to refer to images and strings, using the automatically generated `R` class, which associates each individual resource with a static integer. This can be found by selecting **Packages** under **app** | **package name**. It cannot be edited, but it is helpful to see how it works.

We used the `strings.xml` file to store the long strings. This is not practical for lengthy text and usually we would store these resources as text files in the raw resource folder, and this is something we will cover later in the book.

Android TextViews are able to handle basic markup formatting tags, such as `<b></b>` and `<i></i>`. Below is a list of some of the formatting tags available:

```
<big></big>
<b></b>
<i></i>
<small></small>
<strike></strike>
<sup></sup>
<sub></sub>
<tt></tt>
<u></u>
```

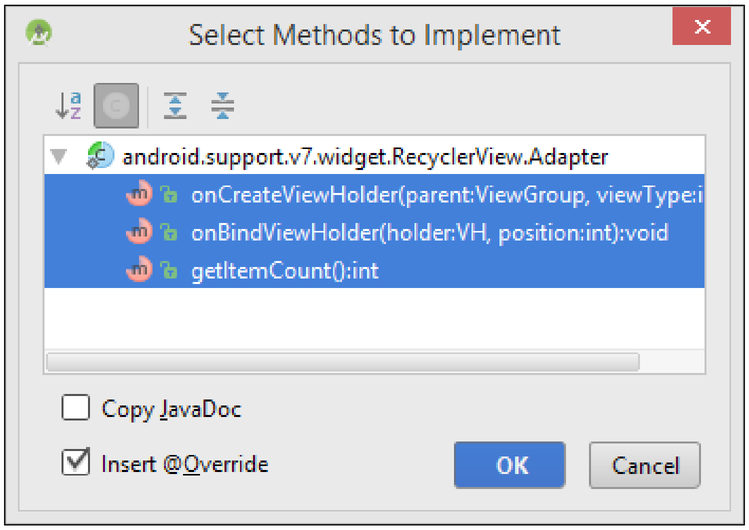
With our data in place, we can now get around to creating our data adapter.

Creating an Adapter

The `RecyclerView.Adapter` is responsible for binding our data to our views. We control how this happens through another `RecyclerView` sub-class, the `ViewHolder`, which we will create inside our Adapter. This can be achieved by following these steps:

1. Create a new Java class alongside the others in our project called `MainAdapter`.
2. Change the class declaration to this:
   ```
   public class MainAdapter extends
     RecyclerView.Adapter<MainAdapter.MainViewHolder> {
   ```
3. Directly under this, type:
   ```
   private ArrayList<MainDataDef> mainData;

   public MainAdapter(ArrayList<MainDataDef> a) {
     this.mainData = a;
   }

   public static class MainViewHolder extends
     RecyclerView.ViewHolder {
   ```
4. This last line will generate an error, indicated by a red underline. Place the cursor somewhere on the class declaration and press *Alt* + *Enter*.

5. Select **Implement Methods** from the drop-down list and then all three methods shown here:


Select Methods to Implement

android.support.v7.widget.RecyclerView.Adapter
onCreateViewHolder(parent:ViewGroup, viewType:i
onBindViewHolder(holder:VH, position:int):void
getItemCount():int

Copy JavaDoc
Insert @Override

6. Beneath this, add the following class:

```
public static class MainViewHolder extends RecyclerView.ViewHolder
{
  ImageView imageIcon;
  TextView textName;
  TextView textInfo;

  public MainViewHolder(View v) {
    super(v);
    this.imageIcon = (ImageView)
      v.findViewById(R.id.card_image);
    this.textName = (TextView)
      v.findViewById(R.id.card_name);
    this.textInfo = (TextView)
      v.findViewById(R.id.card_info);
  }
}
```

7. Complete the `onCreateViewHolder()` method like this:

```
@Override
public MainViewHolder onCreateViewHolder(ViewGroup parent,
  int viewType) {

  View v = LayoutInflater.from(parent.getContext())
```

```
            .inflate(R.layout.card_main, parent, false);

        v.setOnClickListener(MainActivity.mainOnClickListener);

        return new MainViewHolder(v);
    }
```

8. Then, the `onBindViewHolder()` method:

```
@Override
public void onBindViewHolder(final MainViewHolder holder,
  final int position) {
  ImageView imageIcon = holder.imageIcon;
  TextView textName = holder.textName;
  TextView textInfo = holder.textInfo;

  imageIcon.setImageResource(mainData.get(
    position).getImage());
  textName.setText(mainData.get(position).getName());
  textInfo.setText(mainData.get(position).getInfo());
}
```

9. And, the `getItemCount()` method:

```
@Override
public int getItemCount() {
  return mainData.length();
}
```

10. Open the `MainActivity` file.
11. At the bottom of the `onCreate()` method, add this code:

```
ArrayList<MainDataDef> mainData = new
  ArrayList<MainDataDef>();
for (int i = 0; i < MainData.nameArray.length; i++) {
  mainData.add(new MainDataDef(
    MainData.imageArray[i],
    MainData.nameArray[i],
    MainData.infoArray[i]
  ));
}

RecylerView.Adapter adapter = new MainAdapter(mainData);
recyclerView.setAdapter(adapter);
```

12. You can now test the project on an emulator or a handset.

Most of the work of the Adapter is carried out by the ViewHolder. This class is responsible, as its name suggests, for holding information about each of the views in our RecyclerView, including a view's metadata and its position in the list. We use the class definition and constructor to define three views that are associated with those in our card layout. The ViewHolder requires three callback methods, `onCreateViewHolder()`, which inflates the CardView and performs any other operations on it such as adding an `onClickListener`, `onBindViewHolder()`. This takes the ViewHolder's version of our card's inner views and connects them to our data, and `getItemCount()`, which returns the length of our list.

Finally, to connect the data to the RecyclerView, we build our ArrayList `mainData` and then set up a new `MainAdapter` from `mainData` and connect the RecyclerView to its Adapter with the `setAdapter()` method.

With our RecyclerView in place and connected to our dataset with an Adapter, we are ready to add a click listener and a second activity.

Responding to the RecyclerView selections

Having successfully produced a RecyclerView, and populated it with CardViews containing our data, we need to be able to select individual items and do something with them. Next, we will provide the RecyclerView with a **click listener** and add a new Activity that will present our records in greater detail on a separate screen.

Creating the OnClickListener

Before we create our new Activity, we need an `OnClickListener` that can tell which CardView was clicked on. Here's how it's done:

1. Open the `MainActivity` class.
2. Add the following class member:
   ```
   static View.OnClickListener mainOnClickListener;
   ```
3. Assign the listener in the `onCreate()` method like this:
   ```
   mainOnClickListener = new MainOnClickListener(this);
   ```
4. Create the following class:
   ```
   private class MainOnClickListener implements View.OnClickListener
   {
     private final Context context;
   ```

```
        private MainOnClickListener(Context c) {
          this.context = c;
        }

        @Override
        public void onClick(View v) {
          currentItem = recyclerView.getChildPosition(v);
          startActivity(new Intent(getApplicationContext(),
            DetailActivity.class));
        }
      }
```

This is very similar to the way that we implemented the `OnClickListener` before, only here, rather than assigning it to just a single view, we made it available to the whole class. This means that we must construct the listener in such a way as to pass the `Context` object of the calling view. The `onClick()` method is a good time to record which view was clicked on with the RecyclerView's `getChildPosition()` method. With this done, it is now a matter of creating the new Activity.

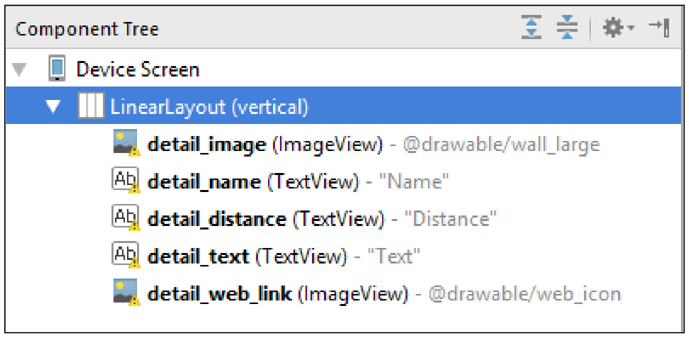
Adding the new Activity

The last method we entered will have generated an error message, as the class `DetailActivity` does not yet exist. The Activity will display greater detail about our ancient sites and provide a web link to each site's Wikipedia page.

Creating the portrait layout

As is often the case, there are two parts to creating an Android Activity, and before we can write the code, we need to define its layouts. To do this, follow these steps:

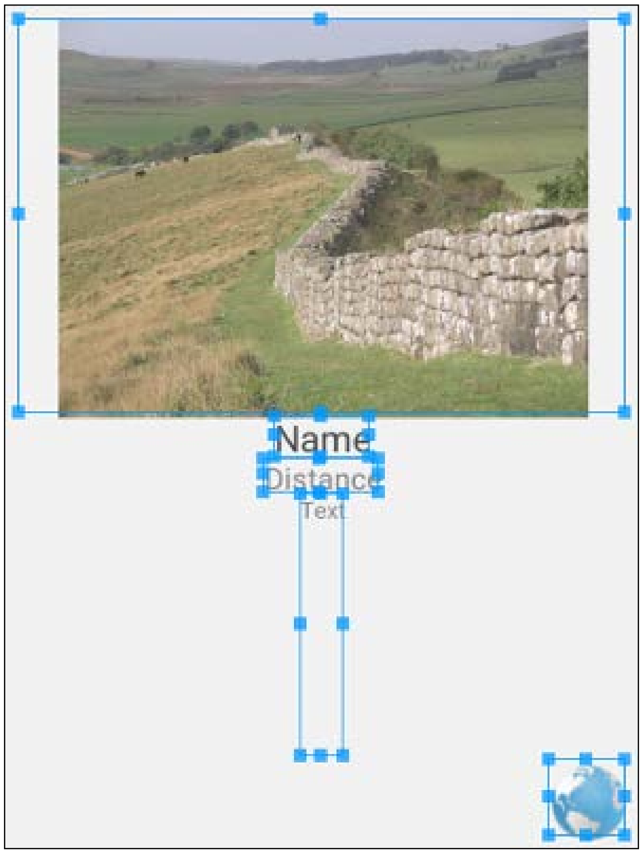
1. Create a new vertical `LinearLayout` XML file in the `layout` directory and call it `activity_detail`.
2. In the design mode, create a layout to match the following component tree and provide the views with the IDs and contents, as shown:



Component Tree

- Device Screen
  - LinearLayout (vertical)
    - **detail_image** (ImageView) - @drawable/wall_large
    - **detail_name** (TextView) - "Name"
    - **detail_distance** (TextView) - "Distance"
    - **detail_text** (TextView) - "Text"
    - **detail_web_link** (ImageView) - @drawable/web_icon

3. If you have not downloaded the project files, find or create a small image, suitable for use as a web icon, about 48 x 48 px. Call it `web_icon.png` and place it in your `drawable` directory.
4. Adjust the view's properties to match the structure shown here:



5. Set the following properties on the ImageView `detail_image`:

```
android:layout_height="0dp"
android:layout_gravity="center_horizontal"
android:layout_weight="3"
```

6. Add `android:textAppearance="?android:attr/textAppearanceLarge"` to the `detail_name` TextView:
7. The TextView `detail_distance` should have `android:textAppearance="?android:attr/textAppearanceMedium"` set.

8. The `detail_text` TextView needs these properties:

   ```
   android:layout_height="0dp"
   android:layout_weight="2"
   android:textAppearance="?android:attr/textAppearanceSmall"
   android:maxLines = "100"
   android:scrollbars = "vertical"
   ```

9. With the `detail_web_icon` ImageView, set `layout_gravity` to `"right"`.

There is nothing in this process that we have not encountered before, apart from the two properties in the `detail_text` TextView, `maxLines` and `scrollbars`, and these are self-explanatory. The text we have allocated to this view may well be too long for many smaller screens and so we have set this view up to be scrollable. There is still a line of code we need to add in Java to fully implement this, but we will come to that shortly.

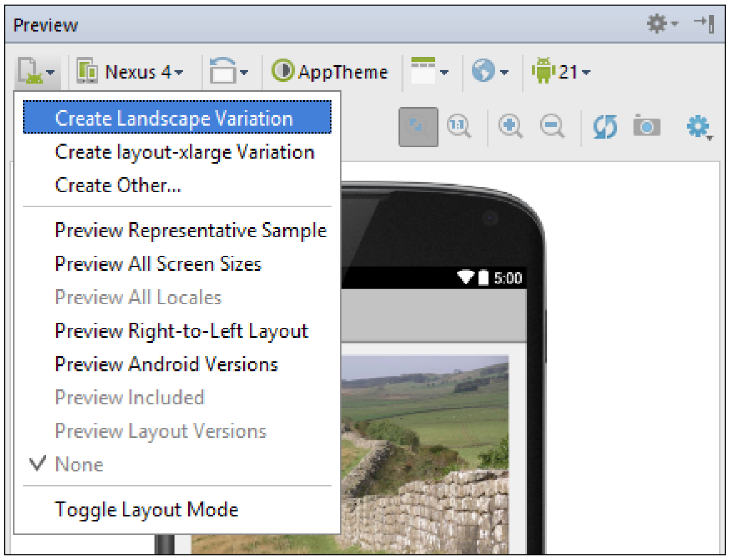
The use of `layout_weight` to define our layout means that, even when we rotate the screen, all of the views will remain in view. Nevertheless, if you test this on a device, you will see that this is not an attractive or space efficient layout. Next, we will redesign the landscape layout to more suitably fit the orientation.

Creating the landscape layout

The vertical linear layout we designed is not at all well suited to a landscape screen. To best fill the space, it is not enough to simply rearrange the views, we will need to change the root layout to a horizontal orientation and insert a vertical layout for the text and web icon. Follow these steps to create the layout:

1. If it is not open already, open the `activity_detail.xml` file in design mode or in text mode with the preview pane open.

2. Select **Create Landscape Variation** from the preview drop-down, as follows:



3. Open the `detail_activity.xml` (land) file in the text mode.
4. In `detail_image` ImageView, change the following lines:

```
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_gravity="center_vertical"
```

5. Insert the following vertical linear layout, so that it contains the three TextViews and the other ImageView:

```
<LinearLayout
  android:orientation="vertical"
  android:layout_width="0dp"
  android:layout_weight="3"
  android:layout_height="fill_parent">

  < TextView ... />
  < TextView ... />
  < TextView ... />
  < ImageView ... />

</LinearLayout>
```

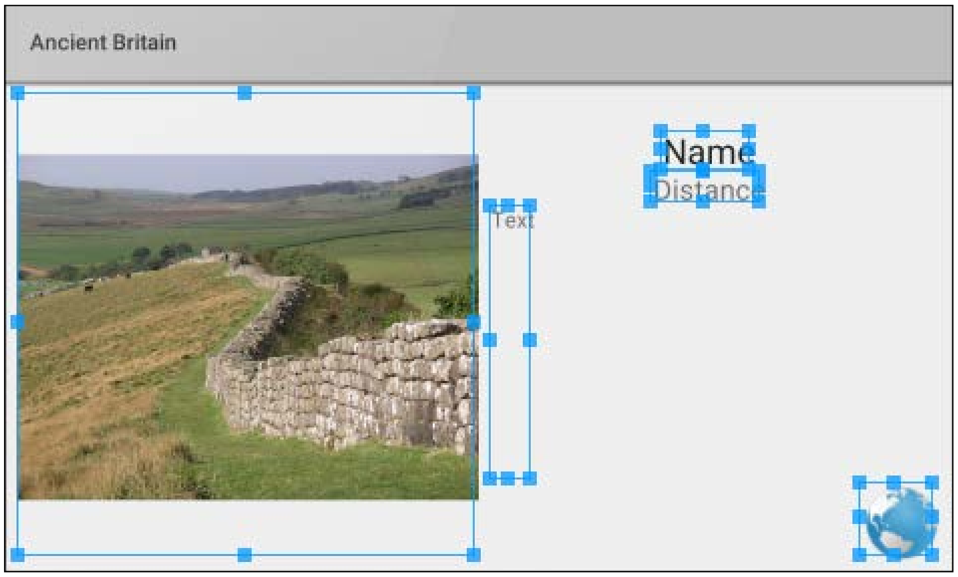
6. Change these lines in the `detail_text` TextView:

```
android:layout_height="0dp"
android:layout_weight="2"
android:layout_gravity="left"
```

7. And, add this one:

```
android:layout_marginLeft="8dp"
```

The landscape layout should now look like this:



There is nothing here that we have not encountered before. However, it is worth noting that, although it would have taken more time, we could have used a `RelativeLayout` for this purpose and had we done so, we would not have needed the inner vertical `LinearLayout`. ViewGroups, such as Layouts, require a considerable amount of memory, and although it does not matter for a small app such as this one, it is a good practice to keep the number of view containers to a minimum. Rebuilding `activity_detail.xml` (land) as a `RelativeLayout` is left as an exercise for the reader.

All that remains now is to create the Java Activity class `ActivityDetail`, which will select and display its content according to the card that was clicked on.

Connecting Views to web pages

The `ActivityDetail` class is not at all complicated, although we do get to add the Java component of the code, which will make our TextView scrollable and we also get to see how we can use an `Intent` object to launch the browser and open a specific web page. Follow these steps to complete the Activity:

1. From the project explorer, by right-clicking any of the classes we have already constructed, create a new Activity by selecting **New** | **Activity** | **Blank Activity**, calling the file `DetailActivity` and accepting the other values the dialog suggests.
2. Open the `DetailActivity` Java code.
3. Change the line `setContentView(R.layout.activity_detail2);` in `onCreate()` to the following:

   ```
   setContentView(R.layout.activity_detail);
   ```

4. When you created the Activity, the IDE will have created a matching layout file `activity_detail2`. Delete this file by selecting it in the project explorer and pressing *Delete*.
5. Add the following class wide fields:

   ```
   private ImageView detailImage;
   private ArrayList<MainDataDef> detailData;
   ```

6. Assign them like this:

   ```
   detailImage = (ImageView) findViewById(R.id.detail_image);
   TextView detailName = (TextView)
     findViewById(R.id.detail_name);
   TextView detailDistance = (TextView)
     findViewById(R.id.detail_distance);

   TextView detailText = (TextView)
     findViewById(R.id.detail_text);
   detailText.setMovementMethod(new
     ScrollingMovementMethod());

   ImageView detailWebLink = (ImageView)
     findViewById(R.id.detail_web_link);
   ```

7. Still in `onCreate()`, add these lines to link our views to our data:

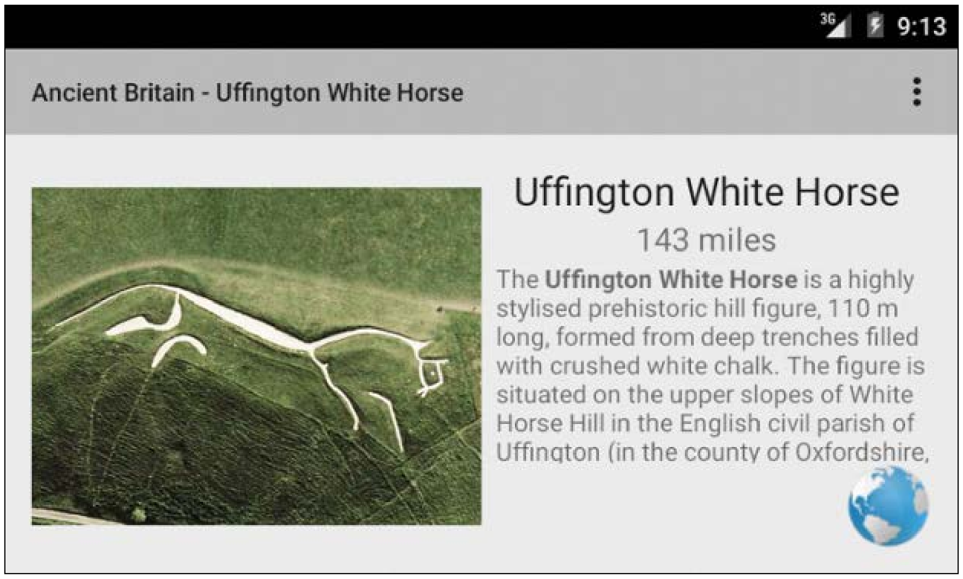
   ```
   int i = MainActivity.currentItem;
   Random n = new Random();
   int m = n.nextInt((600 - 20) + 1) + 20;
   ```

```
setTitle(getString(R.string.app_name) + " - " + MainData.
nameArray[i]);

detailImage.setImageResource(MainData.detailImageArray[i]);
detailName.setText(MainData.nameArray[i]);
detailDistance.setText(String.valueOf(m) + " miles");
detailText.setText(MainData.detailTextArray[i]);
```

8. Beneath that, add these lines to program the web link:

```
detailWebLink.setOnClickListener(new View.OnClickListener() {

  @Override
  public void onClick(View v) {
    Intent intent = new Intent();
    intent.setAction(Intent.ACTION_VIEW);
    intent.addCategory(Intent.CATEGORY_BROWSABLE);
    intent.setData(Uri.parse(
      MainData.detailWebLink[MainActivity.currentItem]));
    startActivity(intent);
  }
});
```

9. Run the app on an emulator or a handset. Hopefully, the second activity will look like this:



Uffington White Horse

143 miles

The **Uffington White Horse** is a highly stylised prehistoric hill figure, 110 m long, formed from deep trenches filled with crushed white chalk. The figure is situated on the upper slopes of White Horse Hill in the English civil parish of Uffington (in the county of Oxfordshire, The Activity wizard automatically created a layout file for us, even though we had already created one, which is why we had to edit the `ContentView` of the Activity and delete the unwanted XML file. We could have of course constructed our class from scratch and avoided doing this, but the Activity wizard builds a time saving class template for us, including all the required members.

The naming and assigning of our views requires no explanation. However, this section of code also provides the final part of our scrollable TextView by calling its `setMovementMethod(new ScrollingMovementMethod())`.

We used the public field `currentItem` to access our data, which we kept in the same array list for convenience. We filled the distance field with a random number for now, but we will return to make this function operate correctly when we get to geo-locations later in the book.

Linking views to web pages is new and requires explanation. This is the only view in the layout that is clickable, so it is a simple matter to provide it with its own `OnClickListener`. Here, we get to see the `Intent` object in more detail and also just how useful it can be. The `setAction()` method tells the Intent which action to perform, in the form of a static, final String which here is `ACTION_VIEW`, the most widely used **Intent action**, and it tells the Intent to display the data in the most appropriate way. This means that, more often than not, the system will select the correct way to display the data, such as displaying images in an ImageView, contacts in the contacts app, and web pages in the default browser.

Note that, there is a corresponding `Intent.getAction()` method for when we need to know what action has been set to an Intent.

The `addCategory(Intent.CATEGORY_BROWSABLE)` call is not strictly necessary and the app will run fine without it in most circumstances. **Intent categories** cause the Intent to only consider apps (or more accurately, activities), that are labeled with that category when resolving the action. To place an app in one or more categories, add them to the `<intent-filter>` tab in the manifest file like so: `<category android:name="android.intent.category.CATEGORY"/>`. The `Intent` object defines a wide selection of category constants, such as, `APP_MUSIC`, `APP_MESSAGING`, and `APP_CALENDAR`.

With the second Activity and its layout files in place, we can now display detailed information about each of our ancient sites and provide a link to the web page with further information. With this done, we have completed the exercise.

Summary

In this chapter, we have constructed a small but complete app. The structure is by no means dictated by the subject matter and is such that it could be applied to any number of apps. We took advantage of both of Android's latest widgets and produced a RecyclerView list of CardViews, which displayed a simple dataset. In doing this, we saw how to connect our data to our views with an Adapter and how this in turn uses a ViewHolder to maintain our individual views and layouts. Adding an `OnClickListener` allowed the user to select data items and navigate to a new screen with further information and a link to a website.

Next, we will delve deeper into the RecyclerView and see how to move items up and down the list. To do this, we will first have to learn how to detect a swipe gesture from the user, using an `OnTouchListener`.

5 Detecting Touchscreen Gestures

Until now, the apps we created have used `OnClickListeners` to detect user input. However, Android handsets are capable of handling sophisticated touchscreen gestures. These inputs are picked up with an `OnTouchListener` and then managed with a `GestureDetector`. These detectors and their own listeners are capable of recognizing several of the simplest and most commonly used gestures, such as **long presses**, **double-taps**, and **flings**. At the heart of all touchscreen events is the `MotionEvent` class, which handles the individual elements of a gesture, such as when and where a finger is placed or removed from the screen or view. This class provides numerous classes for querying these events and, thus, for constructing custom gestures of our own.

To see how to implement gestures in our Ancient Britain app, we will add a feature that allows the user to view a small gallery of images within the same `ImageView` by swiping the view.

In this chapter, we will learn how to:

- Add a `GestureDetector` to a view
- Add an `OnTouchListener` and an `OnGestureListener`
- Detect and refine fling gestures
- Use the DDMS Logcat to observe the `MotionEvent` class
- Edit the Logcat filter configuration
- Simplify code with a `SimpleOnGestureListener`
- Add a `GestureDetector` to an Activity
- Edit the Manifest to control launch behavior

- Hide UI elements
- Create a splash screen
- Lock screen orientation

Adding a GestureDetector to a view

Together, `view.GestureDetector` and `view.View.OnTouchListener` are all that are required to provide our `ImageView` with gesture functionality. The listener contains an `onTouch()` callback that relays each `MotionEvent` to the detector. We are going to program the large `ImageView` so that it can display a small gallery of related pictures that can be accessed by swiping left or right on the image.

There are two steps to this task as, before we implement our gesture detector, we need to provide the data for it to work on.

Adding the gallery data

As this app is for demonstration and learning purposes, and so we can progress as quickly as possible, we will only provide extra images for one or two of the ancient sites in the project. Here is how it's done:

1. Open the Ancient Britain project.
2. Open the `MainData.java` file.
3. Add the following arrays:

   ```
   static Integer[] hengeArray = {R.drawable.henge_large,
     R.drawable.henge_2, R.drawable.henge_3,
     R.drawable.henge_4};
   static Integer[] horseArray = {};
   static Integer[] wallArray = {R.drawable.wall_large,
     R.drawable.wall_2};
   static Integer[] skaraArray = {};
   static Integer[] towerArray = {};

   static Integer[][] galleryArray = {hengeArray, horseArray,
     wallArray, skaraArray, towerArray};
   ```

4. Either download the project files from the Packt website or find four of your own images (around 640 x 480 px). Name them `henge_2`, `henge_3`, `henge_4`, and `wall_2` and place them in your `res/drawable` directory.

This is all very straightforward, and the code that will accompany it allows you to have individual arrays of any length. This is all we need to add to our gallery data. Now, we need to code our `GestureDetector` and `OnTouchListener`.

Adding the GestureDetector

Along with the `OnTouchListener` that we will define for our `ImageView`, the GestureDetector has its own listeners. Here we will use `GestureDetector.OnGestureListener` to detect a fling gesture and collect the `MotionEvent` that describe it.

Follow these steps to program your `ImageView` to respond to fling gestures:

1. Open the `DetailActivity.java` file.
2. Declare the following class fields:

```
private static final int MIN_DISTANCE = 150;
private static final int OFF_PATH = 100;
private static final int VELOCITY_THRESHOLD = 75;
private GestureDetector detector;
View.OnTouchListener listener;
private int ImageIndex;
```

3. In the `onCreate()` method assigns both the `detector` and `listener` like this:

```
detector = new GestureDetector(this, new
GalleryGestureDetector());
listener = new View.OnTouchListener() {

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        return detector.onTouchEvent(event);
    }
};
```

4. Beneath this, add the following line:

```
ImageIndex = 0;
```

5. Beneath the line `detailImage = (ImageView) findViewById(R.id.detail_image);`, add the following line:

```
detailImage.setOnTouchListener(listener);
```

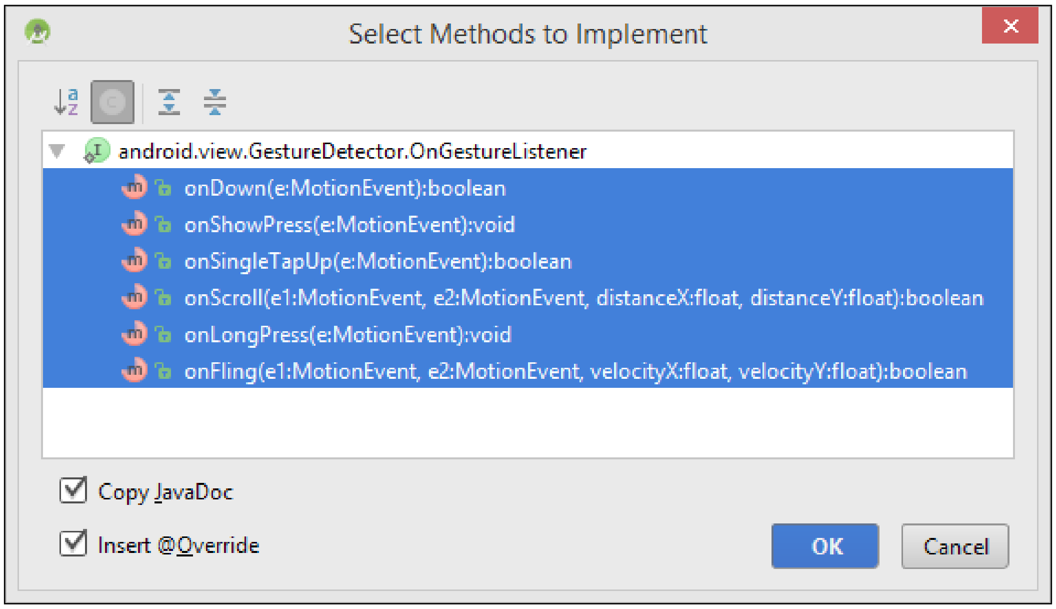
6. Create the following inner class:

```
class GalleryGestureDetector implements
  GestureDetector.OnGestureListener { }
```

7. Before dealing with the errors this generates, add the following field to the class:

```
private int item;
{
    item = MainActivity.currentItem;
}
```

8. Click anywhere on the line registering the error and press *Alt* + *Enter*. Then select **Implement Methods**, making sure that you have the **Copy JavaDoc** and **Insert @Override** boxes checked.


Select Methods to Implement

```
android.view.GestureDetector.OnGestureListener
    onDown(e:MotionEvent):boolean
    onShowPress(e:MotionEvent):void
    onSingleTapUp(e:MotionEvent):boolean
    onScroll(e1:MotionEvent, e2:MotionEvent, distanceX:float, distanceY:float):boolean
    onLongPress(e:MotionEvent):void
    onFling(e1:MotionEvent, e2:MotionEvent, velocityX:float, velocityY:float):boolean
```

☑ Copy JavaDoc

☑ Insert @Override

9. Complete the `onDown()` method like this:

```
@Override
public boolean onDown(MotionEvent e) {
    return true;
}
```

10. Fill in the `onShowPress()` method:

```
@Override
public void onShowPress(MotionEvent e) {
    detailImage.setElevation(4);
}
```

11. Then fill in the `onFling()` method:

```
@Override
public boolean onFling(MotionEvent event1, MotionEvent
  event2, float velocityX, float velocityY) {
  if (Math.abs(event1.getY() - event2.getY()) > OFF_PATH)
    return false;

  if (MainData.galleryArray[item].length != 0) {
    // Swipe left
    if (event1.getX() - event2.getX() > MIN_DISTANCE &&
      Math.abs(velocityX) > VELOCITY_THRESHOLD) {
        ImageIndex++;
        if (ImageIndex ==
          MainData.galleryArray[item].length)
          ImageIndex = 0;
          detailImage.setImageResource(MainData
           .galleryArray[item][ImageIndex]);
    } else {
      // Swipe right
      if (event2.getX() - event1.getX() >
        MIN_DISTANCE && Math.abs(velocityX) >
        VELOCITY_THRESHOLD) {
        ImageIndex--;
        if (ImageIndex < 0) ImageIndex =
          MainData.galleryArray[item].length - 1;
        detailImage.setImageResource(MainData
          .galleryArray[item][ImageIndex]);
    }
  }
  }
  detailImage.setElevation(0);
    return true;
}
```

12. Test the project on an emulator or handset.

The process of gesture detection in the preceding code begins when the `OnTouchListener` listener's `onTouch()` method is called. It then passes that `MotionEvent` to our gesture detector class, `GalleryGestureDetector`, which monitors motion events, sometimes stringing them together and timing them until one of the recognized gestures is detected. At this point, we can enter our own code to control how our app responds as we did here with the `onDown()`, `onShowPress()`, and `onFling()` callbacks. It is worth taking a quick look at these methods in turn.

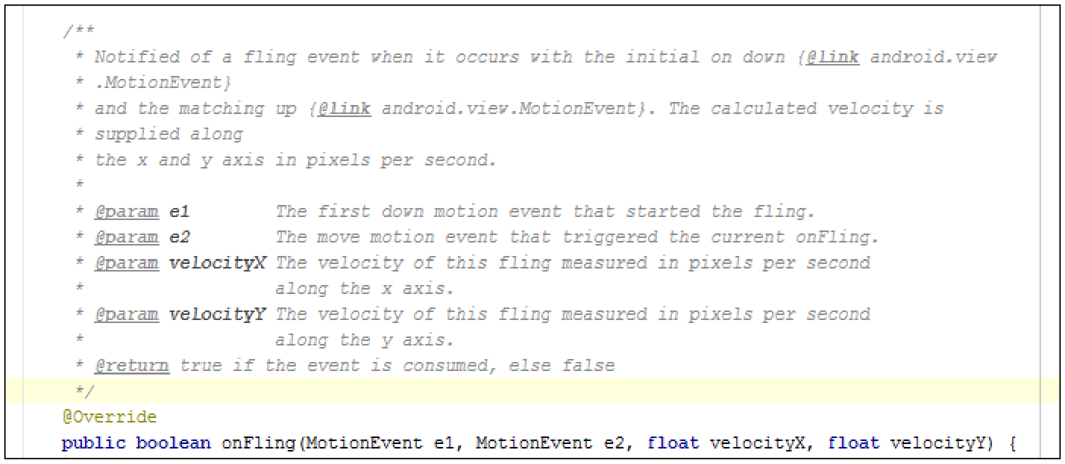
It may seem, at the first glance, that the `onDown()` method is redundant; after all, it's the fling gesture that we are trying to catch. In fact, overriding the `onDown()` method and returning `true` from it is essential in all gesture detections as all the gestures begin with an `onDown()` event.

The purpose of the `onShowPress()` method may also appear unclear as it seems to do a little more than `onDown()`. As the **JavaDoc** states, this method is handy for adding some form of feedback to the user, acknowledging that their touch has been received. The Material Design guidelines strongly recommend such feedback and here we have raised the view's elevation slightly.

Without including our own code, the `onFling()` method will recognize almost any movement across the bounding view that ends in the user's finger being raised, regardless of direction or speed. We do not want very small or very slow motions to result in action; furthermore, we want to be able to differentiate between vertical and horizontal movement as well as left and right swipes. The `MIN_DISTANCE` and `OFF_PATH` constants are in pixels and `VELOCITY_THRESHOLD` is in pixels per second. These values will need tweaking according to the target device and personal preference. The first MotionEvent argument in `onFling()` refers to the preceding `onDown()` event and, like any `MotionEvent`, its coordinates are available through its `getX()` and `getY()` methods.

> The MotionEvent class contains dozens of useful classes for querying various event properties—for example, `getDownTime()`, which returns the time in milliseconds since the current `onDown()` event.

In this example, we used `GestureDetector.OnGestureListener` to capture our gesture. However, the GestureDetector has three such nested classes, the other two being `SimpleOnGestureListener` and `OnDoubleTapListener`. `SimpleOnGestureListener` provides a more convenient way to detect gestures as we only need to implement those methods that relate to the gestures we are interested in capturing. We will shortly edit our Activity so that it implements the `SimpleOnGestureListener` instead, allowing us to tidy our code and remove the four callbacks that we do not need. The reason for taking this detour, rather than applying the simple listener to begin with, was to get to see all of the gestures available to us through a gesture listener and demonstrate how useful JavaDoc comments can be, particularly if we are new to the framework. For example, take a look at the following screenshot:



```
/**
 * Notified of a fling event when it occurs with the initial on down {@link android.view
 * .MotionEvent}
 * and the matching up {@link android.view.MotionEvent}. The calculated velocity is
 * supplied along
 * the x and y axis in pixels per second.
 *
 * @param e1        The first down motion event that started the fling.
 * @param e2        The move motion event that triggered the current onFling.
 * @param velocityX The velocity of this fling measured in pixels per second
 *                  along the x axis.
 * @param velocityY The velocity of this fling measured in pixels per second
 *                  along the y axis.
 * @return true if the event is consumed, else false
 */
@Override
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
```

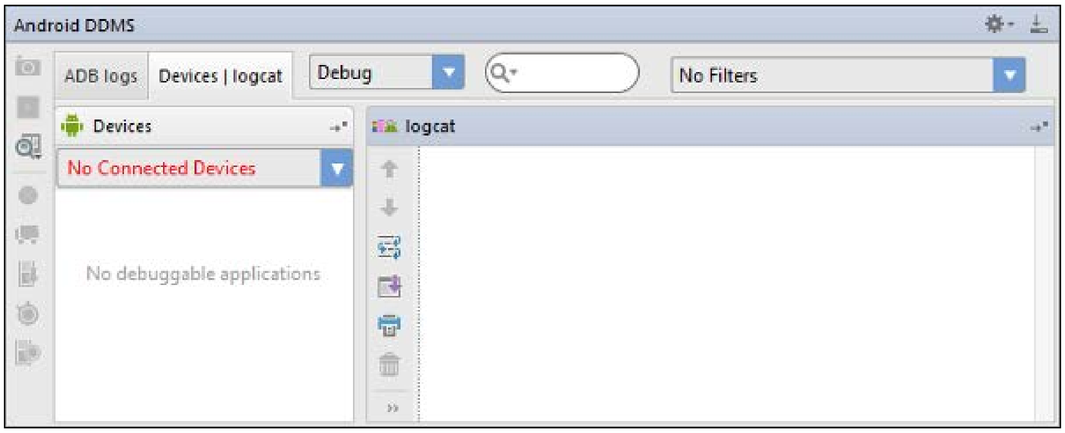
Another very handy tool is the **Dalvik Debug Monitor Server** (**DDMS**), which allows us to see what is going on inside our apps while they are running. The workings of our gesture listener are a good place to do this as most of its methods operate invisibly.

Viewing gesture activity with DDMS

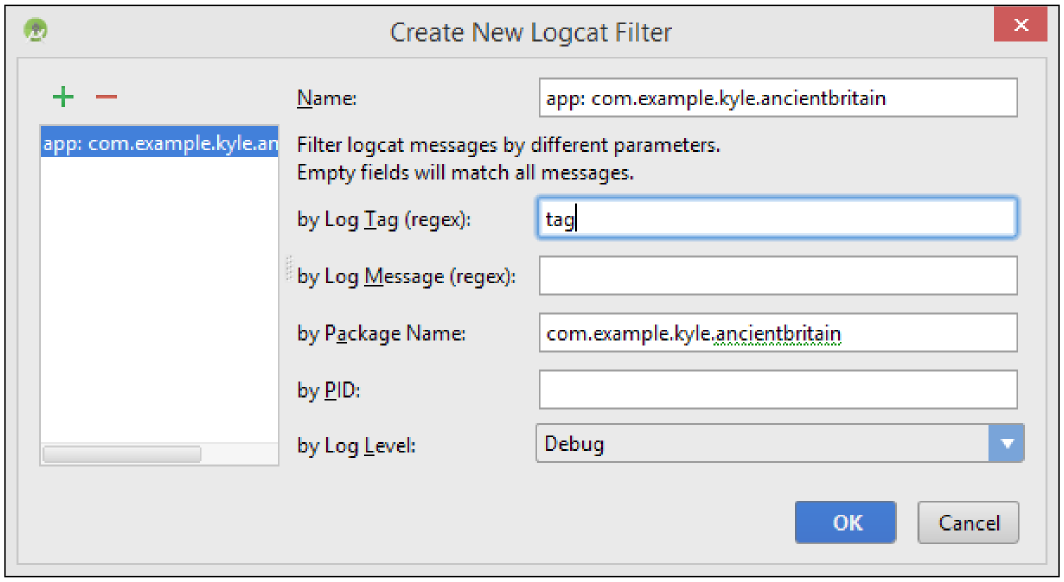
To view the workings of our `OnGestureListener` with DDMS, we need to first create a tag to identify our messages and then a filter to view them. The following steps demonstrate how to do this:

1. Open the `DetailActivity.java` file.
2. Declare the following constant:
   ```
   private static final String DEBUG_TAG = "tag";
   ```
3. Add the following line inside the `onDown()` method:
   ```
   Log.d(DEBUG_TAG, "onDown");
   ```
4. Add the line `Log.d(DEBUG_TAG, "onShowPress");` to the `onShowPress()` method and do the same for each of our `OnGestureDetector` methods.
5. Add the following lines to the appropriate clauses in `onFling()`:
   ```
   Log.d(DEBUG_TAG, "left");
   Log.d(DEBUG_TAG, "right");
   ```

Open the **Android DDMS** pane from the **Android** tab at the bottom of the window or by pressing *Alt* + *6*.



6. If **logcat** is not visible, it can be opened with the icon to the right of the top-right drop-down menu.
7. Click on this drop-down menu and select **Edit Filter Configuration**.
8. Complete the dialog as shown in the following screenshot:



Name: app: com.example.kyle.ancientbritain

Filter logcat messages by different parameters.
Empty fields will match all messages.

by Log Tag (regex): tag

by Log Message (regex):

by Package Name: com.example.kyle.ancientbritain

by PID:

by Log Level: Debug

9. You can now run the project on a handset or emulator and view, in the Logcat, which gestures are being triggered and how. Your output should resemble the one here:

```
02-17 14:39:00.990    1430-
  1430/com.example.kyle.ancientbritain D/tag: onDown
02-17 14:39:01.039    1430-
  1430/com.example.kyle.ancientbritain D/tag: onSingleTapUp
02-17 14:39:03.503    1430-
  1430/com.example.kyle.ancientbritain D/tag: onDown
02-17 14:39:03.601    1430-
  1430/com.example.kyle.ancientbritain D/tag: onShowPress
02-17 14:39:04.101    1430-
  1430/com.example.kyle.ancientbritain D/tag: onLongPress
02-17 14:39:10.484    1430-
  1430/com.example.kyle.ancientbritain D/tag: onDown
02-17 14:39:10.541    1430-
  1430/com.example.kyle.ancientbritain D/tag: onScroll
02-17 14:39:11.091    1430-
  1430/com.example.kyle.ancientbritain D/tag: onScroll
02-17 14:39:11.232    1430-
  1430/com.example.kyle.ancientbritain D/tag: onFling
02-17 14:39:11.680    1430-
  1430/com.example.kyle.ancientbritain D/tag: right
```

DDMS is an invaluable tool when it comes to debugging our apps and seeing what is going on beneath the hood. Once a **Log Tag** has been defined in the code, we can then create a **filter** for it so that we see only the messages we are interested in. The `Log` class contains several methods to report information based on its level of importance. We used `Log.d`, which stands for *debug*. All these methods work with the same two parameters: `Log.[method](String tag, String message)`. The full list of these methods is as follows:

- `Log.v`: Verbose
- `Log.d`: Debug
- `Log.i`: Information
- `Log.w`: Warning
- `Log.e`: Error
- `Log.wtf`: Unexpected error

It is worth noting that most debug messages will be ignored during the packaging for distribution except for the verbose messages; thus, it is essential to remove them before your final build.

Having seen a little more of the inner workings of our gesture detector and listener, we can now strip our code of unused methods by implementing `GestureDetector.SimpleOnGestureListener`.

Implementing a SimpleOnGestureListener

It is very simple to convert our gesture detector from one class of listener to another. All we need to do is change the class declaration and delete the unwanted methods. To do this, perform the following steps:

1. Open the `DetailActivity` file.
2. Change the class declaration for our gesture detector class to the following:
   ```
   class GalleryGestureDetector extends
     GestureDetector.SimpleOnGestureListener {
   ```
3. Delete the `onShowPress()`, `onSingleTapUp()`, `onScroll()`, and `onLongPress()` methods.

This is all you need to do to switch to the `SimpleOnGestureListener`. We have now successfully constructed and edited a gesture detector to allow the user to browse a series of images.

You will have noticed that there is no `onDoubleTap()` method in the gesture listener. Double-taps can, in fact, be handled with the third `GestureDetector` listener, `OnDoubleTapListener`, which operates in a very similar way to the other two. However, Google, in its UI guidelines, recommends that a long press should be used instead, whenever possible.

Before moving on to multitouch events, we will take a look at how to attach a `GestureDetector` listener to an entire Activity by adding a splash screen to our project. In the process, we will also see how to create a Full-Screen Activity and how to edit the `Maniftest` file so that our app launches with the splash screen.

Adding a GestureDetector to an Activity

The method we have employed so far allows us to attach a `GestureDetector` listener to any view or views and this, of course, applies to `ViewGroups` such as `Layouts`. There are times when we may want to detect gestures to be applied to the whole screen. For this purpose, we will create a splash screen that can be dismissed with a long press.

There are two things we need to do before implementing the gesture detector: creating a layout and editing the Manifest file so that the app launches with our splash screen.

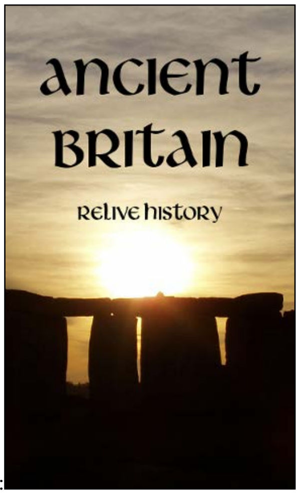
Designing the splash screen layout

The main difference between processing gestures for a whole Activity and an individual widget, is that we do not need an `OnTouchListener` as we can override the Activity's own `onTouchEvent()`. Here is how it is done:

1. Create a new Blank Activity from the Project Explorer context menu called `SplashActivity.java`.
2. The Activity wizard should have created an associated XML layout called `activity_splash.xml`. Open this and view it using the **Text** tab.
3. Remove all the padding properties from the root layout so that it looks similar to this:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.kyle.ancientbritain
      .SplashActivity">
```

   Here we will need an image to act as the background for our splash screen. If you have not downloaded the project files from the Packt website, find an image, roughly of the size and aspect of your target device's screen, upload it to the project drawable folder, and call it `splash`. The file I used is 480 x 800 px.



4. Remove the `TextView` that the wizard placed inside the layout and replace it with this `ImageView`:

```
<ImageView
    android:id="@+id/splash_image"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/splash"/>
```

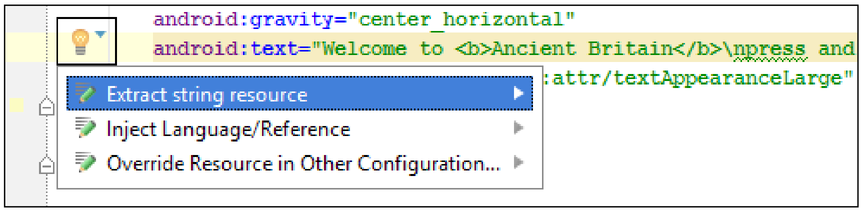
5. Create a `TextView` beneath this, such as the following:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_centerHorizontal="true"
    android:layout_marginBottom="40dp"
    android:gravity="center_horizontal"
    android:textAppearance="?android:attr/
      textAppearanceLarge"
    android:textColor="#fffcfcbd"/>
```

6. Add the following text property:

```
android:text="Welcome to <b>Ancient Britain</b>\npress and hold\
nanywhere on the screen\nto start"
```

To save time adding string resources to the `strings.xml` file, enter a hardcoded string such as the preceding one and heed the warning from the editor to have the string extracted for you like this:



```
android:gravity="center_horizontal"
android:text="Welcome to <b>Ancient Britain</b>\npress and
:attr/textAppearanceLarge"
```

Extract string resource
Inject Language/Reference
Override Resource in Other Configuration...

There is nothing in this layout that we have not encountered before. We removed all the padding so that our splash image will fill the layout; however, you will see from the preview that this does not appear to be the case. We will deal with this next in our Java code, but we need to edit our **Manifest** first so that the app gets launched with our `SplashActivity`.

Editing the Manifest

It is very simple to configure the `AndroidManifest` file so that an app will get launched with whichever Activity we choose; the way it does so is with an intent. While we are editing the Manifest, we will also configure the display to fill the screen. Simply follow these steps:

1. Open the `res/values-v21/styles.xml` file and add the following style:

   ```
   <style name="SplashTheme" parent="android:Theme.Material.
   NoActionBar.Fullscreen">
   </style>
   ```

2. Open the `AndroidManifest.xml` file.
3. Cut-and-paste the `<intent-filter>` element from `MainActivity` to `SplashActivity`.
4. Include the following properties so that the entire `<activity>` node looks similar to this:

   ```
   <activity
       android:name=".SplashActivity"
       android:theme="@style/SplashTheme"
       android:screenOrientation="portrait"
       android:configChanges="orientation|screenSize"
       android:label="Old UK" >
       <intent-filter>
           <action android:name="android.intent.action.MAIN" />
           <category android:name="android.intent.category.LAUNCHER"
   />
       </intent-filter>
   </activity>
   ```

We have encountered **themes** and **styles** before and, here, we took advantage of a built-in theme designed for **full screen activities**. In many cases, we might have designed a landscape layout here but, as is often the case with splash screens, we locked the orientation with the `android:screenOrientation` property.

The `android:configChanges` line is not actually needed here, but is included as it is useful to know about it. Configuring any attribute such as this prevents the system from automatically reloading the Activity whenever the device is rotated or the screen size changed. Instead of the Activity restarting, the `onConfigurationChanged()` method is called. This was not needed here as the screen size and orientation were taken care of in the previous lines of code and this line was only included as a point of interest.

Finally, we changed the value of `android:label`. You may have noticed that, depending on the screen size of the device you are using, the name of our app is not displayed in full on the home screen or apps drawer. In such cases, when you want to use a shortened name for your app, it can be inserted here.

With everything else in place, we can get on with adding our gesture detector. This is not dissimilar to the way we did this before but, this time, we will apply the detector to the whole screen and will be listening for a long press, rather than a fling.

Adding the GestureDetector

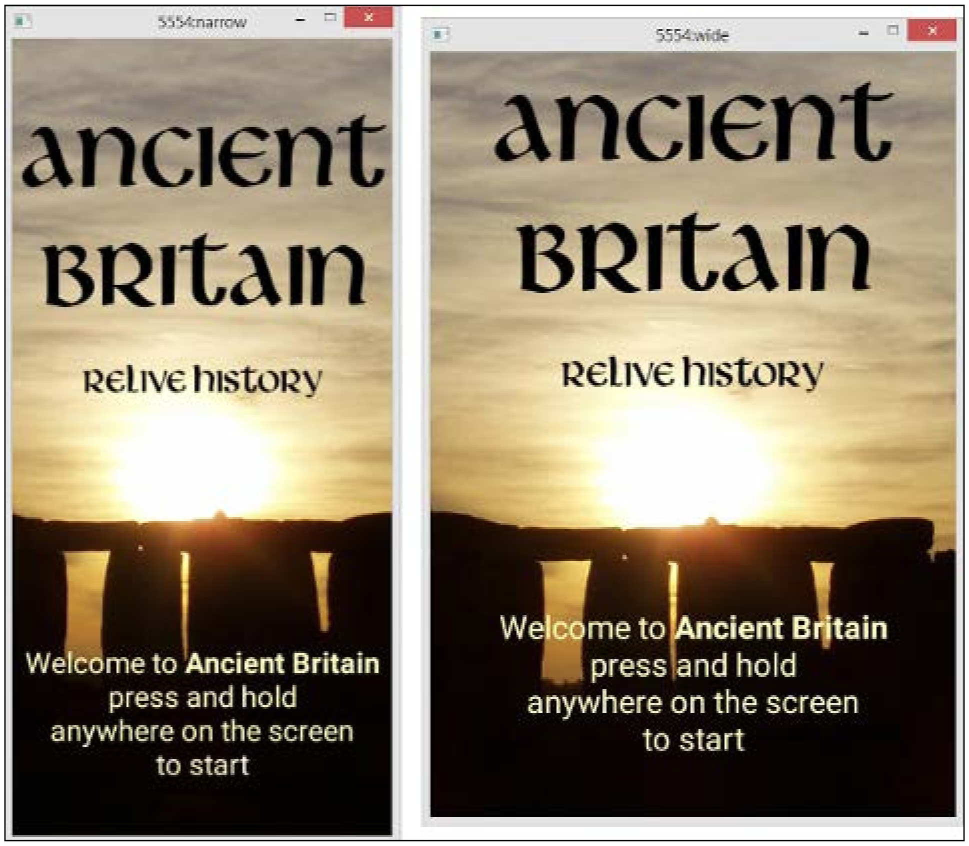
Along with implementing a gesture detector for the entire Activity here, we will also take the final step in configuring our splash screen so that the image fills the screen, but maintains its aspect ratio. Follow these steps to complete the app splash screen.

1. Open the `SplashActivity` file.
2. Declare a `GestureDetector` as we did in the earlier exercise:
   ```
   private GestureDetector detector;
   ```
3. In the `onCreate()` method, assign and configure our splash image and gesture detector like this:
   ```
   ImageView imageView = (ImageView) findViewById(R.id.splash_image);
   imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);

   detector = new GestureDetector(this, new SplashListener());
   ```
4. Now, override the Activity's `onTouchEvent()` like this:
   ```
   @Override
   public boolean onTouchEvent(MotionEvent event) {
       this.detector.onTouchEvent(event);
       return super.onTouchEvent(event);
   }
   ```
5. Create the following `SimpleOnGestureListener` class:
   ```
   private class SplashListener extends GestureDetector.
   SimpleOnGestureListener {

       @Override
       public boolean onDown(MotionEvent e) {
           return true;
       }

       @Override
   ```

```
        public void onLongPress(MotionEvent e) {
            startActivity(new Intent(getApplicationContext(),
MainActivity.class));
        }
}
```

6. Build and run the app on your phone or an emulator.

The way a gesture detector is implemented across an entire Activity should be familiar by this point, as should the capturing of the long press event. The `ImageView.setScaleType(ImageView.ScaleType)` method is essential here; it is a very useful method in general. The `CENTER_CROP` constant scales the image to fill the view while maintaining the aspect ratio, cropping the edges when necessary.



Welcome to **Ancient Britain**
press and hold
anywhere on the screen
to start

Welcome to **Ancient Britain**
press and hold
anywhere on the screen
to start

There are several similar `ScaleTypes`, such as `CENTER_INSIDE`, which scales the image to the maximum size possible without cropping it, and `CENTER`, which does not scale the image at all. The beauty of `CENTER_CROP` is that it means that we don't have to design a separate image for every possible aspect ratio on the numerous devices our apps will end up running on. Provided that we make allowances for very wide or very narrow screens by not including essential information too close to the edges, we only need to provide a handful of images of varying pixel densities to maintain the image quality on large, high-resolution devices.

The scale type of `ImageView` can be set from within XML with `android:scaleType="centerCrop"`, for example.

You may have wondered why we did not use the built-in **Full-Screen Activity** from the wizard; we could easily have done so. The template code the wizard creates for a Full-Screen Activity provides far more features than we needed for this exercise. Nevertheless, the template is worth taking a look at, especially if you want a fullscreen that brings the status bar and other components into view when the user interacts with the Activity.

That brings us to the end of this chapter. Not only have we seen how to make our apps interact with touch events and gestures, but also how to send debug messages to the IDE and make a Full-Screen Activity.

Summary

We began this chapter by adding a `GestureDetector` to our project. We then edited it so that we could filter out meaningful touch events (swipe right and left, in this case). We went on to see how the `SimpleOnGestureListener` can save us a lot of time when we are only interested in catching a subset of the recognized gestures. We also saw how to use DDMS to pass debug messages during runtime and how, through a combination of XML and Java, the status and action bars can be hidden and the entire screen be filled with a single view or view group.

Next, we will explore how our apps can communicate with the user through the various forms of notification the framework provides. We will also see how to allow the user to add their own data to the app and configure some settings.

6
Notifications and the Action Bar

The vertically scrolling `RecyclerView`, which we have been dealing with in the previous chapters, is a great device for negotiating long, simple lists where there is either not much data in each item or where each item only displays a small part of the whole. There are, of course, times when we will want to display a full screen of information and still have this as a part of a list. This is where the `ViewPager` comes in as it allows us to chain together full-page, mini Activities called **Fragments** in such a way that we can navigate through them with simple, intuitive horizontal swipe gestures.

Along with making use of the variety of views and widgets available to us, there are often times when we would wish to inform our user of some event even when they are not focused on our app. Smartphones generally provide some form of notification area, and Android is no exception with its **notification bar** at the top of the screen. Issuing notifications to the user is a very simple process, and Android has added two new features in Lollipop so that it is now possible to add display notifications on the lock screen and produce floating, heads-up notifications.

To demonstrate these features, we will build a small weather forecast app (using fake data) that will issue warnings in the form of notifications to the user when the weather is severe.

In this chapter, you will learn how to:

- Construct a screen slide with `ViewPager`
- Use Fragments instead of Activities
- Create a `ViewPager` and `PagerAdapter`
- Add tabs to the action bar

- Issue notifications to the user
- Manage a back stack
- Design icons for the notification bar
- Produce heads-up notifications
- Create expanded notifications
- Configure lock screen notifications
- Set notification priority and visibility

Constructing a ViewPager

The `ViewPager` and its variants are extended from the `ViewGroup` class and can be thought of as a kind of layout manager that takes care of placing and navigation through the pages. `ViewPagers` work in conjunction with `PagerAdapters`, which populate each page with the appropriate data.

The code behind the pages of our `ViewPager` is contained within a new class, the `Fragment`. Fragments are very similar in structure and purpose to Activities, but can be contained and combined within Activities and, to all intents and purposes, can be thought of as sub-Activities.

As always, we need to consider the layout and appearance of our app before we begin programming. Here, we will need layouts for the `ViewPager` and `Fragment`. The steps in the following section show how to set up our layout and resource files for this project.

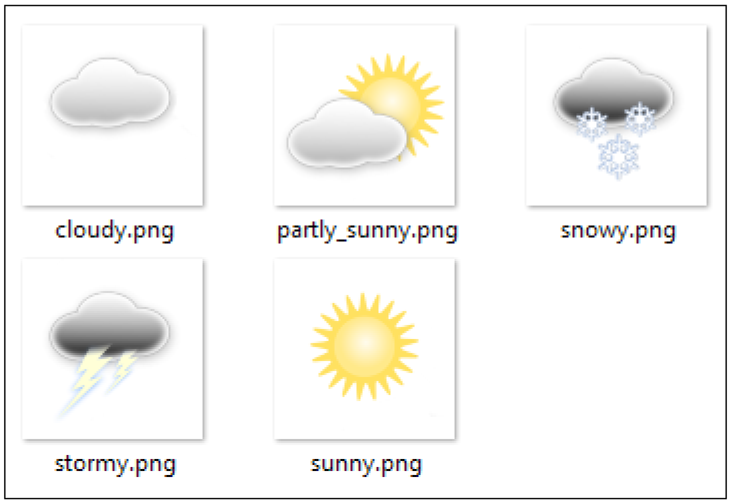
Creating the layout

Once created, this project will require two layout files and, to take advantage of some Material Design features, we will edit the styles resource. The following steps explain how this is done:

1. Create a new project in Android Studio.
2. Name it `Weather Forecast` and start it with the **Blank Activity** template.
3. Open the `build.gradle (Module: app)` file.
4. Add the following dependencies:

```
compile 'com.android.support:support-v4:21.0.+'
compile 'com.android.support:cardview-v7:21.0.+'
```

5. If you have not downloaded the project files, locate five suitable images of approximately 400 x 400 px with the following names and place them in the project `drawable` directory:



6. Open the `res/values-v21/styles.xml` file.
7. Add the following color items inside the `<style>` tag:

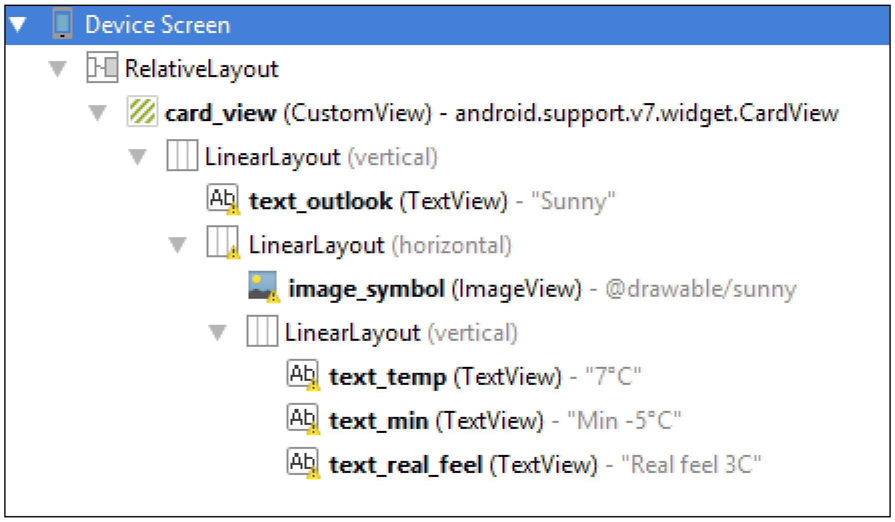
```
<item name="android:colorPrimary">#2b2</item>
<item name="android:colorPrimaryDark">#080</item>
<item name="android:colorAccent">#8f8</item>
<item name="android:textColorPrimary">#004</item>
<item name="android:textColorSecondary">#448</item>
```

8. Open the `activity_main.xml` file.
9. Replace its contents with this code:

```
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

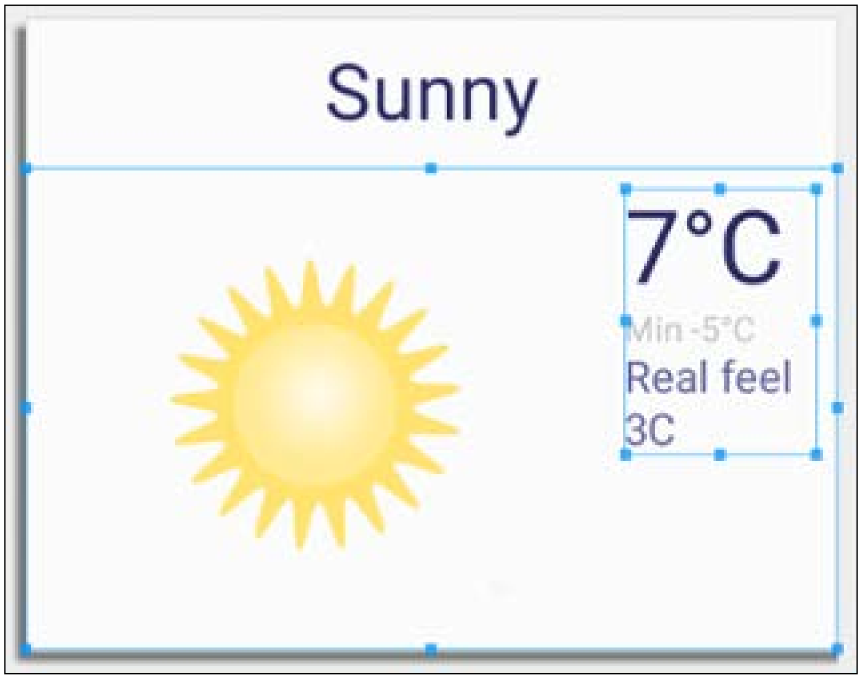
10. Create a new layout file called `fragment_layout.xml`.

11. Create a `ViewGroup` hierarchy to match the one shown here:



```
Device Screen
  RelativeLayout
    card_view (CustomView) - android.support.v7.widget.CardView
      LinearLayout (vertical)
        text_outlook (TextView) - "Sunny"
        LinearLayout (horizontal)
          image_symbol (ImageView) - @drawable/sunny
          LinearLayout (vertical)
            text_temp (TextView) - "7°C"
            text_min (TextView) - "Min -5°C"
            text_real_feel (TextView) - "Real feel 3C"
```

12. Using images from the `drawable` folder and text as placeholders, recreate the layout (with the inner two layouts highlighted), as shown here, selecting margins and other visual attributes as you see fit:



13. The `CardView` element should look like this:

```
<android.support.v7.widget.CardView xmlns:android=
  "http://schemas.android.com/apk/res/android"
  xmlns:card_view="http://schemas.android.com/apk/res-auto"
  android:id="@+id/card_view"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  card_view:cardCornerRadius="4dp"
  card_view:cardElevation="4dp">
```

14. To achieve the correct text coloring, set `android:textAppearance` to apply the `textColors` we set in the `styles.xml` file with `textColorPrimary` being applied to views and with `textAppearance` set to `"?android:attr/textAppearanceLarge"`.

We added a **support library (v4)** to our gradle script along with `CardView` support, which we will use later. Support libraries provide features that are unavailable in the standard SDK and backward-compatibility for many others.

That's all the layout and resource work we need to do. By applying custom colors in the style file, we allow the system to decide which colors to apply to which screen components as always; here, however, it applies our own personal scheme. These colors are displayed throughout the app; not only this method is a good way to ensure that our apps are distinctive but it also gives them a consistent feel for very little effort on our behalf.

[When dealing with large layout files, the process can be tidied up somewhat with the use of `<include layout="@layout/some_layout"/>` where `some_layout` will be inserted in place of the include tag on inflation.]

With the layout, theme and style set up, we can now continue with writing some code.

Adding a ViewPager and FragmentPagerAdapter

As briefly mentioned already, a `ViewPager` is a layout manager for horizontally scrolling screens. The work of filling these pages is undertaken by a `PagerAdapter`, or more specifically in this case `FragmentPagerAdapter`, which fulfills the same function but with Fragments rather than Activities.

Including a small dataset, we will require a total of four classes to implement our `ViewPager`. Here is how it's done:

1. Open the `MainActivity` class.
2. Include the following imports:
   ```
   import android.support.v4.app.FragmentActivity;
   import android.support.v4.app.NotificationCompat;
   import android.support.v4.app.TaskStackBuilder;
   import android.support.v4.view.ViewPager;
   ```
3. Extend the class like this:
   ```
    public class MainActivity extends FragmentActivity{
   ```
4. Add this class field:
   ```
   private ViewPager viewPager;
   ```
5. Add the following three lines to the `onCreate()` method to initiate our `ViewPager`:
   ```
   viewPager = (ViewPager) findViewById(R.id.pager);
   FragmentAdapter adapter = new FragmentAdapter(getSupportFragmentMa
   nager());
   viewPager.setAdapter(adapter);
   ```
6. Create a new class called `FragmentAdapter.java`.
7. Include the following imports:
   ```
   import android.support.v4.app.Fragment;
   import android.support.v4.app.FragmentManager;
   import android.support.v4.app.FragmentPagerAdapter;
   ```
8. Complete the class, as shown here:
   ```
   public class FragmentAdapter extends FragmentPagerAdapter {

       public FragmentAdapter(FragmentManager m) {
           super(m);
       }

       @Override
       public Fragment getItem(int index) {
           Fragment fragment = new WeatherFragment();
           Bundle args = new Bundle();
           args.putInt("day", index);
           fragment.setArguments(args);
           return fragment;
   ```

```
        }

        @Override
        public int getCount() {
            return 5;
        }
    }
```

9. Create a new class called `WeatherFragment.java`.
10. Add this import:

    ```
    import android.support.v4.app.Fragment;
    ```

11. Fill out the class like this:

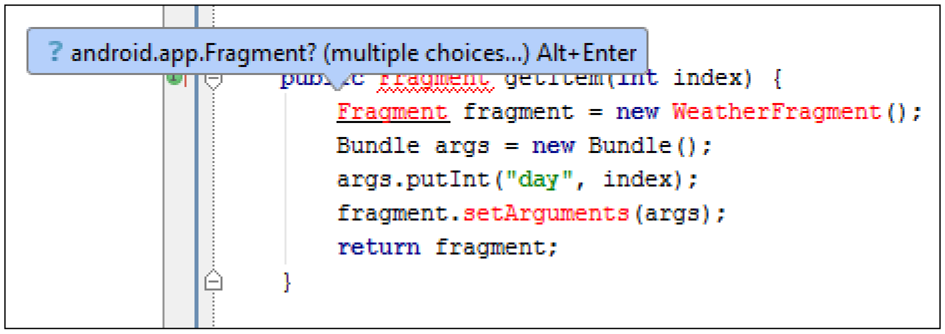
    ```
    public class WeatherFragment extends Fragment {

        @Override
        public View onCreateView(LayoutInflater inflater,
          ViewGroup container, Bundle savedInstanceState) {
          View rootView = inflater.inflate(R.layout
            .fragment_layout, container, false);
          Bundle args = getArguments();
          int i = args.getInt("day");

          TextView textOutlook = ((TextView)
            rootView.findViewById(R.id.text_outlook));
          ImageView symbolView = ((ImageView)
            rootView.findViewById(R.id.image_symbol));
          TextView tempsView = ((TextView)
            rootView.findViewById(R.id.text_temp));
          TextView windView = ((TextView)
            rootView.findViewById(R.id.text_min));
          TextView realFeelView = ((TextView)
            rootView.findViewById(R.id.text_real_feel));

          textOutlook.setText(WeatherData.outlookArray[i]);
          symbolView.setImageResource(
            WeatherData.symbolArray[i]);
          tempsView.setText(WeatherData.tempsArray[i] + "°c");
          windView.setText(("Min " +
            WeatherData.minArray[i] + "°c"));
          realFeelView.setText("Real feel " +
            WeatherData.realFeelArray[i] + "°c");

          return rootView;
        }
    }
    ```

12. Create a last class called `WeatherData.java` and complete it like following:

```
public class WeatherData {
  static String[] outlookArray = {"Developing snow storms",
    "Partly sunny and breezy", "Mostly sunny", "Afternoon
    storms", "Increasing cloudiness"};
  static Integer[] symbolArray = {R.drawable.snowy,
    R.drawable.partly_sunny, R.drawable.sunny,
    R.drawable.stormy, R.drawable.cloudy};
  static Integer[] tempsArray = {0, 1, 3, 2, 4};
  static Integer[] minArray = {-5,-3,-2,0,2};
  static Integer[] realFeelArray = {-1, 2, 0, 1, 3};
}
```

13. You can now run and test the app on a handset or emulator.

You may be wondering why instructions on which libraries to import were included in this section as you probably have Auto Import configured in the settings. This feature works perfectly in almost all situations and is a great time-saver. If you have not already enabled it, it can be found at `File | Settings... | IDE Settings | Editor | Auto Import`. It is only useful when a support library contains imports that have the same name as the standard libraries. Had we not imported these libraries first, we would have encountered something similar to the following screenshot when we tried to enter the code:



```
? android.app.Fragment? (multiple choices...) Alt+Enter
    public Fragment getItem(int index) {
        Fragment fragment = new WeatherFragment();
        Bundle args = new Bundle();
        args.putInt("day", index);
        fragment.setArguments(args);
        return fragment;
    }
```

When this occurs, it is still perfectly possible to import the correct libraries by simply selecting the v4 version from the list the editor offers.

The `MainActivity` code is remarkably short as most of the work is done elsewhere. `ViewPagers` requires an ID, which is the XML-defined `ViewPager` as set in our `activity_main.xml` file. The `PagerAdapter` is implemented in the next class; here, all we do is to connect the two.

The `FragmentAdapter` class is also quite straightforward. All we need to do is pass the index of the selected page to our `Fragment` class, which we do with the Bundle object. The fragment then takes care of laying out the views with the appropriate data. It is important to point out that we did not use a `PagerAdapter` directly, but rather one of its variants, the `FragmentPagerAdapter`, which is designed to work with fragments There is also the `FragmentSatePagerAdapter`, which does an identical job, but is better suited to longer lists. The reason for this is the way the framework handles those pages that are not directly visible to the user. For short lists, such as ours, the best performance can be obtained with `FragmentPagerAdapter`.

The `Fragment` class we created is also very simple to understand as it does little more than inflating our `fragment_layout`, associating and setting our views in a way that is, by now, very familiar to us. Fragments are very similar to Activities as they each have their own life cycle, which can be intercepted at various points with callback methods, such as `onCreate()` and `onPause()` in the same way as we do with Activities. Here, we chose to use the `onCreateView()` method as it provides access to the instances we need to inflate the fragment along with the page index stored in a Bundle by the adapter and is also called when the view is created.

`ViewPagers` provides a lot of functionality with very little effort. Once connected to an adapter, we can create a screen slide without worrying about implementing touch listeners and having to pay much attention to which page has been selected at any given time.

As it stands, our app does very little, and we need to provide more functionality, which we will do by firstly adding tabs to the action bar so that we can associate each page with an actual date, and secondly by programming a notification builder to issue weather alerts via the system notification area. We will begin by adding tabs with dates.

Adding tabs and dates

A weather forecasting app is of no use if the user does not know when to expect the weather predicted. We could easily add another view to our fragment layout, but we will attach our pages to the tabs on the action bar, and access calendar and date format classes to populate them.

To intercept changes between the pages, we will need an `OnPageChangeListener`, which is called whenever the page changes, and to redefine the class declaration so that it implements an `ActionBar.TabListener`. We will use Java's own `Calendar` and `SimpleDateFormat` objects to calculate and format our dates.

Everything we need to do to add tabs to our action bar can be done from within our `MainActivity` class by following these steps:

1. Open the `MainActivity.java` file.
2. Edit the declaration like this:

   ```
   public class MainActivity extends FragmentActivity
     implements ActionBar.TabListener
   ```

3. Select the error this generates, follow the editor's recommendation, and implement the three methods suggested.
4. Edit the `onTabSelected()` method to match the following:

   ```
   @Override
   public void onTabSelected(ActionBar.Tab tab,
     FragmentTransaction ft) {
     viewPager.setCurrentItem(tab.getPosition());
   }
   ```

5. Include the following class field:

   ```
   private ActionBar actionBar;
   ```

6. Add the following lines to the `onCreate()` method:

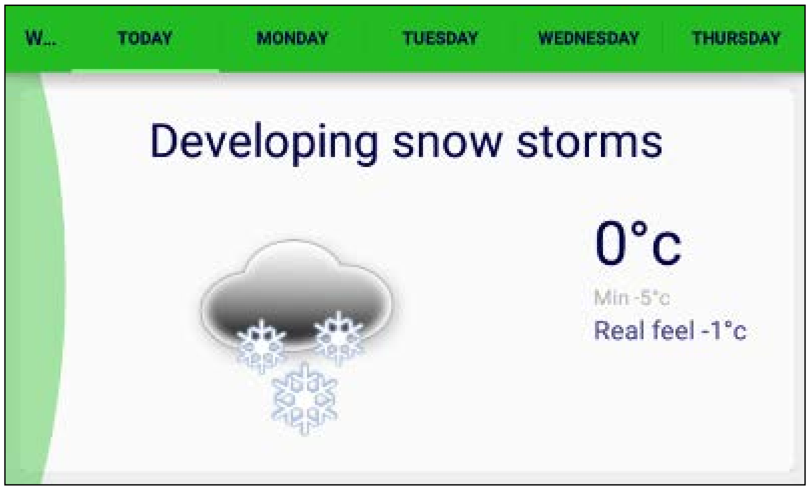
   ```
   Calendar calendar = Calendar.getInstance();
   String weekDay;
   SimpleDateFormat dayFormat;
   dayFormat = new SimpleDateFormat("EEEE",
     Locale.getDefault());

   actionBar = getActionBar();
   actionBar.setNavigationMode(ActionBar
     .NAVIGATION_MODE_TABS);
   actionBar.addTab(actionBar.newTab().setText("Today")
     .setTabListener(this));

   for (int i = 1; i < 5; i++) {
     calendar.add(Calendar.DAY_OF_WEEK, 1);
     weekDay = dayFormat.format(calendar.getTime());
     actionBar.addTab(actionBar.newTab().setText(weekDay)
     .setTabListener(this));
   }
   ```

7. Still within `onCreate()`, include the following listener to detect page changes:

```
viewPager.setOnPageChangeListener(new
  ViewPager.OnPageChangeListener() {

  @Override
  public void onPageScrolled(int i, float v, int i2) {
  }

  @Override
  public void onPageSelected(int position) {
    actionBar.setSelectedNavigationItem(position);
  }

  @Override
  public void onPageScrollStateChanged(int i) {
  }
});
```

8. You can now run the project on a device or AVD.

The tabbed action bar is a familiar sight on many Android applications and the `ActionBar.TabListener` is an interface that is called whenever a tab is selected, added, or removed. Here, we use its selection to inform our `ViewPager` of the change. The `ViewPager` layout also provides a good opportunity to see how the colors we defined earlier are applied to the various components of the UI, such as the colored hint that appears when the user attempts to scroll beyond either end of the list.



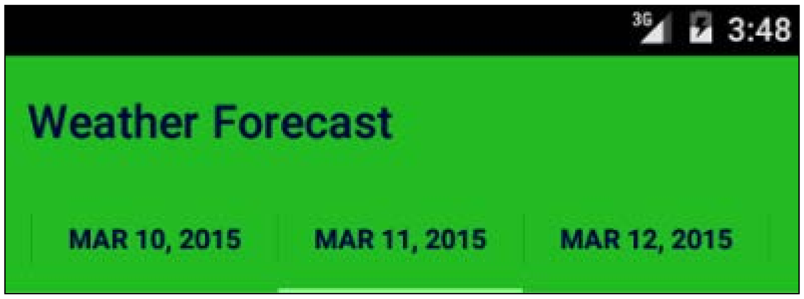
Developing snow storms

0°c

Min -5°c

Real feel -1°c

The `Calendar` and `SimpleDateFormat` classes are not hard to follow. The date formatting follows **Unicode Technical Standards** (**UTS**) #35, the details of which can be found at `http://www.unicode.org/reports/tr35/tr35-dates.html#Date_Format_Patterns`. Here, we applied a stand-alone day-of-week format with `"cccc"`. We could have been more creative and used something like `"c LLL d"` to have something like **Tue Mar 15** or even taken advantage of the system's built-in formatting by changing the `dayFormat` assignation to `dayFormat = (SimpleDateFormat) new SimpleDateFormat().getDateInstance();`, as seen here:



The final part of the puzzle is put into place with the `ViewPager.OnPageChangeListener`, which we use to inform the action bar when a new page is selected with a swipe rather than a tab.

With both forms of navigation in place and connected, we can now move on and program our app to issue notifications.

Programming notifications

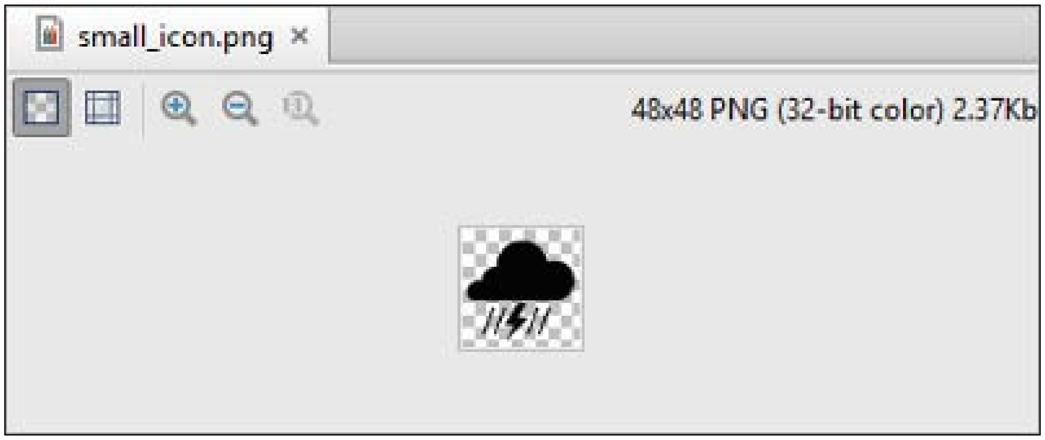
Android 5 provides a more flexible notification framework than the previous versions. In addition to being able to issue standard notifications to the notification bar, we can now expand notifications to include more detail than before, and are able to notify the user when they are using a full-screen app, or even when their screen is locked, with **heads-up notifications** that float above the screen for a moment.

When using notifications, it is important to be sensitive to our user's needs and not issue too many notifications or give them too great a level of importance. It is for this reason that notifications can be assigned **priority and visibility** settings. With that said, it is now time to proceed with adding notifications to our app.

Adding a standard notification and icon

Assuming the notification we wish to deliver is neither important nor personal, and all we wish to do is to inform the user of some event and offer them the opportunity to open our app, we need to issue a standard notification so that a small icon appears on the notification bar and, when the drawer is opened, a small card appears with some brief text and an icon. This, if clicked on, will open our app. Notifications require a particular type of icon for display on the bar. We will begin with this and then continue to implement a notification. Complete the following steps to set this up on our weather app:

1. If you have not downloaded the project files, you will need a small icon of a single color against a transparent background (like the one you see here) called `small_icon.png`, and stored in the `drawable` directory:



2. Open the MainActivity Java file.
3. Add this class field:

```
private static int notificationId;
```

4. In the `onCreate()` method, add the following clause:

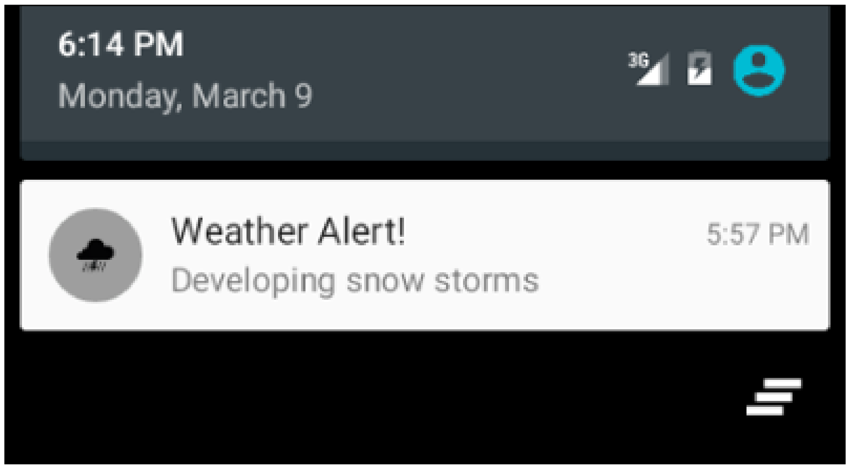
```
if (notificationId == 0) {
    postAlert(0);
}
```

5. Create the `postAlert()` method and complete it like this:

```
private void postAlert(int i) {
  NotificationCompat.Builder builder = new
    NotificationCompat.Builder(this);
  builder.setContentTitle("Weather Alert!")
    .setContentText(WeatherData.outlookArray[i])
```

```
            .setSmallIcon(R.drawable.small_icon)
            .setAutoCancel(true)
            .setTicker("Wrap up warm!");

        Intent intent = new Intent(this, MainActivity.class);
        TaskStackBuilder stackBuilder =
          TaskStackBuilder.create(this);
        stackBuilder.addParentStack(MainActivity.class)
          .addNextIntent(intent);
        PendingIntent pendingIntent =
          stackBuilder.getPendingIntent(0,
          PendingIntent.FLAG_UPDATE_CURRENT);
        builder.setContentIntent(pendingIntent);

        NotificationManager notificationManager =
          (NotificationManager) this.getSystemService(
          Context.NOTIFICATION_SERVICE);
        notificationManager.notify(notificationId,
          builder.build());

        notificationId++;
    }
```

6. The application can now be tested on an emulator or device, and opening the notification drawer should produce the following result:

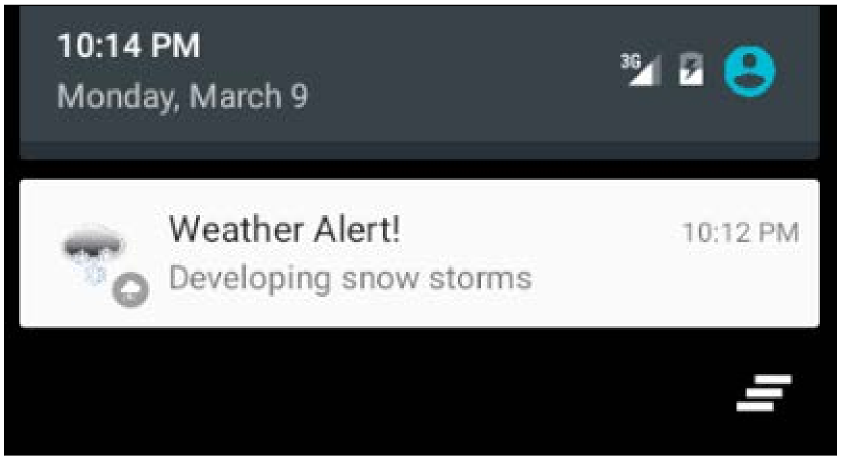

6:14 PM
Monday, March 9
Weather Alert!
Developing snow storms
5:57 PM

There are quite a few things going on here. Firstly, there was the **notification icon**. This is required by the builder and has to be of a specific format. Apart from being small, the icon needs to be a simple design on a transparent background. The color of the image is immaterial as only the alpha channel is considered by the system. For this reason, colors with intermediate alpha levels are strongly discouraged.

Before getting to the rest of the code, it would be amiss not to point out that there is an alternative to using the small icon on the notification pop-down, which, although suitable for the minimal space on the bar, does not always look great in the circular icon to the left of the larger view. Add the following setting to our builder:

```
.setLargeIcon(BitmapFactory.decodeResource(getResources(),
  WeatherData.symbolArray[i]))
```

Running the app now produces an icon like this:



We used `NotificationCompat` to construct our builder. `NotificationCompat` is a helper class provided by the support libraries we imported earlier. The `Builder` itself has three required parameters: `ContentTitle`, `ContentText`, and `SmallIcon`. There are many others, which we will get to, but only these three are mandatory. I have also set `AutoCancel` to true as this is extremely helpful because it automatically closes the notification once it has been selected from the drawer. The purpose of setting `Builder.Ticker` would have become apparent when the app was run.

Because of their available position, notifications can be viewed and triggered when other apps have the focus. This can lead to confusing results for the user when they press the back key. We want the navigation keys to behave as if the app had been launched in the usual way. This is the function of the **TaskStackBuilder**, which takes an **Intent** in this case to launch our MainActivity, and places it onto the recent activities back stack and the PendingIntent makes it available outside our app.

The actual notification is called with the `NotificationManager.notify()` call. The use of an ID was not strictly necessary here as our app only issues one notification at a time. However, not only is it a useful way to keep track of multiple notifications, but can also be used, as we did here, to ensure that a notification is issued only once.

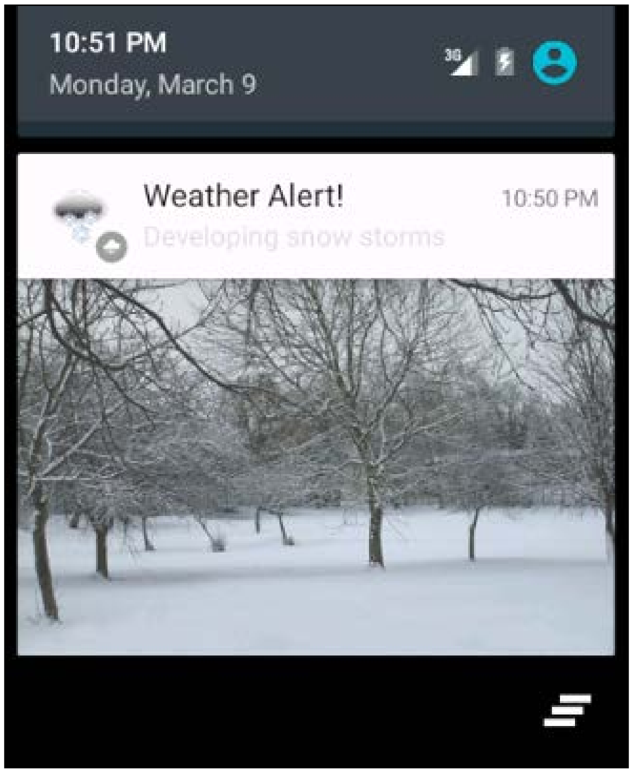
Expanding a notification

The `NotificationCompat` class provides us with three built-in styles. Here, we will use `BigPictureStyle`. Before we start, you will need an image to use as the "big picture". Find a suitable image of around 640 x 480 px, place it in your `drawable` folder, and name it `snow_scene` or change the reference in the following code. Once your image is in place, follow these short steps:

1. Open the `MainActivity.java` class.
2. Add the following code to the `onCreate()` method, just before our Intent is declared:

```
NotificationCompat.BigPictureStyle bigStyle = new
  NotificationCompat.BigPictureStyle();
bigStyle.bigPicture(BitmapFactory
  .decodeResource(getResources(),
  R.drawable.snow_scene));
builder.setStyle(bigStyle);
```

3. That's it! Run the app on your handset or emulator and open the notification drawer.



The `NotificationCompat.Style` objects are very handy to add details to our notifications, and simple to understand and use. Along with `BigPictureStyle`, there is `BigTextStyle` for headline type notifications and `InboxStyle` for lists.

This leaves us with two other new notification features: heads-up notifications and lock screen notifications. Neither of these requires any additional coding as such, but they are triggered by adjusting certain privacy and priority settings of the existing notifications. The next and final section of this chapter demonstrates how this is done.

Issuing heads-up and lock screen notifications

Heads-up and lock screen notifications are created in the same way as the standard notifications. The difference depends on the `VISIBILTY` and `PRIORITY` properties of our `NotificationCompat.Builder` class. The following are the steps to do this:

1. Open the `MainActivity` class.
2. Give our builder instance the following properties:

```
.setPriority(Notification.PRIORITY_HIGH)
.setVisibility(Notification.VISIBILITY_PUBLIC)
.setVibrate(new long[]{100, 50, 100, 50, 100})
.setCategory(Notification.CATEGORY_ALARM)
```

3. When the app is run, it will now display heads-up and lock screen notifications.

The screen can be locked on an emulator by pressing *F7*.

Here, the `setPriority()` method is how we decide whether a notification is important enough to consider using a heads-up message that may intrude on some other task they are performing more than the standard version. Priority must be set to `PRIORITY_HIGH` or `PRIORITY_MAX` and the notification must be set to trigger a vibration for notifications to appear in this manner. The other priorities are `MIN`, `LOW`, and `DEFAULT`.

If you do not wish for your notification to set off a vibration, but still appear in a heads-up fashion, you can use the following line instead:

```
builder.setVibrate(0);
```

Although there are more intricate ways to configure a vibration, the constructor used here will suit most purposes. The array of longs represents alternating times in milliseconds, representing pulses of vibration and silence alternatively so that, in the example here, the device would buzz three times for 100 ms with pauses of 50 ms in between.

Unlike other notifications, the user can choose, via settings, whether or not the device displays lock screen messages. As developers, however, we can decide how much, if any, information is displayed on the user's lock screen.

Setting the visibility of a notification to `PUBLIC` will cause both content title and content text to be displayed, setting it to `PRIVATE` will display only the title, and `SECRET` will display nothing at all.

Summary

In this chapter, we explored an alternative to the vertically scrolling `RecyclerView` with the `ViewPager`. We used Fragments instead of Activities, and saw some of the things we can do with the Action Bar. We learnt how to issue notifications of all kinds and how to manage the back stack so as to provide consistent navigation back through our app for the user. In most cases, notifications are sent when our app is not even actively running; to do this, we will need to use Services, which are a kind of background Activity.

In the next chapter, we will take a look at how to include Google Maps into our apps and make them location-aware. This involves having to register our app for an API key and employing a `LocationListener` to keep our app updated with current location data; as this can be resource hungry, we will also see how to optimize this process.

7
Maps, Locations, and Google Services

The Standard SDK APIs that we have used until now provide a powerful set of tools for developing all manner of apps. However, Google also provides a number of mobile services such as Gmail, Translate, and Maps. These, and a dozens of others, are available to us as developers, and Google provides APIs for us to interact with them and incorporate them into our own apps.

[A complete and up-to-date list of all Google Services APIs can be found at: `https://developers.google.com/apis-explorer/#p/`.]

Because apps connecting to Google Services use Google's data and servers, there is a simple authentication process required which is known as an API Key. In this chapter, we will see how to do this as we build a simple map-based app that displays a location of our choice. After this is done, we will use a `LocationListener` to track our app as the user moves around.

In this chapter, you will:

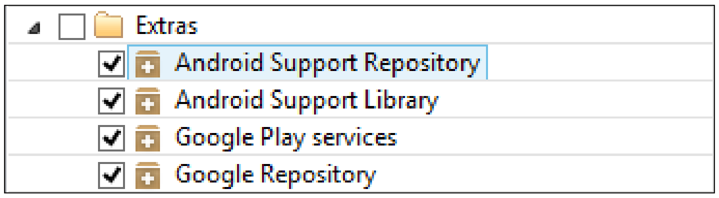
- Obtain an API Key to access Google Maps for Android
- Understand permissions and how to apply them
- Access `LocationServices` with a `GoogleApiClient`
- Acquire a device's last known location
- Use a `LocationListener` to update the location
- Optimize location update intervals
- Add Google Maps UI features
- Set mock locations
- Acquire a location with a `MapClickListener`

Building a location-aware app with Google Maps

To get a very basic Google Map into one of our apps requires two distinct steps. First, we need to register our app with Google and acquire an API Key to uniquely identify our app; once we have a map up-and-running, we can locate our position using GPS and then zoom in to that, or any other, location. We begin with the first of these steps.

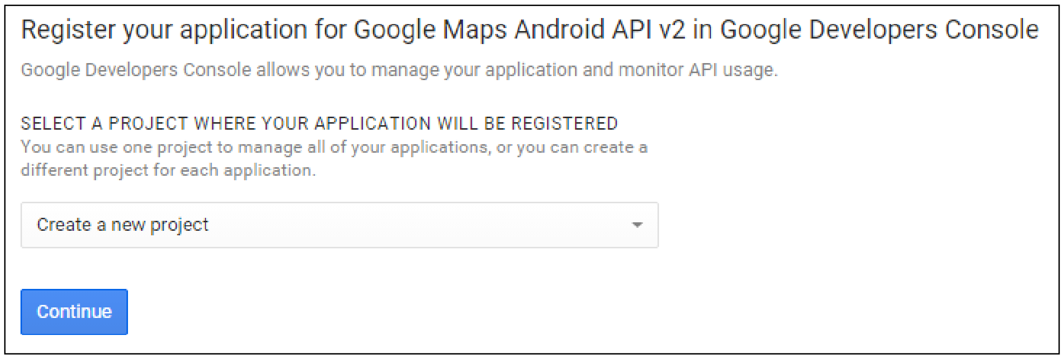
Acquiring an API Key

There is nothing to stop us getting on with this right away, although you will need to first check that, when we were installing all the components of the SDK way back in *Chapter 1, Setting Up the Development Environment*, we included the following packages:


Extras
[✓] Android Support Repository
[✓] Android Support Library
[✓] Google Play services
[✓] Google Repository

The only other thing to note before we begin is that, if you intend to test this app using an emulator, then you will need to construct a new one, where the system image target is **Google APIs (Google Inc.) – google_apis [Google APIs]**, rather than **Android 5.x**. Third-party virtual devices may require their own configuration to run Play Services. With this done, we are more than ready to create our location-based app:

1. Start a new Android Studio project.
2. Select **Google Maps Activity**, where previously we have selected **Blank Activity**, on the appropriate screen.
3. Leave everything else as suggested by the wizard.
4. The editor should open with the `google_maps_api.xml` file; if not, open it from the `res/values` directory.
5. Examine the code. Google will have provided a link beginning with `https://console.developers.google.com/flows/` and ending in your package name, for example `com.example.kyle.distancefinder`.
6. Follow this link and you will be taken to the Google Developers Console.


Register your application for Google Maps Android API v2 in Google Developers Console

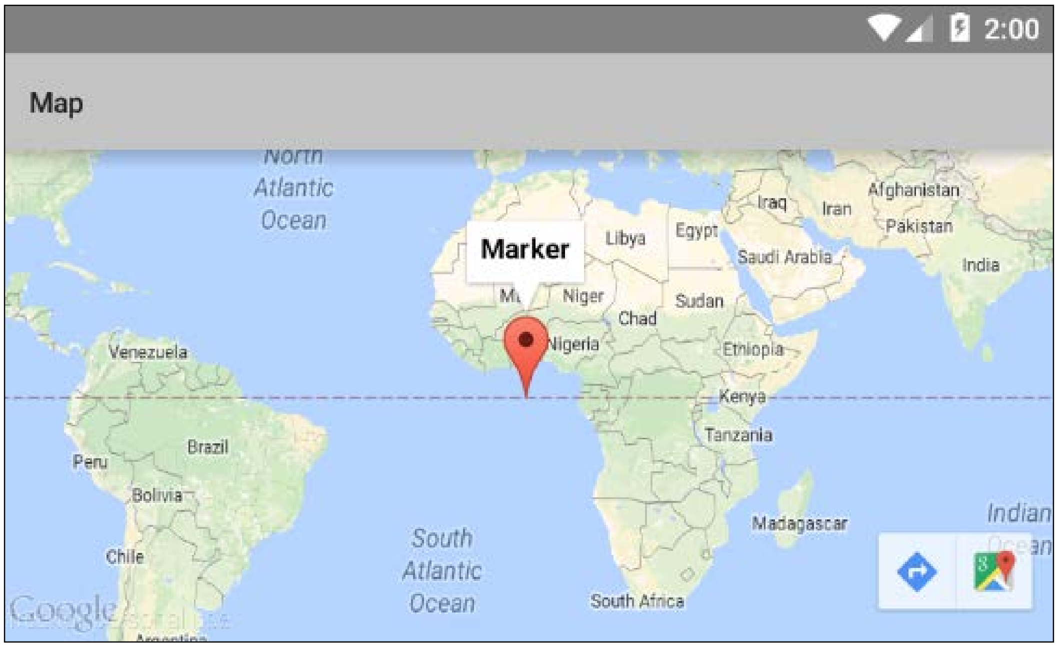
Google Developers Console allows you to manage your application and monitor API usage.

SELECT A PROJECT WHERE YOUR APPLICATION WILL BE REGISTERED

You can use one project to manage all of your applications, or you can create a different project for each application.

7. Sign up, if needed.
8. You will be prompted to create a new project. Call this whatever you choose, as you will be able to use this again for other apps.
9. From the **Project Dashboard** sidebar under **APIs & auth**, select **APIs**.
10. Enable the **Google Maps Android API v2** API.
11. Again, from the **Dashboard** under **APIs & auth**, select credentials and click on the **Create New Key** button.
12. On the resultant screen, copy the API Key and paste it into the `google_maps_api.xml` file, where it says `YOUR KEY HERE`, making sure there are no extra spaces at either end.

13. Now test the app on a handset or Google APIs emulator. The result will resemble the following screenshot:



Incorporating a basic map into our app is pleasantly simple. However, by using the Maps Activity wizard, a lot of essential work has been done for us. And it is vital to understand that before being able to include maps anywhere in our app that we choose.

Take a look at the `build.gradle` file and note how the dependencies have been modified for us:

```
dependencies {
  compile fileTree(dir: 'libs', include: ['*.jar'])
  compile 'com.google.android.gms:play-services:6.5.87'
  compile 'com.android.support:appcompat-v7:22.0.0'
}
```

When including maps in other projects, we will always need the `gms:play-services` libraries built in here. The `support:appcompat` library may be less obvious. It is used to make apps backward-compatible. It is not strictly necessary here and we will return to it when we cover how to reach the maximum number of users in the final chapter.

Now open the project's `Manifest` file. You will notice several differences from those of previous projects, the first being the following lines:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
  android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name=
  "com.google.android.providers.gsf.permission.READ_GSERVICES" />
<uses-permission
  android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission
  android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Anyone who has downloaded an Android app will be familiar with the way the user has to grant permissions to use various device functions, such as Internet access before installation. These tags, in the manifest, are how this is done, and you will need to apply them any time you include a feature that requires user permission. Thankfully, they all have very self-explanatory references and a full list can be found at `developer.android.com/reference/android/Manifest.permission.html`.

The other elements in the manifest that will be needed in future projects are the two meta-data children of the application element shown below. The first automatically keeps our app running the latest version of play services and the second is where the API Key we acquired earlier is applied:

```
<meta-data
  android:name="com.google.android.gms.version"
  android:value="@integer/google_play_services_version" />
<meta-data
  android:name="com.google.android.maps.v2.API_KEY"
  android:value="@string/google_maps_key" />
```

Overall, setting up API keys is very straightforward. We only need to register once and individual projects can be reused for apps requiring similar functions; speaking of functions, it is about time we added some functionality to our app. The most widely used, and arguably most useful Google Map APIs are the Location Services, which, among other things, allow the user to locate their device's geographical location using GPS, WiFi, and network signal strength.

Acquiring the last known location

The first step in incorporating location-aware technology into our apps is to identify the user's last known location. This, like much location-based work, is done with the help of a GoogleApiClient, and an interface that is a main entry point for these services.

Before adding the Java code to do this, we will edit the layout itself, so that we can see what is going on. Follow the next few steps to acquire our device's last known location:

1. Open the **Distance Finder** project.
2. Open the `activity_maps.xml` layout file.
3. Edit the content so:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:id="@+id/text_view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:text="No last location" />

  <fragment
    android:id="@+id/map"
    class="com.google.android.gms.maps.SupportMapFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MapsActivity" />

</LinearLayout>
```

4. Open the `MapsActivity.java` file.
5. Along with the `GoogleMap` that has been declared for us, include the following fields:

```
private static final String DEBUG_TAG = "tag";
private TextView textView;
private GoogleApiClient googleApiClient;
```

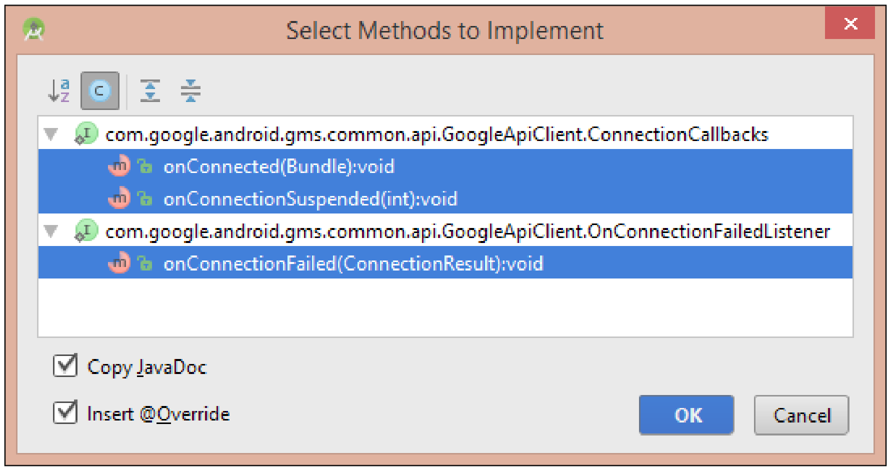
6. Add the following code to the `onCreate()` method:

```
textView = (TextView) findViewById(R.id.text_view);

googleApiClient = new GoogleApiClient.Builder(this)
  .addApi(LocationServices.API)
  .addConnectionCallbacks(this)
  .addOnConnectionFailedListener(this)
  .build();
```

7. Now, change the class declaration itself, so that it implements the following interfaces:

```
public class MapsActivity extends FragmentActivity
implements GoogleApiClient.ConnectionCallbacks,
  GoogleApiClient.OnConnectionFailedListener {
```

8. This will generate an error. Use the quick fix to implement the following methods:



Select Methods to Implement

com.google.android.gms.common.api.GoogleApiClient.ConnectionCallbacks
- onConnected(Bundle):void
- onConnectionSuspended(int):void

com.google.android.gms.common.api.GoogleApiClient.OnConnectionFailedListener
- onConnectionFailed(ConnectionResult):void

☑ Copy JavaDoc

☑ Insert @Override

OK Cancel

9. Complete the `onConnected()` callback like so, to display our location:

```
@Override
public void onConnected(Bundle bundle) {
  Location loc =
    LocationServices.FusedLocationApi.getLastLocation(
    googleApiClient);
  Log.d(DEBUG_TAG, "Connected");
  if (loc != null) {
    textView.setText(loc.toString());
  }
}
```

10. To report connection status to the LogCat, edit the other two new methods like this:

```
@Override
public void onConnectionSuspended(int i) {
  Log.d(DEBUG_TAG, "Connection suspended");
}

@Override
public void onConnectionFailed(ConnectionResult connectionResult)
{
  Log.d(DEBUG_TAG, "Connection  failed");
}
```

11. Edit the `onResume()` method like this:

```
@Override
protected void onResume() {
  super.onResume();
  setUpMapIfNeeded();
  googleApiClient.connect();
  Log.d(DEBUG_TAG, "onResume() called - connected");
}
```

12. Add a new `onPause()` method and complete it like so:

```
@Override
protected void onPause() {
  super.onPause();
  if (googleApiClient.isConnected()) {
    googleApiClient.disconnect();
    Log.d(DEBUG_TAG, "Disconnected");
  }
}
```

13. Finally, rewrite the `setUpMap()` like this:

```
private void setUpMap() {
  mMap.addMarker(new MarkerOptions()
    .position(new LatLng(51.178844, -1.826189))
    .title("Stonehenge"));
}
```

14. Run the project on a handset or AVD. Unless you are running the app on an emulator that you have just created, your location will appear in the text view with the following format:

    Location[fused 51.507350,-0.127757 acc=4 et=+5m16s558ms alt=19.809023 vel=0.0].

At the beginning of this exercise, we changed the layout a little to include a `TextView`. However, we still set the fragment's class as a `SupportMapFragment`, and hopefully a little more clearly. The `SupportMapFragment` is the obvious choice for any map container we might want in an app. It's simple and handles most map processes almost automatically. With this container, almost everything else we need can be accomplished with a `GoogleApiClient`. As can be seen from the code, it allows us to put in place listeners and callbacks to manage the map and connectivity activity. Once the API client has connected us, finding our device's last known location requires nothing more than a call to the single function, `getLastLocation()`.

It is vitally important to ensure that we disconnect from any services we are using, whenever our user may no longer need them. Failure to do this results in apps that use up the device's power and data unnecessarily. By using the `onPause()` callback to disconnect the client whenever our Activity loses focus and then `onResume()` to re-connect it, we can ensure that our map stays connected when the Activity is visible and also that we won't be wasting our user's battery and data when it is not.

The `getLastLocation()` method is very useful; it doesn't require a network connection, or GPS, and it's instantly available. However, there are many situations where we need to update our app's location as the user moves around. This is done with `LocationListen3er` callbacks and the `LocationRequest` object, which is laid out in the following section.

Requesting location updates

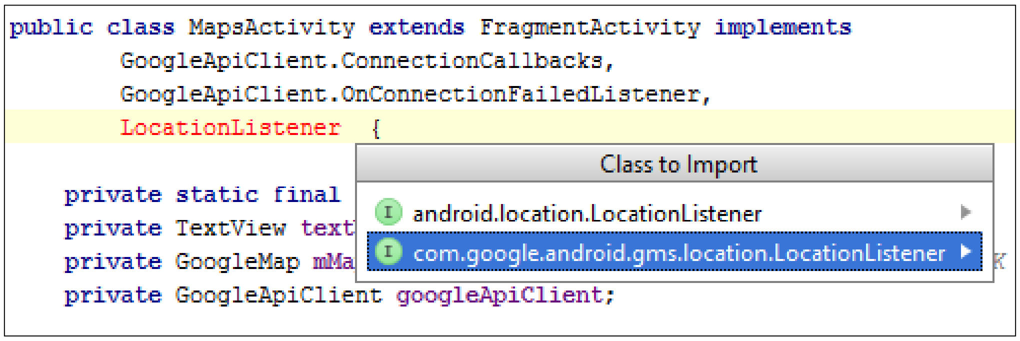
The `LocationRequest` object is highly configurable and allows us to control the frequency of requests and the accuracy of the information received. This means we can design apps that do not use more resources than required, while still allowing us highly accurate and frequent location data when the purpose of the app demands it.

In the next stage, we will implement a `LocationListener` to track our app's location. We will also add one or two features to demonstrate some of the functions the maps APIs provide us with. There are only a few lines of code to this section, and here they are:

1. Include a `LocationListener` in our `FragmentActivity` class declaration, like so:

   ```
   public class MapsActivity extends FragmentActivity implements
     GoogleApiClient.ConnectionCallbacks,
     GoogleApiClient.OnConnectionFailedListener,
     LocationListener {
   ```

2. This will generate an error. Use the quick fix to import the Google version of the `LocationListener`, like so:



```
public class MapsActivity extends FragmentActivity implements
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener,
        LocationListener  {

    private static final
    private TextView text
    private GoogleMap mMa
    private GoogleApiClient googleApiClient;
```

Class to Import
android.location.LocationListener
com.google.android.gms.location.LocationListener

3. Create the following class field:

```
private LocationRequest locationRequest;
```

4. In the `onCreate()` method, create the following `LocationRequest`:

```
locationRequest = LocationRequest.create()
  .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
  .setInterval(30000)
  .setFastestInterval(5000);
```

5. Next complete the `onLocationChanged()` method like this:

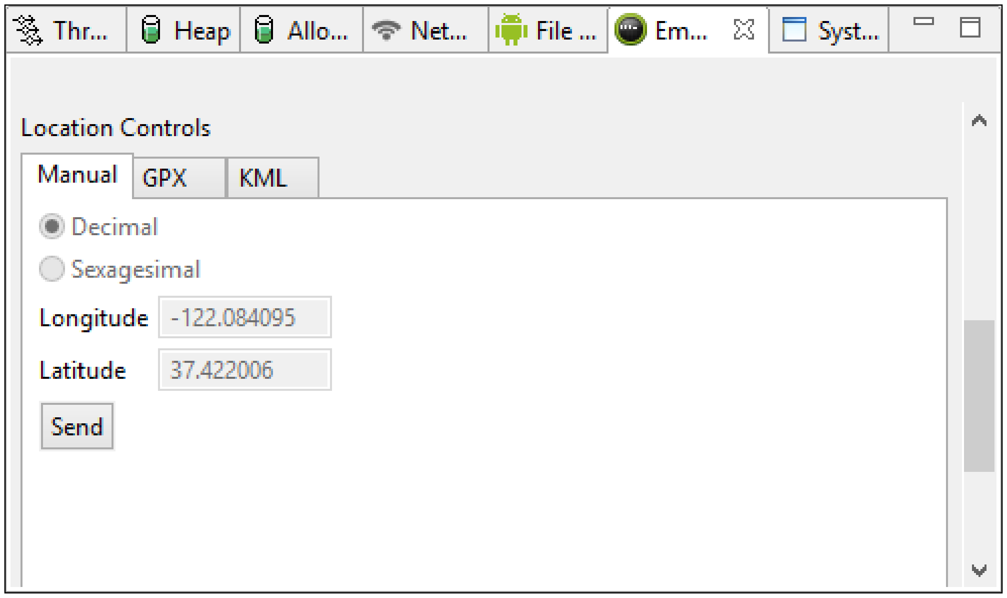
```
@Override
public void onLocationChanged(Location location) {
  LocationServices.FusedLocationApi.requestLocationUpdates(googleA
piClient, locationRequest, this);
  textView.setText(location.toString());
}
```

6. Now expand the `setUpMap()` method like so:

```
private void setUpMap() {
  mMap.addMarker(new MarkerOptions()
    .position(new LatLng(51.178844, -1.826189))
    .title("Stonehenge")
    .snippet("3000 BC")
    .icon(BitmapDescriptorFactory.fromResource(
      R.mipmap.ic_launcher)));

  mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
```

```
        mMap.setMyLocationEnabled(true);
        mMap.getUiSettings().setMyLocationButtonEnabled(true);
        mMap.getUiSettings().setZoomControlsEnabled(true);
        mMap.getUiSettings().setZoomGesturesEnabled(true);
        mMap.getUiSettings().setCompassEnabled(true);
        mMap.getUiSettings().setRotateGesturesEnabled(true);
    }
```

7. Run the project on your handset or emulator and either go for a short walk or set a mock-location using the Android Device Monitor:



Decimal (selected)

Sexagesimal

Longitude -122.084095

Latitude 37.422006

As with other aspects of coding for maps, the additions we made here are nicely straightforward. Although there is an Android version of the `LocationListener`, the Google one is easier to use, more powerful, and strongly recommended by Google themselves. We used the built-in icon here, but of course any image would have done. It is unlikely that we would use all the UI controls we set here and they are included more for demonstration purposes than practical use.

The way we set the intervals is of interest. The units are milliseconds and so here we have set the app to request a new location every 30 seconds. However, the use of `setFastestInterval()` allows us to take advantage of other apps that might be requesting a location and so we could have updates as frequently as 5 seconds.

We set the accuracy to as high as possible, with the `PRIORITY_HIGH_ACCURACY` constant. This obviously can be a drain on the user's power. Many apps require location to just within a few hundred feet and in such cases, one would use `PRIORITY_BALANCED_POWER_ACCURACY` or, if city level is sufficient, there is `PRIORITY_LOW_POWER`. There is also `PRIORITY_NO_POWER`, which will attempt to produce the best possible accuracy with no additional power consumption.

We also set the map to satellite type. We could also have used others, such as `MAP_TYPE_TERRAIN`, `MAP_TYPE_HYBRID`, or `MAP_TYPE_NORMAL`, depending on the purpose of our app.

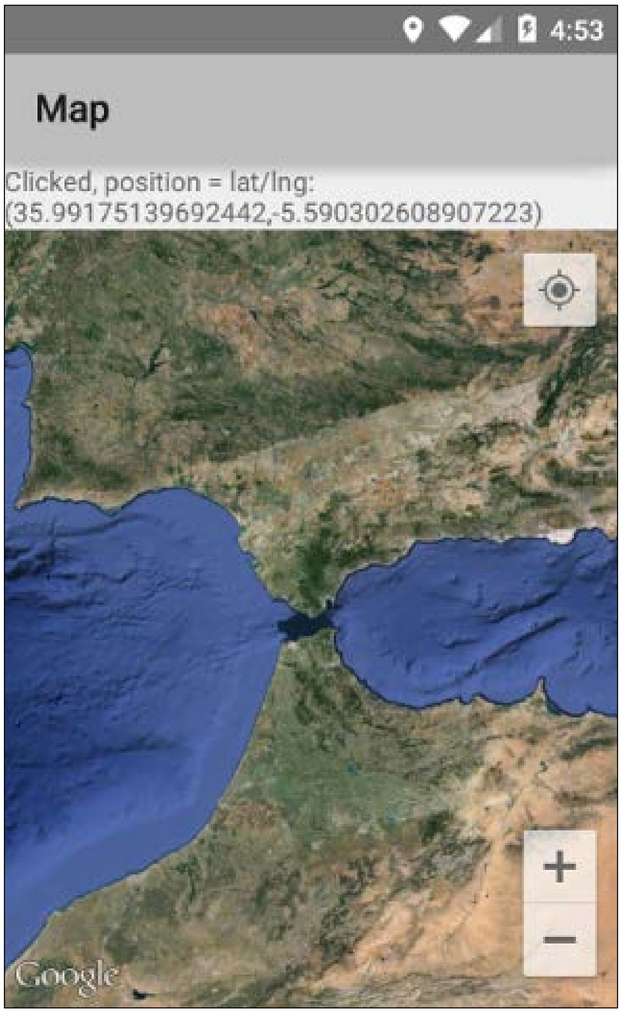
Google provides a very simple way to zoom in on our location with the `MyLocationButton`. However, there may well be times when we want to zoom to another location, not zoom at all, or even zoom out. How simple this is to do is demonstrated in the next section, along with how to determine location from a click on the map.

Moving around and animating a Google Map

The final section of this chapter requires very little in the way of coding. Google provides a specialized click listener for maps and a sort of callback method that we are very familiar with. Here, we will use this to zoom to any point on the map that is clicked.

1. Open the `MapsActivity` file.
2. In the declaration, implement this interface:
   ```
   GoogleMap.OnMapClickListener
   ```
3. Next add the following line to the `setUpMap()` method:
   ```
   mMap.setOnMapClickListener(this);
   ```
4. Create a method called `onMapClick()` and complete it like so:
   ```
   public void onMapClick(LatLng latLng) {
     textView.setText("Clicked, position = " + latLng);
     CameraPosition position = new CameraPosition.Builder()
       .target(latLng)
       .zoom(6)
       .build();
     mMap.animateCamera(CameraUpdateFactory.newCameraPosition(
       position));
   }
   ```

5. That's all there is to it. You can now run the app, which will zoom in on any point that is clicked.



Clicked, position = lat/lng: (35.99175139692442,-5.590302608907223)

The use of the `GoogleMap.OnMapClickListener` is more or less self-explanatory and provides us with the location in the form of a `LatLng` object without us having to do any extra work. The zoom level is all that needs an explanation, which too is simple. There are 12 levels with 12 being street level and 1 showing the entire planet. Being able to obtain locations in this manner opens up the way to all number of useful and interesting functions we can include in our apps.

Summary

With mobile devices becoming ever more present, being able to include maps in our apps is essential. The Google Maps API makes this task remarkably simple and probably the only complicated part was obtaining the API Key and setting up the permissions in the manifest file.

Developing with maps is further simplified by being able to set mock locations for real and emulated devices and also including UI components that users of Android devices have become used to.

In the next chapter, we will delve into what is probably one of the most exciting aspects of Android 5: the ability to develop for wearables, TVs, and cars.

8
Apps for TVs, Cars, and Wearables

One of the most exciting new directions that Android has gone in recently, is the extension of the platform from phones and tablets to televisions, car dashboards and wearables such as watches. These new devices allow us to provide added functionality to our existing apps, as well as creating wholly original apps designed specifically for these new environments.

We have already acquired the skills needed to develop such apps and this chapter is really more concerned with explaining the idiosyncrasies of each platform and the guidelines Google is keen for us to follow. This is particularly vital when it comes to developing apps that people will use when driving, as safety has to be of prime importance. There are also certain technical issues that need to be addressed when developing wearable apps, such as the pairing of the device with a handset and the entirely different UI and methods of use.

In this chapter, you will:

- Create wearable AVDs
- Connect a wearable emulator to a handset with adb commands
- Connect a wearable emulator to a handset emulator
- Create a project with both mobile and wearable modules
- Use the Wearable UI Library
- Create shape-aware layouts
- Create and customize cards for wearables
- Understand wearable design principles
- Access wearable sensors

- Make an app available for Google TV
- Include Leanback support
- Understand Android Auto safety guidelines
- Configure an Auto project
- Install Google simulators
- Send SMS messages using the Android Device Monitor

Android Wear

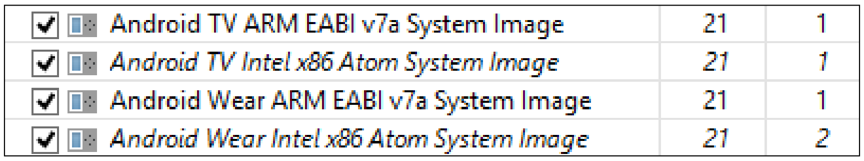
Creating or adapting apps for wearables is probably the most complicated of the three form factors dealt with in this chapter and requires a little more setting up than the other projects. However, wearables often give us access to one of the more fun new sensors, the heart rate monitor. In seeing how this works, we also get to see, how to manage sensors in general.

Do not worry if you do not have access to an Android wearable device, as we will be constructing AVDs. You will ideally have an actual Android 5 handset, if you wish to pair it with the AVD. If you do not, it is still possible to work with two emulators but it is a little more complex to set up. Bearing this in mind, we can now prepare our first wearable app.

Constructing and connecting to a wearable AVD

It is perfectly possible to develop and test wearable apps on the emulator alone, but if we want to test all wearable features, we will need to pair it with a phone or a tablet. The next exercise assumes that you have an actual device. If you do not, still complete tasks 1 through 4 and we will cover how the rest can be achieved with an emulator a little later on.

1. Open Android Studio. You do not need to start a project at this point.

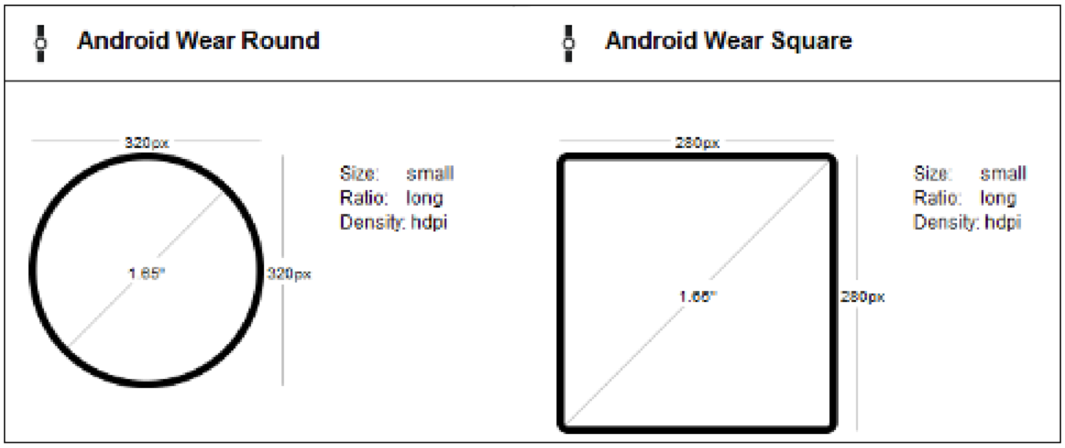


✓	Android TV ARM EABI v7a System Image	21	1
✓	*Android TV Intel x86 Atom System Image*	*21*	*1*
✓	Android Wear ARM EABI v7a System Image	21	1
✓	*Android Wear Intel x86 Atom System Image*	*21*	*2*

Start the SDK Manager and ensure you have the relevant packages installed.

2. Open the AVD Manager.

3. Create two new Android Wear AVDs, one round and one square, like so:



4. Ensure USB Debugging is selected on your handset.
5. Install the Android Wear app from the Play Store at this URL: `https://play.google.com/store/apps/details?id=com.google.android.wearable.app`. Connect it to your computer and start one of the AVDs we just created.
6. Locate and open the folder containing the `adb.exe` file. It will probably be something like `user\AppData\Local\Android\sdk\platform-tools\`.
7. Using *Shift* + right-click, select **Open command window here**.
8. In the command window, issue the following command:
   - `adb -d forward tcp:5601 tcp:5601`
9. Launch the companion app and follow the instructions to pair the two devices.

Being able to connect a real-world device to an AVD is a great way to develop form factors without having to own the devices. The wearable companion app simplifies the process of connecting the two. If you have had the emulator running for any length of time, you will have noticed that many actions, such as notifications, are sent to the wearable automatically. This means that very often our apps will link seamlessly with a wearable device, without us having to include code to pre-empt this.

The `adb.exe` (**Android Debug Bridge**) is a vital part of our development toolkit. Most of the time, the Android Studio manages it for us. However, it is useful to know that it is there and a little about how to interact with it. We used it here to manually open a port between our wearable AVD and our handset.

There are many `adb` commands that can be issued from the command prompt and perhaps the most useful is `adb devices`, which lists all currently debuggable devices and emulators, and is very handy when things are not working, to see if an emulator needs restarting. Switching the ADB off and on can be achieved using `adb kill-server` and `adb start-server` respectively. Using `adb help` will list all available commands.

> The port forwarding command we used in step 10, needs to be issued every time the phone is disconnected from the computer.

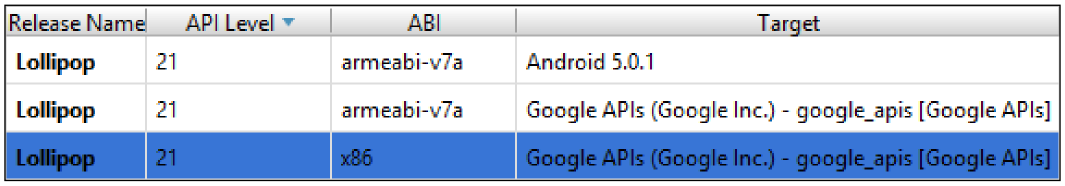
Without writing any code as such, we have already seen some of the features that are built into an Android Wear device and the way that the Wear UI differs from most other Android devices.

Even if you usually develop with the latest Android hardware, it is often still a good idea to use an emulator, especially for testing the latest SDK updates and pre-releases. If you do not have a real device, then the next, small section will show you how to connect your wearable AVD to a handset AVD.

Connecting a wearable AVD with another emulator

Pairing two emulators is very similar to pairing with a real device. The main difference is the way we install the companion app without access to the Play Store. Follow these steps to see how it is done:

1. Start up, an AVD. This will need to be targeting Google APIs as seen here:



Release Name	API Level	ABI	Target
Lollipop	21	armeabi-v7a	Android 5.0.1
Lollipop	21	armeabi-v7a	Google APIs (Google Inc.) - google_apis [Google APIs]
Lollipop	21	x86	Google APIs (Google Inc.) - google_apis [Google APIs]

2. Download the `com.google.android.wearable.app-2.apk`. There are many places online where it can be found with a simple search, I used `www.file-upload.net/download`.
3. Place the file in your `sdk/platform-tools` directory.
4. *Shift* + right-click in this folder and select **Open command window here**.
5. Enter the following command:

```
adb install com.google.android.wearable.app-2.apk.
```

6. Start your wearable AVD.
7. Enter `adb devices` into the command prompt, making sure that both emulators are visible with an output similar to this:

```
List of devices attached
emulator-5554   device
emulator-5555   device
```

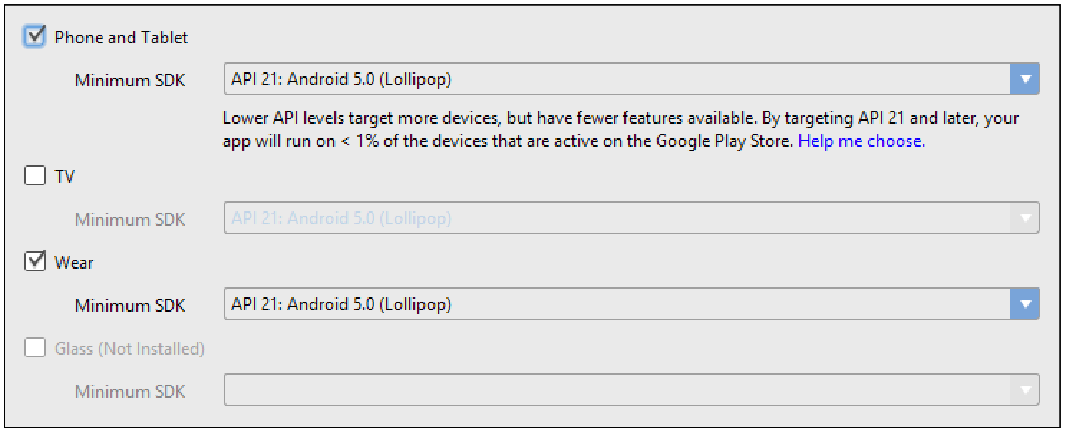
8. Enter `adb telnet localhost 5554` at the command prompt, where `5554` is the phone emulator.
9. Next, enter `adb redir add tcp:5601:5601`.
10. You can now use the Wear app on the handheld AVD to connect to the watch.

As we've just seen, setting up a Wear project takes a little longer than some of the other exercises we have performed. Once set up though, the process is very similar to that of developing for other form factors, and something we can now get on with.

Creating a wearable project

All of the apps that we have developed so far, have required just a single module, and this makes sense as we have only been building for single devices. In this next step, we will be developing across two devices and so will need two modules. This is very simple to do, as you will see in these next steps.

1. Start a new project in the Android Studio and call it something like `Wearable App`.
2. On the **Target Android Devices** screen, select both **Phone and Tablet** and **Wear**, like so:



[✓] Phone and Tablet

Minimum SDK: API 21: Android 5.0 (Lollipop)

Lower API levels target more devices, but have fewer features available. By targeting API 21 and later, your app will run on < 1% of the devices that are active on the Google Play Store. Help me choose.

[ ] TV

Minimum SDK: API 21: Android 5.0 (Lollipop)

[✓] Wear

Minimum SDK: API 21: Android 5.0 (Lollipop)

[ ] Glass (Not Installed)

Minimum SDK:

3. You will be asked to add two Activities. Select **Blank Activity** for the Mobile Activity and **Blank Wear Activity** for Wear.
4. Everything else can be left as it is.
5. Run the app on both round and square virtual devices.

The first thing you will have noticed is the two modules, mobile and wear. The first is the same as we have seen many times, but there are a few subtle differences with the wear module and it is worth taking a little look at. The most important difference is the `WatchViewStub` class. The way it is used can be seen in the `activity_main.xml` and `MainActivity.java` files of the wear module.

This frame layout extension is designed specifically for wearables and detects the shape of the device, so that the appropriate layout is inflated. Utilizing the `WatchViewStub` is not quite as straightforward, as one might imagine, as the appropriate layout is only inflated after the `WatchViewStub` has done its thing. This means that, to access any views within the layout, we need to employ a special listener that is called once the layout has been inflated. How this `OnLayoutInflatedListener()` works can be seen by opening the `MainActivity.java` file in the wear module and examining the `onCreate()` method, which will look like this:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  final WatchViewStub stub = (WatchViewStub)
    findViewById(R.id.watch_view_stub);
  stub.setOnLayoutInflatedListener(new
    WatchViewStub.OnLayoutInflatedListener() {
    @Override
    public void onLayoutInflated(WatchViewStub stub) {
      mTextView = (TextView) stub.findViewById(R.id.text);
    }
  });
}
```

Other than the way that wearable apps and devices are set up for developing, the other significant difference is the UI. The widgets and layouts that we use for phones and tablets are not suitable, in most cases, for the diminished size of a watch screen. Android provides a whole new set of UI components, that we can use and this is what we will look at next.

Designing a UI for wearables

As well as having to consider the small size of wearable when designing layouts, we also have the issue of shape. Designing for a round screen brings its own challenges, but fortunately the **Wearable UI Library** makes this very simple. As well as the `WatchViewStub`, that we encountered in the previous section that inflates the correct layout, there is also a way to design a single layout that inflates in such a way, that it is suitable for both square and round screens.

Designing the layout

The project setup wizard included this library for us automatically in the `build.gradle (Module: wear)` file as a dependency:

```
compile 'com.google.android.support:wearable:1.1.0'
```

The following steps demonstrate how to create a shape-aware layout with a `BoxInsetLayout`:

1. Open the project we created in the last section.
2. You will need three images that must be placed in the `drawable` folder of the wear module: one called `background_image` of around 320 x 320 px and two of around 50 x 50 px, called `right_icon` and `left_icon`.
3. Open the `activity_main.xml` file in the wear module.
4. Replace its content with the following code:

   ```
   <android.support.wearable.view.BoxInsetLayout
     xmlns:android=
       "http://schemas.android.com/apk/res/android"
     xmlns:app="http://schemas.android.com/apk/res-auto"
     android:background="@drawable/background_image"
     android:layout_height="match_parent"
     android:layout_width="match_parent"
     android:padding="15dp">

   </android.support.wearable.view.BoxInsetLayout>
   ```

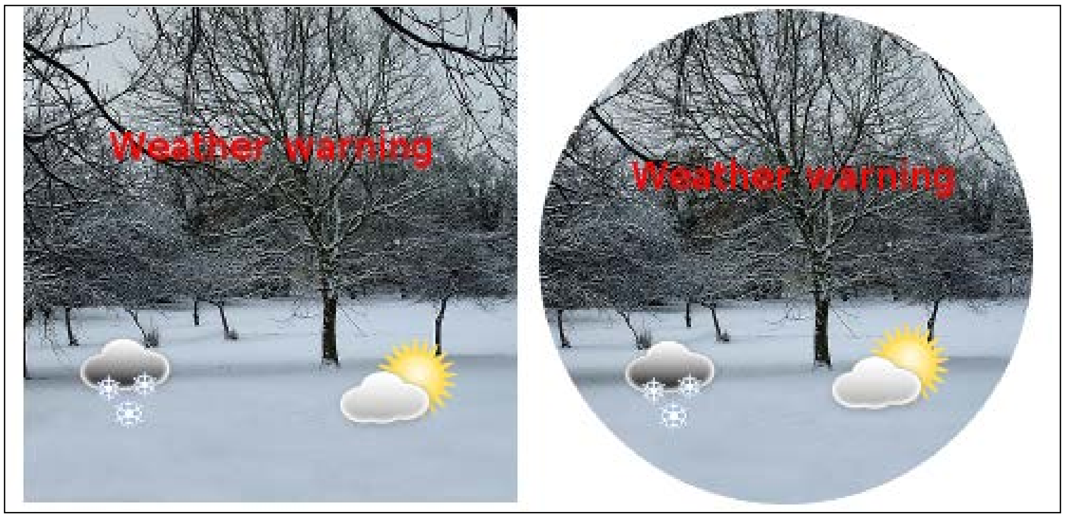
5. Inside the `BoxInsetLayout`, add the following `FrameLayout`:

   ```
   <FrameLayout
     android:id="@+id/wearable_layout"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:padding="5dp"
     app:layout_box="all">

   </FrameLayout>
   ```

6. Inside this, add these three views:

```
<TextView
  android:gravity="center"
  android:layout_height="wrap_content"
  android:layout_width="match_parent"
  android:text="Weather warning"
  android:textColor="@color/black" />

<ImageView
  android:layout_gravity="bottom|left"
  android:layout_height="60dp"
  android:layout_width="60dp"
  android:src="@drawable/left_icon" />

<ImageView
  android:layout_gravity="bottom|right"
  android:layout_height="60dp"
  android:layout_width="60dp"
  android:src="@drawable/right_icon" />
```

7. Open the `MainActivity.java` file in the wear module.
8. In the `onCreate()` method, delete all lines after the line `setContentView(R.layout.activity_main);`.
9. Now, run the app on both square and round emulators.



As we can see, the `BoxInsetLayout` does a fine job of inflating our layout regardless of screen shape. How it works is very simple. The `BoxInsetLayout` creates a square region, that is as large as can fit inside the circle of a round screen. This is set with the `app:layout_box="all"` instruction, which can also be used for positioning components, as we will see in a minute.

We have also set the padding of the `BoxInsetLayout` to 15 dp and that of the `FrameLayout` to 5 dp. This has the effect of a margin of 5 dp on round screens and 15 dp on square ones.

Whether you use the `WatchViewStub` and create separate layouts for each screen shape or `BoxInsetLayout` and just one layout file depends entirely on your preference and the purpose and design of your app. Whichever method you choose, you will no doubt want to add Material Design elements to your wearable app, the most common and versatile of these being the card. In the following section, we will explore the two ways that we can do this, the `CardScrollView` and the `CardFragment`.

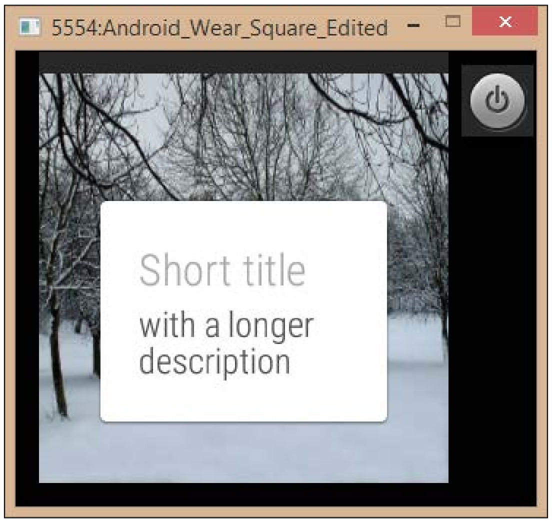
Adding cards

The `CardFragment` class provides a default card view, providing two text views and an image. It is beautifully simple to set up, has all the Material Design features such as rounded corners and a shadow, and is suitable for nearly all purposes. It can be customized, as we will see, although the `CardScrollView` is often a better option. First, let us see, how to implement a default card for wearables:

1. Open the `activity_main.xml` file in the wear module of the current project.
2. Delete or comment out the the text view and two image views.
3. Open the `MainActivity.java` file in the wear module.
4. In the `onCreate()` method, add the following code:

```
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction = fragmentManager.
beginTransaction();
CardFragment cardFragment = CardFragment.create("Short title",
"with a longer description");
fragmentTransaction.add(R.id.wearable_layout, cardFragment);
fragmentTransaction.commit();
```

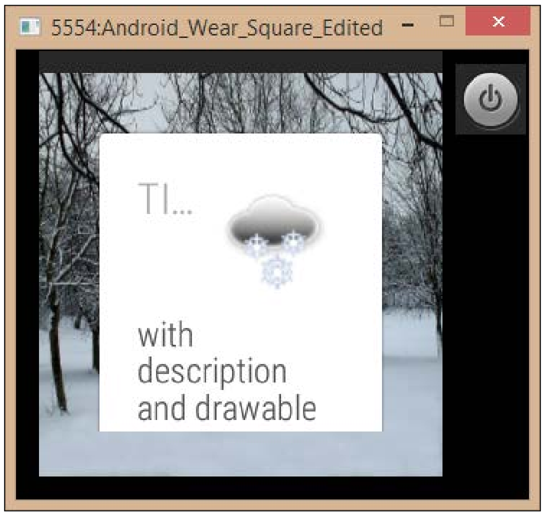
5. Run the app on one or other of the wearable emulators to see how the default card looks.



Short title

with a longer description

We met the `FragmentManager` in *Chapter 6*, *Notifications and the Action Bar* and here it operates in a very similar fashion and requires little explanation. The way we created the `CardFragment` itself, is also very straightforward. We used two string parameters here, but there is a third, drawable parameter and if the line is changed to `CardFragment cardFragment = CardFragment.create("TITLE", "with description and drawable", R.drawable.left_icon);` then we will get the following output:



TI...

with description and drawable

This default implementation for cards on wearable is fine for most purposes and it can be customized by overriding its `onCreateContentView()` method. However, the `CardScrollView` is a very handy alternative, and this is what we will look at next.

Customizing cards

The `CardScrollView` is defined from within our layout and furthermore it detects screen shape and adjusts the margins to suit each shape. To see how this is done, follow these steps:

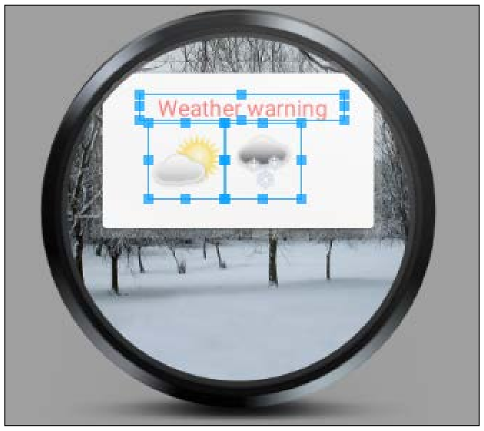
1. Open the `activity_main.xml` file in the wear module.
2. Delete or comment out every element, except the root `BoxInsetLayout`.
3. Place the following `CardScrollView` inside the `BoxInsetLayout`:

```
<android.support.wearable.view.CardScrollView
  android:id="@+id/card_scroll_view"
  android:layout_height="match_parent"
  android:layout_width="match_parent"
  app:layout_box="bottom">

</android.support.wearable.view.CardScrollView>
```

4. Inside this, add this `CardFrame`:

```
<android.support.wearable.view.CardFrame
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

</android.support.wearable.view.CardFrame>
```

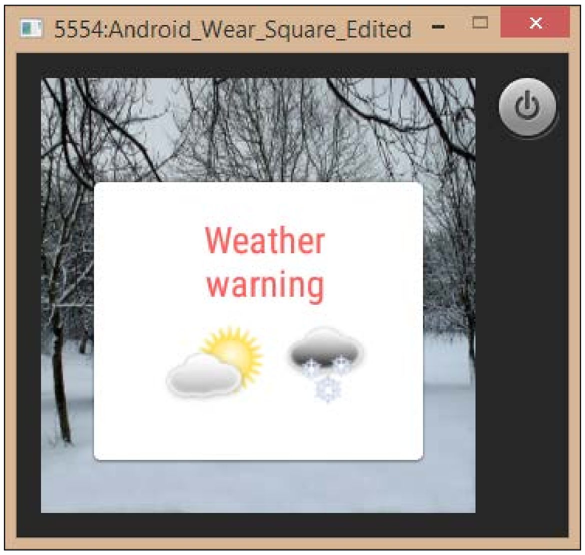
5. Inside the `CardFrame`, add a `LinearLayout`.
6. Add some views to this, so that the preview resembles the layout here:



7. Open the `MainActivity.java` file.
8. Replace the code we added to the `onCreate()` method with this:

```
CardScrollView cardScrollView = (CardScrollView)
  findViewById(R.id.card_scroll_view);
cardScrollView.setCardGravity(Gravity.BOTTOM);
```

9. You can now test the app on an emulator, which will produce the following result:



As can be seen in the previous image, the Android Studio has preview screens for both wearable shapes. Like some other previews, these are not always what you will see on a device, but they allow us to put layouts together very quickly, by dragging and dropping widgets.

As we can see, the `CardScrollView` and `CardFrame` are even easier to implement than the `CardFragment` and also far more flexible, as we can design almost any layout we can imagine. We assigned `app:layout_box` here again, only this time using `bottom`, causing the card to be placed as low on the screen as possible.

It is very important, when designing for such small screens, to keep our layouts as clean and simple as possible. Google's design principles state that wearable apps should be glanceable. This means that, as with a traditional wrist watch, the user should be able to glance at our app and immediately take in the information and return to what they were doing.

Another of Google's design principle—*Zero to low interaction*—is only a single tap or swipe a user needs to do to interact with our app. With these principles in mind, let us create a small app, with some actual functionality. In the next section, we will take advantage of the new heart rate sensor found in many wearable devices and display current beats-per-minute on the display.

Accessing sensor data

The location of an Android Wear device on the user's wrist, makes it the perfect piece of hardware for fitness apps, and not surprisingly, these apps are immensely popular. As with most features of the SDK, accessing sensors is pleasantly simple, using classes such as managers and listeners and requiring only a few lines of code, as you will see by following these steps:

1. Open the project we have been working on in this chapter.
2. Replace the background image with one that might be suitable for a fitness app. I have used a simple image of a heart.
3. Open the `activity_main.xml` file.
4. Delete everything, except the root `BoxInsetLayout`.
5. Place this `TextView` inside it:

   ```
     <TextView
       android:id="@+id/text_view"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:layout_gravity="center_vertical"
       android:gravity="center"
       android:text="BPM"
       android:textColor="@color/black"
       android:textSize="42sp" />
   ```

6. Open the Manifest file in the wear module.
7. Add the following permission inside the root manifest node:

   ```
   <uses-permission
     android:name="android.permission.BODY_SENSORS" />
   ```

8. Open the `MainActivity.java` file in the wear module.
9. Add the following fields:

   ```
   private TextView textView;
   private SensorManager sensorManager;
   private Sensor sensor;
   ```

10. Implement a `SensorEventListener` on the Activity:

    ```
    public class MainActivity extends Activity implements
      SensorEventListener {
    ```

11. Implement the two methods required by the listener.
12. Edit the `onCreate()` method, like this:

    ```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      textView = (TextView) findViewById(R.id.text_view);

      sensorManager = ((SensorManager)
        getSystemService(SENSOR_SERVICE));
      sensor =
        sensorManager.getDefaultSensor(Sensor.TYPE_HEART_RATE);
    }
    ```

13. Add this `onResume()` method:

    ```
    protected void onResume() {
      super.onResume();

      sensorManager.registerListener(this, this.sensor, 3);
    }
    ```

14. And this `onPause()` method:

    ```
    @Override
    protected void onPause() {
      super.onPause();

      sensorManager.unregisterListener(this);
    }
    ```

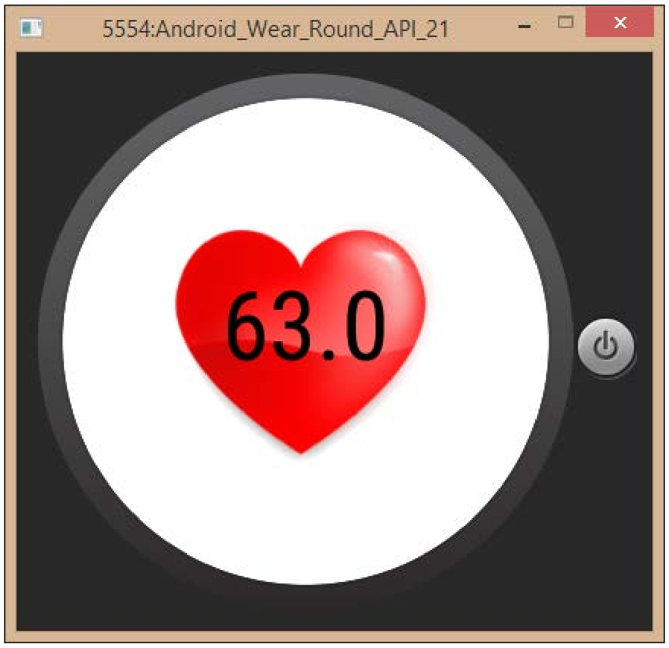
15. Edit the `onSensorChanged()` callback, like so:

    ```
    @Override
    public void onSensorChanged(SensorEvent event) {
      textView.setText("" + (int) event.values[0]);
    }
    ```

16. If you do not have access to a real device, you can download a sensor simulator from here:

    `https://code.google.com/p/openintents/wiki/SensorSimulator`

17. The app is now ready to test.



We began by adding a permission in the `AndroidManifest.xml` file in the appropriate module; this is something we have done before and need to do any time we are using features, that require the user's permission before installing.

The inclusion of a background image may seem necessary, but an appropriate background is a real aid to glancability as the user can tell instantly which app they are looking at.

It should be clear, from the way the `SensorManager` and the `Sensor` are set up in the `onCreate()` method, that all sensors are accessed in the same way and different sensors can be accessed with different constants. We used `TYPE_HEART_RATE` here, but any other sensor can be started with the appropriate constant, and all sensors can be managed with the same basic structures as we found here, the only real difference being the way each sensor returns `SensorEvent.values[]`. A comprehensive list of all sensors, and descriptions of the values they produce can be found at `http://developer.android.com/reference/android/hardware/Sensor.html`.

As with any time our apps utilize functions that run in the background, it is vital that we unregister our listeners, whenever they are no longer needed, in our Activity's `onPause()` method. We didn't use the `onAccuracyChanged()` callback here, but its purpose should be clear and there are many possible apps where its use is essential.

This concludes our exploration of wearable apps and how they are put together. Such devices continue to become more prevalent and the possibility of ever more imaginative uses is endless. Providing we consider why and how people use smart watches, and the like, and develop to take advantage of the location of these devices by programming glanceable interfaces that require the minimum of interactivity, Android Wear seems certain to grow in popularity and use, and the developers will continue to produce ever more innovative apps.

Android TV

At the opposite end of the size spectrum to Android Wear, is Android TV. Like Wear, the size of this form factor is all important, when it comes to how we design apps for it. The main consideration is the distance the user is from the screen, which is usually around 10 feet. This means designing simple and clean layouts and avoiding small and/or lengthy text.

Unlike Wear, many of the apps we design for phones and tablets can be made available for TVs as well. As one would imagine, this requires some fiddling around with the manifest, so as to make our apps visible to users searching for TV-specific apps in the Google Play store. Also, TVs do not have many of the functions that our phones do, such as GPS and touchscreens, and we need to take this into consideration as well.

The TV app template generated by the project wizard in Android Studio has bugs, and unless Google have fixed it by the time you read this, it is far from straightforward to use it to generate a working app. It is still worth taking a look though, as the Java directory contains over a dozen purpose-built classes that are very handy for apps designed for streaming and broadcasting television shows and the like.

The fact that the template does not work, is actually a good thing in our case, as we can use this section to see how to build a TV-compatible app from scratch or, if you prefer, how to convert an already developed app so that it is available to install on an Android TV.

The following exercise can be carried out using a blank activity template for a phone and a tablet or any app you have developed.

1. Open your project's manifest file.
2. Inside the root manifest node, add these permissions:

```
<uses-permission android:name="android.permission.INTERNET"
  />

<uses-permission
  android:name="android.permission.RECORD_AUDIO" />
```

3. In the same node, add these feature uses:

```
<uses-feature
  android:name="android.hardware.touchscreen"
  android:required="false" />

<uses-feature
  android:name="android.software.leanback"
  android:required="false" />

<uses-feature
  android:name="android.hardware.microphone"
  android:required="false" />

<uses-feature
  android:name="android.hardware.screen.portrait"
  android:required="false" />
```

4. Find or create a 320 x 180 px `xhdpi` banner image to represent your app. Ideally, it should contain text as well as an identifiable image, such as the following:

Android 5
programming by example

5. Place the image in your `drawable` folder, and call it `banner`.
6. Add the following line inside the application node:

```
android:banner="@drawable/banner"
```

7. Open your `build.gradle` file and add this dependency:

```
compile 'com.android.support:leanback-v17:22.0.0'
```

8. Back in the manifest, change the theme declaration line to the following:

```
android:theme="@style/Theme.Leanback"
```

9. Create a TV AVD and test the app.

Although, we have not had to do very much here, there is still quite a lot to explain. Most of what we have done, is to make sure that our app is visible when browsing the Play store for TV apps. TVs do not support portrait screen layouts, so if this feature is required, it will simply not appear in the Play store as available for TV. We want to include the feature for devices that do support this orientation, and this is how we do that and still make our app available to TV users. We have to include the permission to record audio with all TV apps, but TVs do not generally support microphones.

The **Leanback Support Library** that we added in step 7 is a very useful tool for developing for TV. It provides a very suitable theme, `Theme.Leanback`, several useful widgets designed for TV, and it manages margins in such a way that our layout is not clipped. Without it, we would have to set wide margins of about 10 percent to avoid this kind of TV over-scanning. If we wanted to write specifically for TV, we would also have changed the category inside our main activity's intent filter to `<category android:name="android.intent.category.LEANBACK_LAUNCHER" />`.

There is quite a lot more to programming for Android TV, especially when it comes to broadcasting and streaming, that we cannot cover here. Generally speaking though, developing TV apps requires the same skills as programming for handheld devices and many apps run perfectly well on both formats. Providing we take into account the distance the user is from the screen and the limited input methods, we can write programs that provide a satisfying experience across all platforms.

The Android OS is a very flexible system and well suited to a wide variety of form factors. We've seen how it can run on screens as small as an inch and a half or as large as a home cinema. Android Lollipop heralded yet another new and exciting platform, allowing users to run Android apps when they are out in their cars.

Android Auto

Android Auto apps are apps that, when connected to a compatible car dashboard, run certain restricted content on the driver's dashboard. When developing for cars, the primary concern has to be safety and no Auto app that does not meet these strict standard will not be published on the Play store. Many Android apps are far too distracting to be suitable for use when driving. In fact, Android Auto really only supports two functions: audio playback and text-to-speech messaging. It is beyond the scope of this book to explore this comprehensively, but this is certainly a good time to see how such apps are set up and how to use the media and messaging simulators provided in the SDK. First though, we need to take a look at the safety rules insisted on by Google for Auto apps:

- There must be no animated elements on the Auto screen
- Only audio ads are allowed

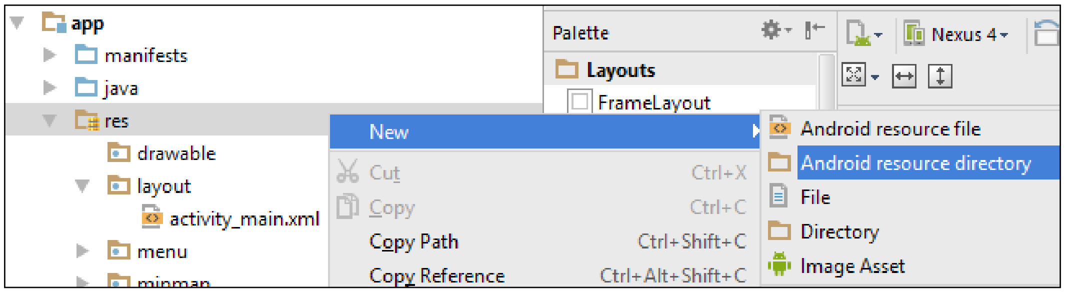
- Apps must support voice control
- All buttons and clickable controls must respond within two seconds
- Text must be longer than 120 characters and must always be in the default Roboto font
- Icons must be white, so that the system can control contrast
- Apps must support day and night mode
- App must support voice commands
- App-specific buttons must respond to user actions with no more than a two-second delay

IMPORTANT: These, and several other stipulations, will be tested by Google before publication, so it is essential that you run all of these tests yourself. The full list can be found at `http://developer.android.com/distribute/essentials/quality/auto.html`.

[Designing apps that are suitable for day and night modes and that can have contrast controlled by the system to automatically remain readable in different light conditions, is quite a detailed subject and Google has produced a very useful guide to this, which can be found at `http://commondatastorage.googleapis.com/androiddevelopers/shareables/auto/AndroidAuto-custom-colors.pdf`.]

Despite the restrictions we have covered Auto apps are developed in the same way as any other. There is one minor difference though, when developing for Auto, we need to define which in-car capabilities our app is using. This is done using an XML file. Follow these short steps, to see how:

1. Start a new Android Studio project for phone and tablet, or open an existing one.
2. Create a new **Android resource directory** inside the `res` directory and call it `xml`.



3. Inside this `xml` folder, create a new Android resource file and call it something like `auto_config.xml`.
4. Complete it as follows:

```
<?xml version="1.0" encoding="utf-8"?>

  <automotiveApp>
    <uses name="media" />
    <uses name="notification" />
  </automotiveApp>
```

5. Open the manifest file.
6. Add these lines inside the `application` node:

```
<meta-data
  android:name="com.google.android.gms.car.application"
  android:resource="@xml/auto_config"/>
```

7. That's it, our app will now detect whether its host device is connected to a car dashboard.

There are just two things here that need to be pointed out here. The two named uses are for running audio playback apps and receiving messages respectively. We would only need both, if our app was designed for both functions.

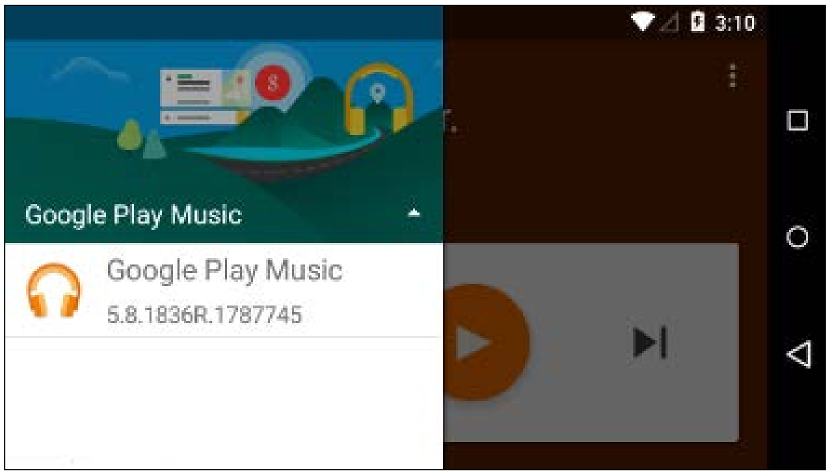
Google provide simulators for testing both media browsing and messaging apps, letting us test projects from the safety of a desk. The following steps demonstrate how to install them and use a telnet connection to send dummy SMS messages:

1. From Android studio, open the SDK Manager.
2. Make sure you have the latest versions of the Android Auto API Simulators, which are in the `Extras` folder.
3. Connect a device or an emulator.
4. Go to your `sdk/extras/google/simulators` folder and open the command window there, with *Shift* + right-click.
5. Check that your device is connected with `adb devices`.
6. Enter these two commands to install both simulators:

```
adb install media-browser-simulator.apk
adb install messaging-simulator.apk
```

> If you are using a third party virtual device, such as Genymotion, you will be able to install these apps by dragging and dropping them onto the emulator's screen.

Installing the simulators with an `adb` command is very simple, and of course any `.apk` file can be installed this way to a connected device. The media browser simulator can be tested with most media services such as Play Music.



Text can be sent to the messaging simulator by launching the **Android Device Monitor** from the **Emulator Control** tab.

That's about as much as we can cover of Android Auto here. The platform provides some of the most exciting new possibilities offered by Lollipop, and no doubt Android will be found in more and more vehicles in the future.

Summary

Android Wear, TV, and Auto represent radical departures from traditional form factors and are equally different from each other. This means that we have had to cover a lot of varied ground here.

Despite their diminutive size and functionality, wearables offer us an enormous range of possibilities. We know now how to create and connect wearable AVDs and how to develop easily for both square and round devices. We then went on to see what is required to set up a TV app, how to convert existing apps to be available on TV and about the useful library and features provided by Leanback support. We concluded by exploring the stringent safety rules that must be applied when developing and the tools available for testing Auto on.

One of the biggest overhauls, although perhaps less visible to the user, involves the camera APIs. These are entirely new to Lollipop and these along with adding multimedia to our apps is what we will move on to in the next chapter.

9

Camera, Video, and Multimedia

Recent years have brought about great advancements in mobile multimedia technologies, with many people not only listening to music and watching movies on their mobile devices but using them to produce high quality media of their own. The SDK provides APIs that allow us to include both media playback as well as media capture, and with the camera APIs being completely overhauled, there has never been a better time to be developing Android multimedia apps.

Many multimedia features can be very easily incorporated into our apps by simply harnessing the system's native applications such as the camera. Alternatively, we can work directly with the APIs and develop apps that handle all the photos and videos capturing process themselves, although this is no simple task. One thing that is simple to achieve, though, is including the recording and playing of multimedia, including audio, within our apps.

In this chapter, you will:

- Preview images using the native Camera app
- Automatically refactor code
- Save images from the native camera to our app
- Handle IO exceptions
- Create a unique filename
- Add images to the device gallery
- Make images private
- Capture and play back video
- Add video controls

- Handle video interruptions without losing position
- Package videos with an app
- Play videos from memory
- Stream video from the Web
- Record audio files with a MediaRecorder
- Play back audio files with a MediaPlayer

Capturing images

More often than not, when including image or video capture in our apps, all we need to do is take advantage of the fact that the system already has applications designed for these purposes and we can call upon them with an Intent, in just the same way that we call Activities inside our own apps. We do not even need to know which app is called, as the system will automatically seek out the most appropriate, even offering the user a choice when one is available.

Here, we will include a photo taking function in our Ancient Britain app that harnesses the native Camera application to capture an image, display it in a view and save it to a specific directory. We will then make our image available to the device gallery and other apps. This is not a short exercise, so we will split it into three parts: preparing and refactoring, previewing a camera shot, and saving a camera shot.

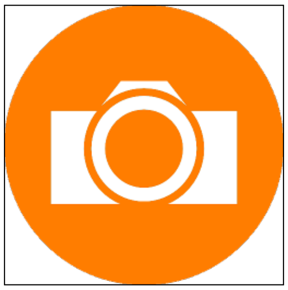
Refactoring code

To save time, we will not set up another button for our camera function. Instead we will reuse the ImageView that currently takes the user to the relevant Wikipedia page. We will also need to set some permissions and feature uses and add a new graphic. Follow these steps to prepare the Ancient Britain app, which we began in *Chapter 4, Managing RecyclerViews and Their Data* to incorporate calls to the native camera:

1. Open the `Ancient Britain` project in Android Studio and open the manifest file.
2. Include these tags:

```
<uses-feature android:name="android.hardware.camera"
  android:required="true" />

<uses-permission android:name="android.permission.CAMERA"
  />

<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  />
```

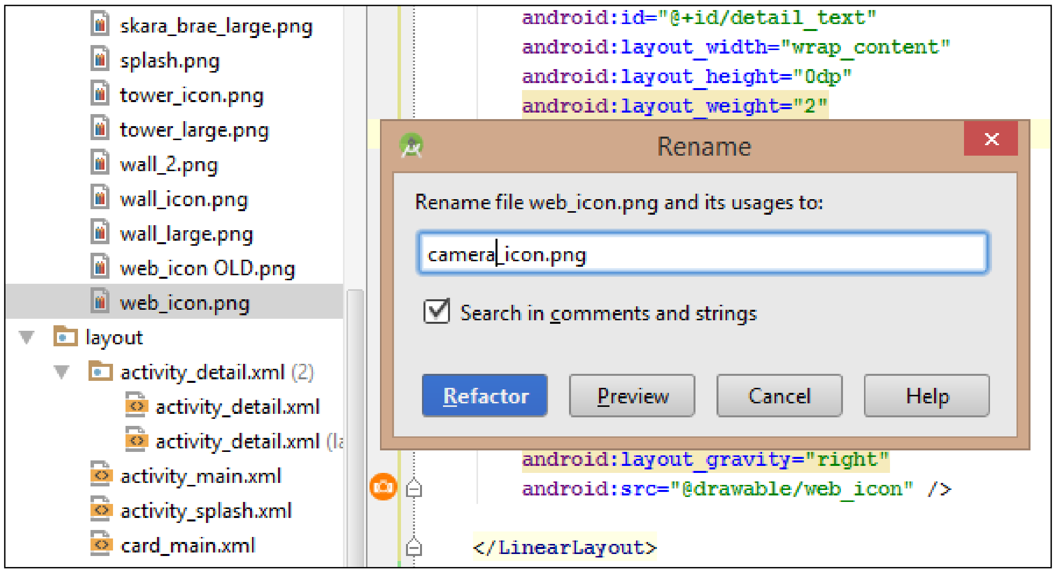
3. Find an icon-sized image, suitable for a camera function, such as the following:



4. Save this in the `res/drawable` directory in file explorer view and replace the `web_icon.png` file with the one you just made.
5. Open the `DetailActivity.java` file.
6. Locate the following line and right-click on `detailWebLink`:

```
ImageView detailWebLink = (ImageView)
  findViewById(R.id.detail_web_link);
```

7. Press *Shift* + *F6* and rename the instance `detailCameraButton`.
8. Do the same for the XML reference, renaming it `detail_camera_button`.
9. Select `web_icon` in the `drawable` folder in the project explorer and rename it `camera_icon`.



```
skara_brae_large.png
splash.png
tower_icon.png
tower_large.png
wall_2.png
wall_icon.png
wall_large.png
web_icon OLD.png
web_icon.png
layout
  activity_detail.xml (2)
    activity_detail.xml
    activity_detail.xml (la
  activity_main.xml
  activity_splash.xml
  card_main.xml
```

```
android:id="@+id/detail_text"
android:layout_width="wrap_content"
android:layout_height="0dp"
android:layout_weight="2"
```

Rename

Rename file web_icon.png and its usages to:

camera_icon.png

☑ Search in comments and strings

```
android:layout_gravity="right"
android:src="@drawable/web_icon" />

</LinearLayout>
```

The first thing we did here, was apply permissions and feature to the manifest, the feature being included here so as to prevent devices without cameras being able to find it on the Play store.

> If you are developing for API level 17 or below, you will need to add the permission: `android.permission.CAMERA`.

The refactoring that we did next was not strictly necessary but made the code easier to follow and demonstrated just how easy it is to rename things with the *F6* key. The effects propagate throughout the project, for example when we renamed an XML reference in Java, the corresponding layout file was edited accordingly, and there are many handy refactoring tools available through the Refactor menu.

Previewing the camera output

To preview the camera, we need to fire an intent that calls the native camera as well as a way of responding when the camera returns to our app. These three steps achieve that:

1. Add these fields to the `detailActivity` class:

```
private static final int PREVIEW_REQUEST_CODE = 1;
private static final int SAVE_REQUEST_CODE = 2;
private String photoPath;
private File photoFile;
```

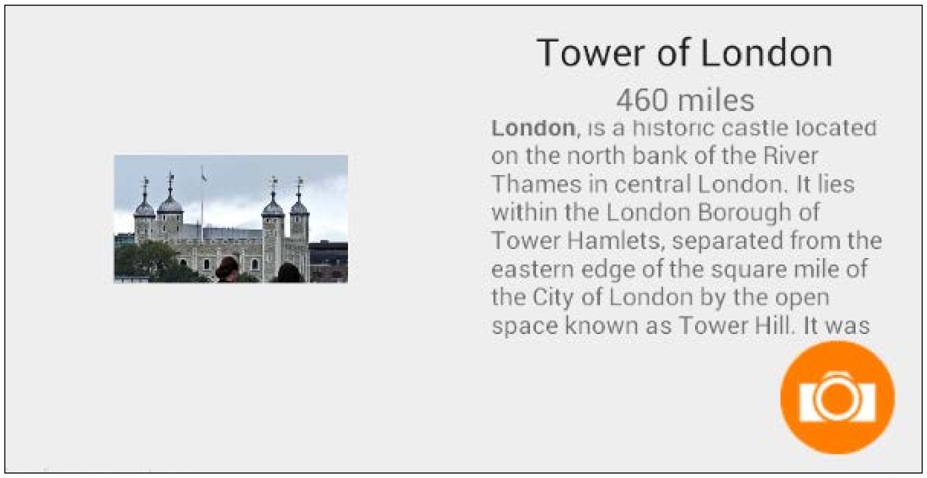
2. Replace the `onClick()` method of the now, `detailCameraButton` button's `onClickListener` with this code:

```
@Override
public void onClick(View v) {
  Intent intent = new
    Intent(MediaStore.ACTION_IMAGE_CAPTURE);
  if (intent.resolveActivity(getPackageManager()) != null) {
    startActivityForResult(intent, PREVIEW_REQUEST_CODE);
  }
}
```

3. Provide the class with the following `onActivityResult()` method:

```
  @Override
  protected void onActivityResult(int requestCode,
    int resultCode, Intent data) {
    if (requestCode == PREVIEW_REQUEST_CODE &&
      resultCode == RESULT_OK) {
            Bundle extras = data.getExtras();
            Bitmap imageBitmap = (Bitmap) extras.get("data");
            detailImage.setImageBitmap(imageBitmap);
        } else if (requestCode == SAVE_REQUEST_CODE &&
            resultCode == RESULT_OK) {
            // To complete
        }
    }
```

You can now run the app. Clicking the camera icon will allow you to take a picture which will be displayed in the layout's other `ImageView`.



London, is a historic castle located on the north bank of the River Thames in central London. It lies within the London Borough of Tower Hamlets, separated from the eastern edge of the square mile of the City of London by the open space known as Tower Hill. It was

The intent we create here is called on the `MediaStore` class, using a constant that opens the native camera. Note how the Camera Activity is protected by the intent's `resolveActivity()` method. If there is no app on the device that can fulfill the request and the intent is fired, then the app will crash. The `PacketManager()` will contain nothing if no suitable app is found.

When control is handed back to our app, the `onActivityResult()` method is called. The `requestCode` is used to check where the camera Activity was called from and the `resultCode` to test that it worked. We used the data value pair `data` to extract the bitmap the camera returned in its Bundle. This particular image is only a thumbnail. The full image is available and next we will see how to store it on the SD card.

Saving the camera output

Again to save time and extra coding, we will use an existing widget as a button to trigger an Intent to save as well as take a photo. We will replace the main image view's `onTouchListener` with an `onClickListener` and call the methods we need from there. Follow these steps to see how:

1. In the `detailActivity` class, replace the line `detailImage.setOnTouchListener(listener);` in the `onCreate()` method with this code:

```
detailImage.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        takePhoto();
    }
});
```

2. Create the `takePhoto()` method to look like this:

```
private void takePhoto() {
  Intent takePictureIntent = new
    Intent(MediaStore.ACTION_IMAGE_CAPTURE);
  if (takePictureIntent.resolveActivity(
    getPackageManager()) != null) {
    File photoFile = null;
    try {
      photoFile = filename();
    } catch (IOException ex) {
      Toast toast = Toast.makeText(getApplicationContext(),
        "No SD card",
        Toast.LENGTH_SHORT);
      toast.show();
    }
    if (photoFile != null) {
      takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
        Uri.fromFile(photoFile));
      startActivityForResult(takePictureIntent,
        SAVE_REQUEST_CODE);
    }
  }
}
```

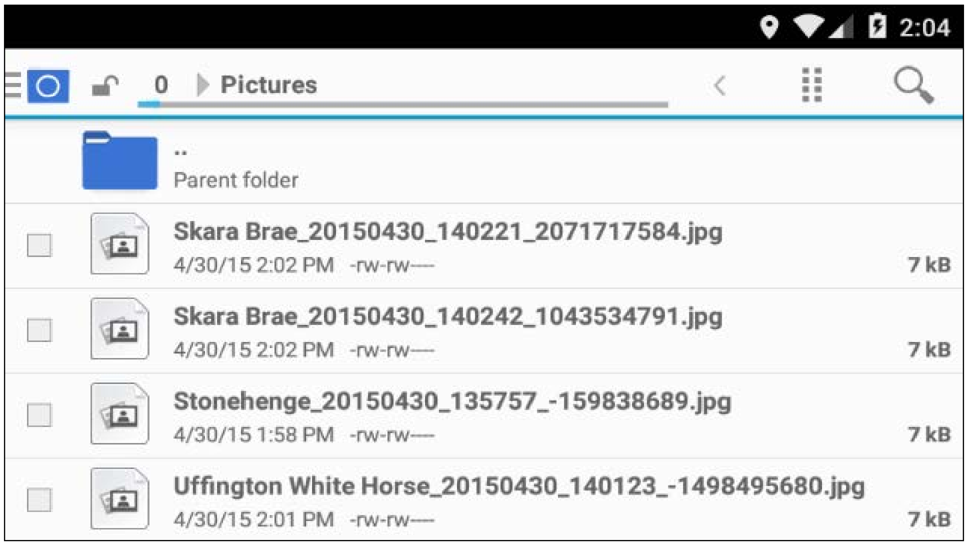
3. Include the `filename()` method like so:

```
private File filename() throws IOException {
  String time = new
    SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date());
  String file =
    MainData.nameArray[MainActivity.currentItem] +
    "_" + time + "_";
  File dir = Environment.getExternalStoragePublicDirectory(
    Environment.DIRECTORY_PICTURES);
  File image = File.createTempFile(file, ".jpg", dir);
  photoPath = "file:" + image.getAbsolutePath();
  return image;
}
```

4. In the `onActivityResult()` method, replace the commented `// To complete line with this code`:

```
else if (requestCode == SAVE_REQUEST_CODE && resultCode ==
  RESULT_OK) {
  Intent intent = new
    Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
  Uri contentUri = Uri.fromFile(photoFile);
  intent.setData(contentUri);
  this.sendBroadcast(intent);
}
```

5. You can now run and test the app. Tapping the camera icon replaces the main image with the one just taken and tapping on the image itself will let you save a picture to the device's SD card in the Pictures directory.



Skara Brae_20150430_140221_2071717584.jpg 4/30/15 2:02 PM -rw-rw--- 7 kB
Skara Brae_20150430_140242_1043534791.jpg 4/30/15 2:02 PM -rw-rw--- 7 kB
Stonehenge_20150430_135757_-159838689.jpg 4/30/15 1:58 PM -rw-rw--- 7 kB
Uffington White Horse_20150430_140123_-1498495680.jpg 4/30/15 2:01 PM -rw-rw--- 7 kB

Clearly, the way this example processes user input, is a little on the clumsy side. Ideally, we would have added new buttons or even another Activity to handle previewing and saving images. We took this approach for the sake of brevity and to highlight the processes themselves. The two methods applied warrant a little examination themselves.

The `takePhoto()` method fires the same intent as the camera button's `onClick()` method. A different request code is used to show how we can call the same external Activity but respond differently depending on where it was called from. Android generally manages exceptions rather well, but we cannot guarantee the presence of a SD card and it makes sense to try to catch this exception. We could create a message. If the creation of the File `photoFile` is successful (and it rarely isn't), it can be included in our Intent with the line `takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(photoFile));`.

When creating a filename, we need to be careful that it does not collide with others. Without a lot of precautionary code, this can be done by setting up a unique filename, which is done here by appending a timestamp. This method is called during our attempt to catch system exceptions and therefore requires the throws `IOException` declaration.

Finally, we added some code in the `else` clause of our `onActivityResult()` method, which is called after the image is saved and control returned to our app. The `ACTION_MEDIA_SCANNER_SCAN_FILE` Intent is a request to the media scanner to add it to the media database the next time it is run. This means that our image will appear in the native gallery app and be available to any other app that uses the media database, such as wallpaper pickers.

If you want your images to be only available from within your app, it is not enough simply to omit these lines, as the images are still accessible from any file browsing software. To prevent this, use `Environment.getExternalFilesDir()` instead of `Environment.getExternalStoragePublicDirectory()`. This will also have the effect of deleting these files when your app is uninstalled.

> The media scanner does not necessarily run at predictable times and, when testing, you may have to restart your device or emulator to force it to include your file.

Commandeering the platform's camera like this is a wonderfully convenient way to incorporate its functionality with a minimum of coding. Of course, it is quite possible to recreate a camera or a video app from scratch and we will take a quick look at how that can be done shortly. First, let us see how to record and play back video in the same way we did here with the camera.

Capturing and playing video

Using native apps to capture video content from within our own apps is achieved in an almost identical way, to the one we just applied. The main difference is that when dealing with video content a lot of the functionality is provided by the purpose-built widget, the `VideoView`. We will also add video control buttons with the `MediaController` and see how to pause a video when our app is sent to the background. Follow these steps to build a simple video app:

1. Start a new Android Studio project.
2. Add the feature uses and permissions we included in the last exercise to the manifest.
3. Open the `activity_main.xml` file and replace the `TextView` with this `VideoView`:

   ```
   <VideoView
     android:id="@+id/video_view"
     android:layout_width="match_parent"
     android:layout_height="match_parent" />
   ```

4. Open the `MainActivity.java` and add these fields:

   ```
   private static final int VIDEO_REQUEST_CODE = 1;
   private android.widget.VideoView videoView;
   private int position = 0;
   private MediaController mediaController;
   ```

5. Include this code in the `onCreate()` method:

   ```
   videoView = (VideoView) findViewById(R.id.video_view);

   if (mediaController == null) {
     mediaController = new MediaController(this);
   }
   videoView.setMediaController(mediaController);

   takeVideo();.
   Add the takeVideo() method, like so:
   private void takeVideo() {
     Intent takeVideoIntent = new Intent(MediaStore.ACTION_VIDEO_
   CAPTURE);
     if (takeVideoIntent.resolveActivity(getPackageManager())
       != null) {
       startActivityForResult(takeVideoIntent,
         VIDEO_REQUEST_CODE);
     }
   }
   ```

6. Then the `onActivityResult()` method:

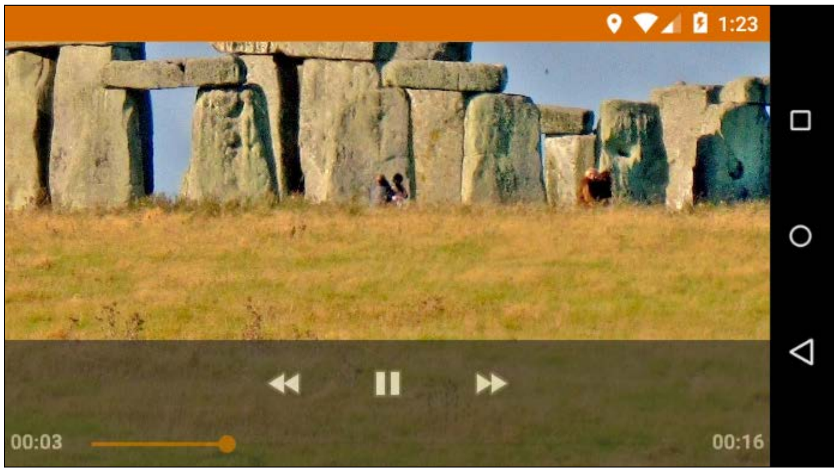
```
@Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
  if (requestCode == VIDEO_REQUEST_CODE && resultCode ==
    RESULT_OK) {
    Uri videoUri = data.getData();
    videoView.setVideoURI(videoUri);
  }
}
```

7. If you test the project now, you will be able to record and play back a video. However, if the Activity loses focus and restarts, the video will also start from the beginning.

   To rectify this, add these two methods:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
  super.onSaveInstanceState(savedInstanceState);
  savedInstanceState.putInt("Position",
    videoView.getCurrentPosition());
  videoView.pause();
}

@Override
public void onRestoreInstanceState(Bundle
    savedInstanceState) {
  super.onRestoreInstanceState(savedInstanceState);
  position = savedInstanceState.getInt("Position");
  videoView.seekTo(position);
}
```

8. If you now test the app and interrupt the playback with another app, and then return to the Activity, it will resume from where it left off.



Up until step 6, the way we dispatched our intent to capture video was almost identical to the method we used for still images, with the exception of the `MediaController`, which adds the familiar controls we all associate with video playback. When dealing with videos, particularly long ones, the user may wish to pause playback and engage with another app. To ensure that the video resumes from the last position when the user returns, we had to intercept the Activity lifecycle before the app is sent to the background with the `onSaveInstanceState()` method and again when it returns with `onRestoreInstanceState()`. We used `VideoView.pause()` and `VideoView.seekTo()` here. The following methods are available in `VideoView` for controlling video playback:

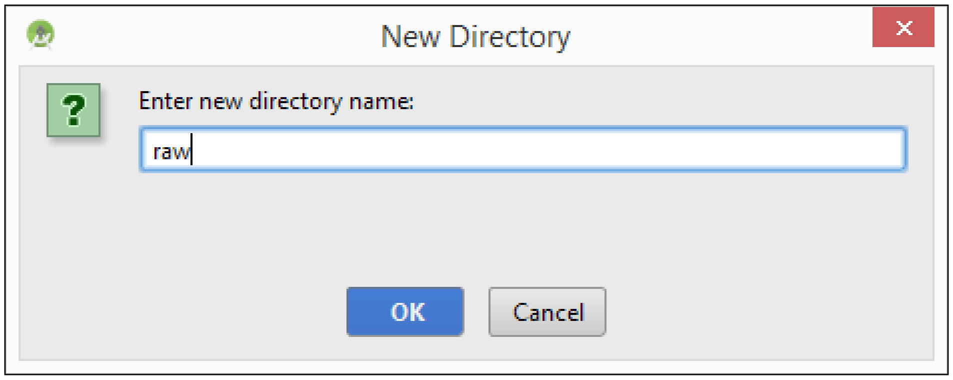
- `VideoView.start()`
- `VideoView.pause()`
- `VideoView.resume()`
- `VideoView.seekTo(position)`

As useful as it is to be able to provide video recording capabilities in our apps, there are often times when we will want to play videos packaged within our app, or from an external source such as the device SD card or even streamed from the Internet. The example above only requires a few minor adjustments and the next section shows how to adapt it to play video from other sources than the camera itself.

Playing video from memory and the Internet

There are countless reasons why we might want to include video content in our app or play videos produced by other apps, and in this section we will see how to package videos within our app and how to play videos from the device's storage and the Web. The following exercise will take you through how to do each of these:

1. Open the project that we just worked on.
2. Create a new folder in the `res` directory called `raw`.



Enter new directory name: raw

3. Find a short video file with one of the following formats, name it `movie`, and paste it into the `res/raw` folder: `.webm`, `.3gp`, `.mp4`, or `.mkv`.
4. Open the `MainActivity` file.
5. In the `onCreate()` method, comment out the call to `takePhoto()` and add these two lines:

```
videoView.setVideoURI(Uri.parse("android.resource://" +
getPackageName() + "/" + R.raw.movie));
videoView.start();
```

6. If playing an in-app video is what you are after, you can stop here and run the app.
7. To play a video stored on the device's SD card, replace the lines you just entered with these:

```
videoView.setVideoPath(
  "/sdcard/some_directory/some_movie.mp4");
videoView.start();
```

8. If run, the app will now play the indicated file from the SD card. To stream a video, use this code:

   ```
   videoView.setVideoPath(
     "http://www.your_site.com/movies/movie.mp4");
   videoView.start();
   ```

That's all there is to it. We stored our in-app video in the `res/raw` directory. Although not included, when we create a project, `raw` is a recognized resource folder and can be used for storing any file that we do not want compiled when the project is built and/or packaged.

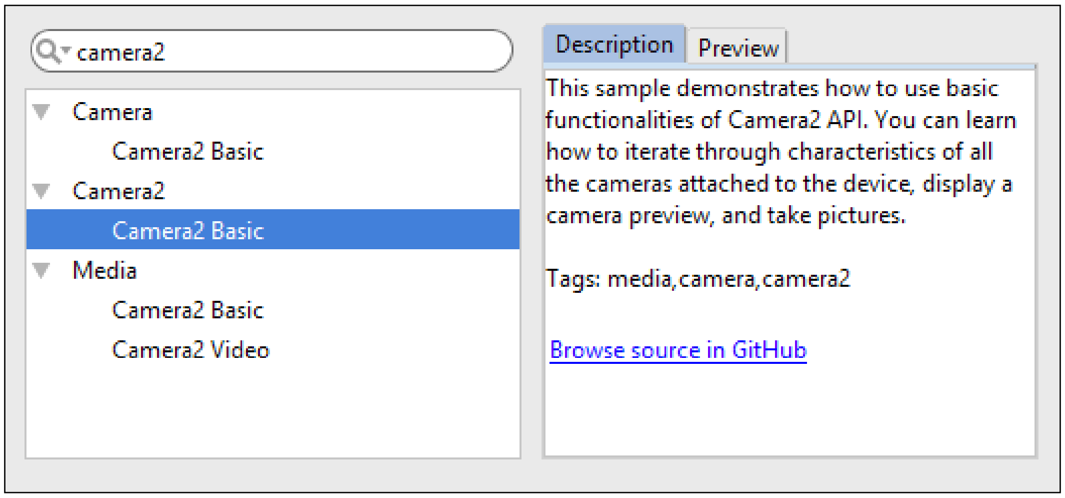
The other thing to note about this otherwise straightforward code is how we use `VideoView.setVideoPath()` when loading from internal storage or the Web, rather than `VideoView.setVideoURI()`.

Calling other apps, such as the camera app, is a very convenient way to incorporate such features without a great deal of coding. There are, of course, times when we will want a deeper integration of the camera APIs. This requires building the camera from the ground up and is beyond the scope of this chapter. However, Android 5 does introduce a whole new set of camera APIs, the `android.hardware.camera2`, which supersedes the `android.hardware.Camera` APIs. **Camera2** allows for some exciting new features, such as control over individual cameras and improved storage capabilities, and although there is no room to build a camera2 app from scratch here, there is a very informative sample, packaged with the SDK, which we will now take a look at.

Exploring the camera2 APIs

The camera2 APIs are a lot more sophisticated than their predecessors, but they are also a lot more involved. Building a camera app from scratch is far from simple. Fortunately, Android packages numerous sample apps within the SDK and there is a suitable camera2 sample, that we can take a look at.

Samples can be loaded directly into the Android Studio from the Quick Start pane of the startup window **File** | **Import Sample...** from within the IDE.



This sample demonstrates how to use basic functionalities of Camera2 API. You can learn how to iterate through characteristics of all the cameras attached to the device, display a camera preview, and take pictures.

Tags: media,camera,camera2

Browse source in GitHub

Most camera2 processes begin with the `CameraManager`. This class allows us to identify and connect to any cameras attached to the device, as well as determining their properties. In the sample, the `Camera2BasicFragment` class is where most of the interesting work is done and you can see how a `CameraManager` is used in the `openCamera()` method to open a camera, and in `setUpCameraOutputs()` to acquire the camera ID and whether it is the front-facing camera using the `CameraCharateristics` class. This class can be seen in action in the `setUpCameraOutputs()` method too, where it is used to rule out the front-facing camera.

The `CameraDevice` is the class used to represent individual cameras within an app, and is used for setting up a `CaptureRequest` and a `CaptureRequestSession` for the actual process of taking photos. This also gives us control over functions such as auto focus and white balance. The `CameraCaptureSession` is where camera2 features, such as being able to take multiple images in a burst, are made available.

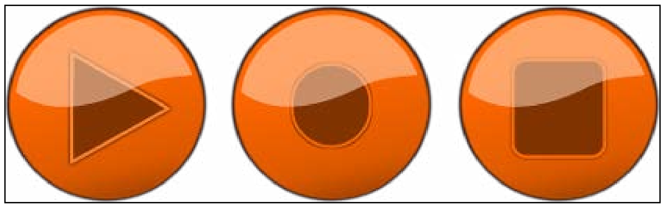
It is well worth exploring and experimenting with the `Camer2Basic` sample, and there is also a `Camera2Video` sample. If you are interested in building a camera app from scratch using Android 5 capabilities, then the official documentation at `http://developer.android.com/reference/android/hardware/camera2/package-summary`.html is well worth checking out.

The Camera2 APIs, despite their sophistication, have one serious drawback: they are the one set of Android 5 APIs that cannot be made backward compatible easily. Any app that relies largely on camera and video functions would require an necessary amount of alternative code to make it available for older platforms. Jelly Bean and KitKat current occupy over three-quarters of the market and look likely to make up a significant proportion of your target audience for a good time to come. Unless you plan to utilize camera2-specific features, such as capturing images in RAW format or taking multiple shots in a burst, you should seriously consider using the original Camera APIs, which despite having depreciated are still perfectly usable.

Recording and playing audio

We saw earlier in the chapter how to capture and play multimedia content using native apps and the `VideoView`. There is also another very handy tool for recording and playing media files, especially audio: the `MediaRecorder` class. The `MediaRecorder` allows us to simply set such things as audio source, output location, and format, as well as giving us control over play and record functions. In this exercise, we will develop a small app that records and plays back audio captured with a device's inbuilt microphone:

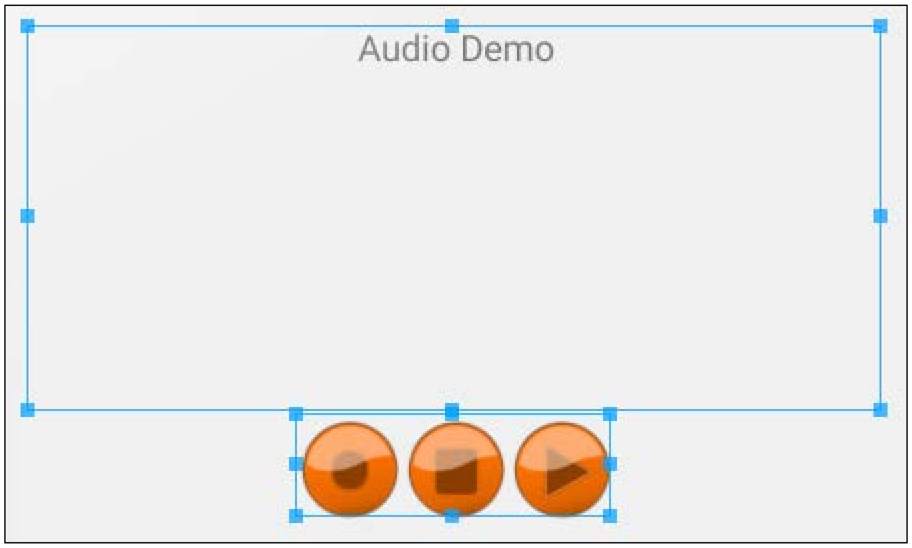
1. Start a new Android Studio project.
2. Find three button-sized media images like those below and place them in your `drawable` folder.



3. Call them Play, Record, and Stop.
4. Open the manifest files and include these permissions:

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE"
  />
<uses-permission
  android:name="android.permission.RECORD_AUDIO" />
```

5. Create a layout similar to the one here:



6. Use `ImageViews` as the buttons and give them the IDs `record_button`, `stop_button` and `play_button`. Call the `TextView`, `text_view`.

7. Open the `MainActivity` and include these two fields:

```
private MediaRecorder recorder;
private String filename;
```

8. In the `onCreate()` method, add this `TextView`:

```
final TextView textView = (TextView)
  findViewById(R.id.text_view);
```

9. Add this file path:

```
filename =
  Environment.getExternalStorageDirectory()
  .getAbsolutePath() + "/recording.3gp";
```

10. Then these `MediaRecorder` configurations:

```
recorder = new MediaRecorder();
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
recorder.setAudioEncoder(
  MediaRecorder.OutputFormat.AMR_NB);
recorder.setOutputFormat(
  MediaRecorder.OutputFormat.THREE_GPP);
recorder.setOutputFile(filename);
```

11. Each `ImageView` we created in the layout requires an `OnClickListener` within the `onCreate()` method as follows:
    - ImageView `recordButton`:

```
ImageView recordButton = (ImageView)
  findViewById(R.id.record_button);
recordButton.setOnClickListener(
  new View.OnClickListener() {

  @Override
  public void onClick(View view) {
    try {
      recorder.prepare();
    } catch (IOException e) {
    }
    recorder.start();
    textView.setText("Recording...");
  }
});
```

    - ImageView `stopButton`:

```
ImageView stopButton = (ImageView)
  findViewById(R.id.stop_button);
stopButton.setOnClickListener(new View.OnClickListener() {

  @Override
  public void onClick(View view) {
    recorder.stop();
    recorder.release();
    textView.setText("Recording complete");
  }
});
```

    - ImageView `playButton`:

```
ImageView playButton = (ImageView)
  findViewById(R.id.play_button);
playButton.setOnClickListener(new View.OnClickListener() {

  @Override
  public void onClick(View view) {
    try {
      play();
    } catch (IOException e) {
    }
    textView.setText("Playing...");
  }
});
```

12. Finally, add the `play()` method, which looks like this:

    ```
    public void play() throws IllegalArgumentException,
      SecurityException, IllegalStateException, IOException {
      MediaPlayer player = new MediaPlayer();
      player.setDataSource(filename);
      player.prepare();
      player.start();
    }
    ```

13. You can now run the app on a handset (as the stock emulators do not yet have the microphone functionality) and record and play back audio. The file `recording.3gp` can be found in the root directory of the SD card.

The `MediaRecorder` class makes light work of recording audio and can also be used to record video. Like the `MediaPlayer` class we used to play the audio back, this is similarly intuitive to use. The use of `MediaRecorder.release()` is important, as without it the system would continue to use resources. We only prepared and played the file here, but the `MediaPlayer` can do a lot more and it is well worth taking a look at its documentation, which can be found at `http://developer.android.com/reference/android/media/MediaPlayer.html`.

Again, we used `Environment.getExternalStorageDirectory()` to automatically select the user's preferred external storage device and although we took a different approach to the way we managed multimedia earlier in the chapter, either method can be applied in many situations. The `MediaRecorder` and `MediaPlayer` together provide a simple but powerful way to incorporate audio in our apps.

Summary

Multimedia such as audio and video have become an integral part of the way we use our mobile devices. Whether it is to create it or consume it, including multimedia functionality in our apps gives them greater appeal and usefulness. In this chapter, we saw how to incorporate native apps such as the camera into our own apps, saving us a great deal of coding in the process. We saw how to capture, record, and play back camera images, video, and finally audio.

This concludes not just our exploration of Android 5 multimedia, but also, more or less, the programming aspect of this book, as the final chapter looks at how we can take our finished product and make it available to the world, and how to hopefully turn our hard work into a financial gain. There are one or two exercises in the chapter, as we will look at how to make our apps backwards compatible to reach a larger number of potential users, and we will return one last time to the Ancient Britain app to add a mobile advertisement to it using the Google AdMob service.

10
Publishing and Marketing

Having covered all the topics in this book, and if you have not already done so, you are now in a position to create, develop, and market an app of your own design. There is of course much more to learn about Android 5, and Android in general, but exploring the SDK further is quite simple, now we understand how some of the most frequently used structures and objects are applied. Once we understand how a listener interface is implemented, then it is simply a matter of looking up in the documentation when we want to imply a new one.

The entire purpose of developing an app for Android is to distribute it. Although there are many ways to make our work available to others, the obvious choice is via the Android Play Store. This final chapter is a step by step guide on how to do that. On the way, we will see how to make our apps compatible with earlier versions, keeping much of the Android 5 API's functionality as well as many of the Material Design features programmed into our Lollipop UIs.

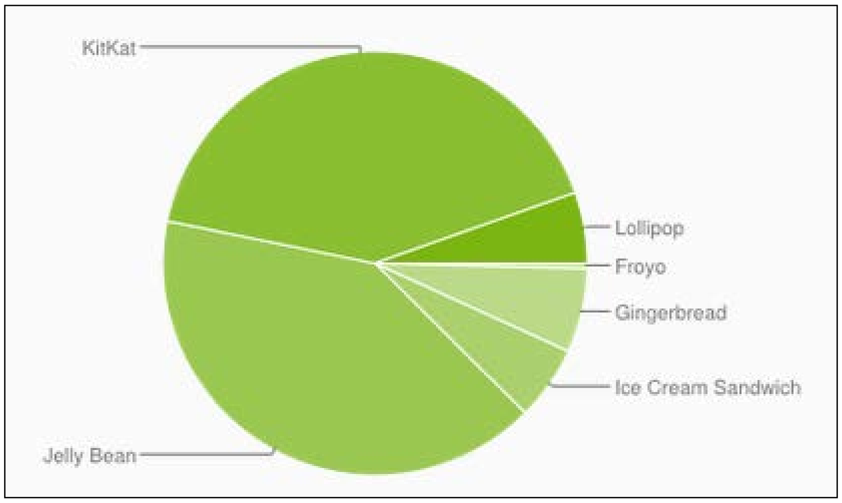
In this chapter, you will:

- Making apps backward-compatible
- See how to create alternative layouts for older systems
- Apply a Material Theme to older versions
- Replace the Action Bar with a Material Design Toolbar
- Prepare an app for publication
- Create a digital certificate and a private key
- Generate a signed APK file
- Prepare promotional media
- Complete a Store Listing
- Publish an app
- Learn how to distribute apps via e-mail and websites

- License an app
- Provide links to a product or a publisher
- Add official branding
- Build a template project for in-app billing
- Include an AdMob banner advertisement

Making apps backward-compatible

Throughout this book, we have focused entirely on developing for Android 5, and although the number of devices running this platform is bound to increase dramatically, they still only make up a small proportion of all active Android devices. In fact, **Jelly Bean** and **KitKat** (APIs 16 through 19) still make up the vast majority of platform versions accessing the Google Play Store.



An up-to-date report of the relative distribution of platforms over all active devices can be found at `http://developer.android.com/about/dashboards/index.html`. This page also contains similar information about currently used screen sizes and densities, and can greatly facilitate how we target users.

Obviously, we want our apps to reach as many people as possible and many of the apps we develop take very little adjustment to make them available to users running earlier versions. Fortunately, Android provides support libraries, such as **v7 AppCompat r21** (or higher) to facilitate this.

Adding the v7 support libraries

Realistically, we should consider carefully which platforms we want our app to be available for, long before we start developing; for demonstration purposes, however, in this next short exercise we will make an app that we developed earlier in the book available to devices running API 16 and greater.

1. Open the **Ancient Britain** app that we developed earlier.
2. Open the `manifest` file.
3. Inside the root node, include this tag:

```
<uses-sdk android:minSdkVersion="16"
    android:targetSdkVersion="22" />
```

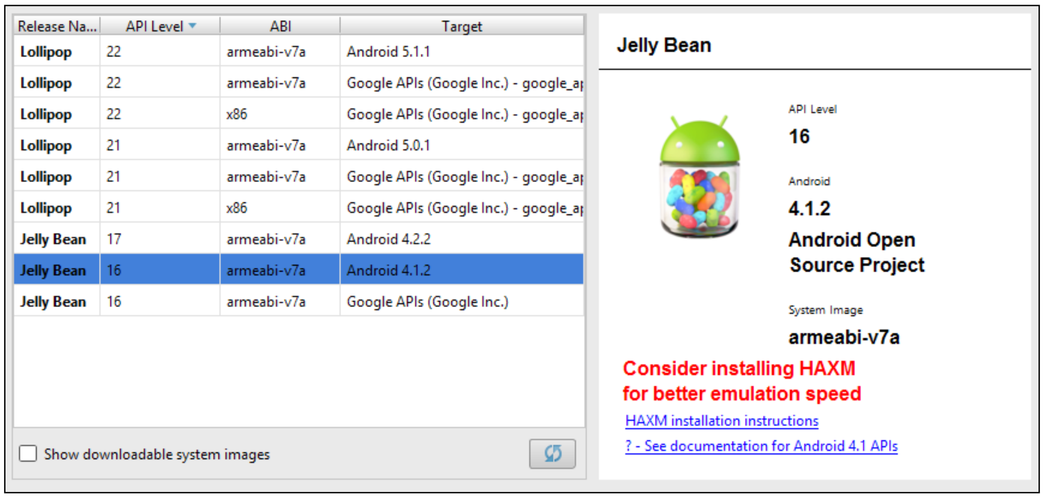
4. Open the `build.gradle` file and add these dependencies:

```
compile 'com.android.support:cardview-v7:22.0.+'
compile 'com.android.support:recyclerview-v7:22.0.+'
compile "com.android.support:appcompat-v7:22.0.+"
```

5. Edit the default configuration, like so:

```
defaultConfig {
    applicationId "com.example.kyle.ancientbritain"
    minSdkVersion 16
    targetSdkVersion 22
    versionCode 1
    versionName "1.0"
}
```

6. Prepare an AVD or handset targeting API level 16:



Release Na...	API Level	ABI	Target
Lollipop	22	armeabi-v7a	Android 5.1.1
Lollipop	22	armeabi-v7a	Google APIs (Google Inc.) - google_a
Lollipop	22	x86	Google APIs (Google Inc.) - google_a
Lollipop	21	armeabi-v7a	Android 5.0.1
Lollipop	21	armeabi-v7a	Google APIs (Google Inc.) - google_a
Lollipop	21	x86	Google APIs (Google Inc.) - google_a
Jelly Bean	17	armeabi-v7a	Android 4.2.2
Jelly Bean	16	armeabi-v7a	Android 4.1.2
Jelly Bean	16	armeabi-v7a	Google APIs (Google Inc.)

Show downloadable system images

Jelly Bean

API Level
16

Android
4.1.2
Android Open Source Project

System Image
armeabi-v7a

Consider installing HAXM for better emulation speed

HAXM installation instructions

? - See documentation for Android 4.1 APIs

7. Run the app on the device. It will appear to work normally until you try to swipe the images on the DetailActivity screen, when it will crash.
8. Open the `DetailActivity.java` file.
9. In the `onShowPress()` and the `onFling()` methods, there is a call to `detailImage.setElevation()`. Apply conditional clauses to each, like so:

   ```
   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
       detailImage.setElevation(4);
   }

   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
       detailImage.setElevation(0);
   }
   ```

10. Run the app again to check that this fix has worked.

Declaring `uses-sdk` in the manifest is essential as this is how the Play Store decides which devices it is visible to. The **v7 AppCompat r21+** library is what makes backward-compatibility possible. Among other things, it provides very passable, Material Design widgets and other UI components. There are also libraries for the **RecyclerView** and **CardView** and, although the shadows are not dynamic, this is a small price to pay, considering the vast number of users our app can now reach.

Changing the minimum SDK level is only the first thing we need to do to make our apps available to older versions As we saw, APIs with a level of 21 or higher will cause the app to crash when called, as with the `setElevation()` call in this task. By being able to query a device's API at runtime, we have a way to work around this limitation and often with very little loss to the quality of user experience.

Another convenient way to counter this problem, is to create separate layouts for different platforms. You can create a `res/layout/v-21/` directory for your Material Design layouts and older alternatives in `res/layout/`.

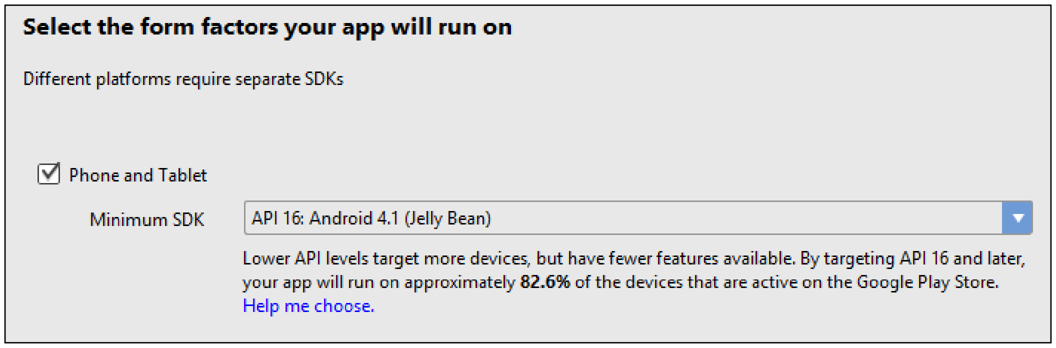
To really bring the **Material Design** feel to older platforms, there is a lot more we can do to with these libraries, such as making our customized themes available, and this is what we will do next.

Applying Material Design to older platforms

Usually when developing an app for pre-API 21 platforms, we set the minimum SDK at the lowest target level when we create the app rather than reverse-engineering the process as we just did. Here, we will see how we can add many **Material Design** features. Developing this way round is also a good way to judge just how far backwards we want our app to be, and how much functionality we are prepared to lose.

This next exercise demonstrates how to build an app for API 16 and apply Material Design to the UI. To do this, follow these steps:

1. Start a new project in the Android Studio called **Material Jelly Bean**.
2. Do not use `com.example` in the package name.
3. Select **Phone and Tablet** as the form factor and **API 16** as the minimum SDK. Note the number of devices that support your app.



Select the form factors your app will run on

Different platforms require separate SDKs

☑ Phone and Tablet

Minimum SDK: API 16: Android 4.1 (Jelly Bean)

Lower API levels target more devices, but have fewer features available. By targeting API 16 and later, your app will run on approximately **82.6%** of the devices that are active on the Google Play Store. Help me choose.

4. Select **Blank Activity** and leave everything else as is, or choose your own values.
5. Add the following to the root node of the manifest file:

```
<uses-sdk android:minSdkVersion="16"
        android:targetSdkVersion="22" />
```

6. Open the `res/values/styles.xml` file and fill it out like this:

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light">
        <item name="colorPrimary">#ff7d00</item>
        <item name="colorPrimaryDark">#d96a00</item>
        <item name="colorAccent">#b25900</item>
    </style>
</resources>
```

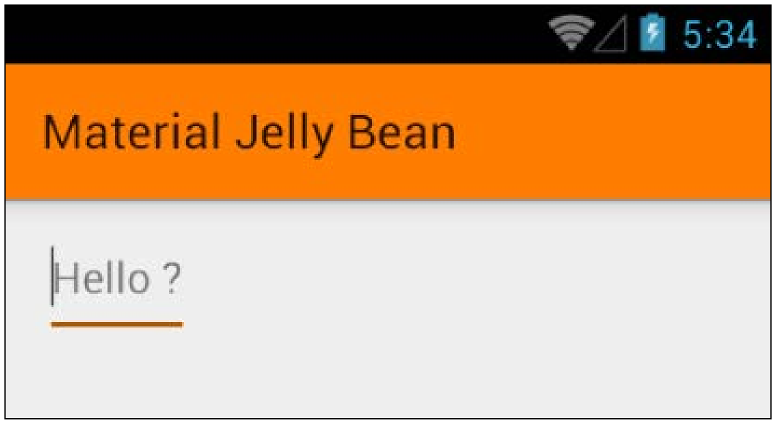
7. Open the `activity_main.xml` file.
8. Change the root layout from a relative layout to a linear one and set its orientation by adding this:

```
android:orientation="vertical"
```

9. Convert `TextView` to this `EditText`:

```
<EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="Hello ?"
    android:inputType="text" />
```

10. Run the app at this point and our Material Theme palette will have been applied:



11. Disable the Action Bar by setting the theme in the `styles.xml` file to:

```
Theme.AppCompat.Light.NoActionBar
```

12. Above `EditText`, place this toolbar:

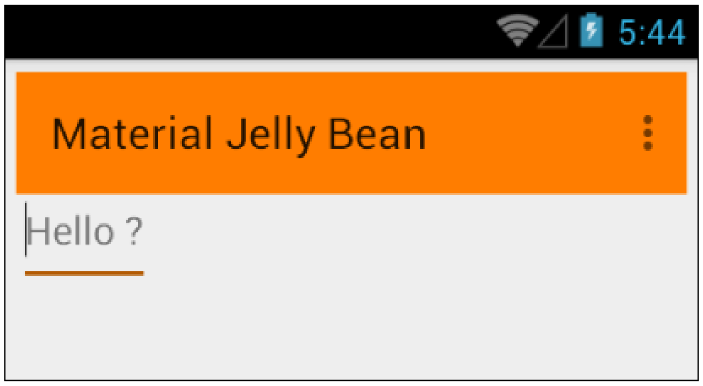
```
<android.support.v7.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"/>
```

13. You may wish to adjust the padding of the layout in the `dimens.xml` file.

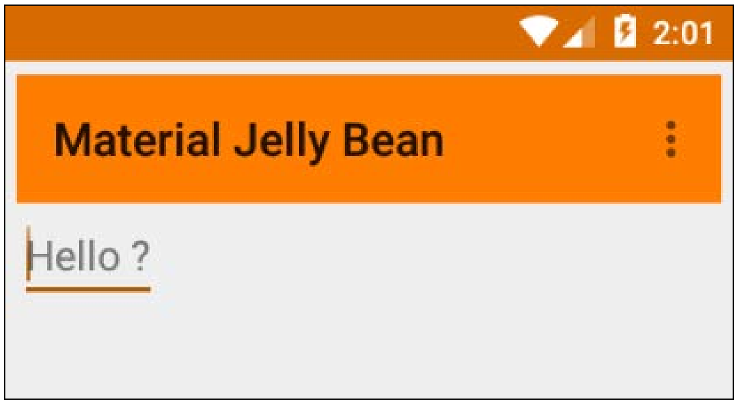
14. Open your main activity file and add the following code to the `onCreate()` method:

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
if (toolbar != null) {
    setSupportActionBar(toolbar);
}
```

15. This will generate an import error. Use the quick fix to select the v7 Toolbar.
16. Run the app on a API 16 device or emulator to see the Material Design style Toolbar.



The **AppCompat** family of themes provides nearly all of the Material Design features we have become used to. The colors we have chosen to represent our app still appear throughout the app in familiar places and tinting various widgets, giving our app a consistent and recognizable feel. However, certain elements are still lost, and if you run the app on an Android 5 device, you will get the following output.



There are two important things to notice about the way we implemented a **Material Toolbar**. First, notice that our `MainActivity` class extends **ActionBarActivity** and this has to be the case for any app we build using AppCompat, if it is to have a Toolbar. Second, note that the way we inflated it, unlike most views, with `setSupportActionBar()`. These are the only two real differences between the ways we are accustomed to managing Toolbars; other than this, everything can be done with classes and methods that we are familiar with.

There are one or two other things that can be done to bring Material Design to earlier versions but, for now, this is enough to set us on our way to bringing our apps to as many people as possible. Next, we move on to the serious subject of publishing our apps to the world.

Publishing apps

It goes without saying that you will have exhaustively tested your app on a wide variety of handsets and emulators, probably prepared your promotional material, and checked out **Google Play Policies and Agreements**. There are many things to consider before publication, such as **content rating** and **country distribution**. From a programming point of view there are just three things that we need to check before we proceed.

- Remove all logging from the project, such as:

  ```
  private static final String DEBUG_TAG = "tag";
  Log.d(DEBUG_TAG, "some info");
  ```

- Make sure you have an application `label` and `icon` declared in your manifest, for example:

  ```
  android:icon="@mipmap/my_app_icon"
  android:label="@string/my_app_name"
  ```

- Ensure you have declared all the necessary permissions in the manifest, for example:

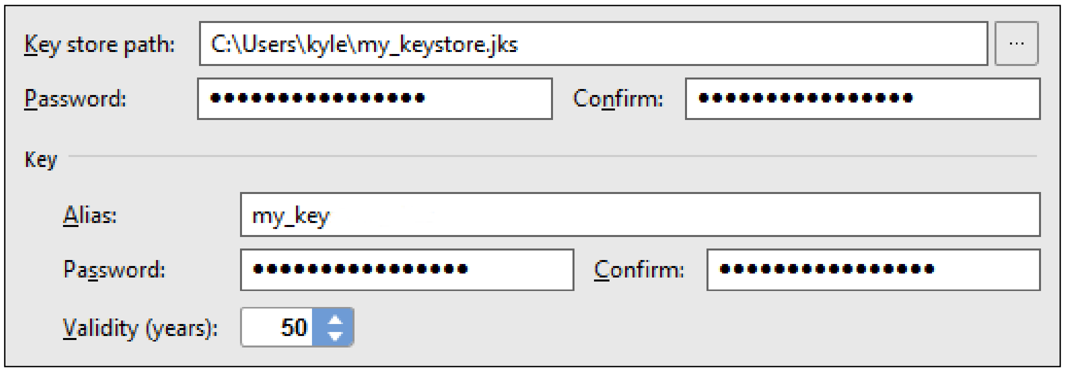
  ```
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.ACCESS_NETWORK_
  STATE" />
  ```

We are now just three steps from seeing our app on the Google Play Store. All we need to do is generate a **Signed Release APK**, register as a **Google Play Developer**, and finally upload our app to the Store or publish it on our own site. There are also one or two other ways of publishing an app and we will see how they are done at the end of the section. First, though, we will begin by generating an APK that is ready for uploading onto the Google Play Store.

Generating a signed APK

All published Android apps require a digitally signed certificate. This is used to prove the authenticity of an app. Unlike many other digital certificates, there is no authority and you hold the signed key, which clearly has to be securely protected. To do this, we need to generate a private key and then use it to generate a signed APK. This can all be done in the Android Studio with the Generate Signed APK Wizard. These steps will take you through it.

1. Open the app you want to publish.
2. Start the Generate Signed APK Wizard from the **Build | Generate Signed APK...** menu.
3. Select **Create new...** on the first screen.
4. On the next screen, provide a path and name for your key store along with a strong password.
5. Do the same for the Alias.
6. Select a Validity of greater than 27 years, like so:



Key store path:	C:\Users\kyle\my_keystore.jks		
Password:	••••••••••••••••	Confirm:	••••••••••••••••
Key			
Alias:	my_key		
Password:	••••••••••••••••	Confirm:	••••••••••••••••
Validity (years):	50		

7. Fill in at least one of the Certificate fields. Click on **OK** and you will be taken back to the wizard.
8. Select **release** as the Build Variant and click on **Finish**.
9. You now have a signed APK ready for publication.

The key store (`.jks` file) can be used to store any number of keys (aliases). Normally you would have a different key for each app you publish, and you must use the same key when producing updates of an app. Google require certificates to be valid until at least 22nd October, 2033 and any number that surpasses this date will suffice.

IMPORTANT: Keep at least one secure backup of your keys. If you lose them, you will not be able to develop future versions of those apps.

Most Android apps are packaged this way with one exception: Google Wear. If you are publishing a Wear app, you will first need to visit `http://developer.android.com/training/wearables/apps/packaging.html`. With our digital certificate signed and in place, we are now only two steps from being published. If you have not already done so, it is time to register as a Google Play Developer.

Registering as a developer

As with Signing an APK, registering as a developer is similarly straightforward. Note that Google charges a one-off fee of USD 25 and 30% of any revenue your app may generate. The following directions assume that you already have a Google account.

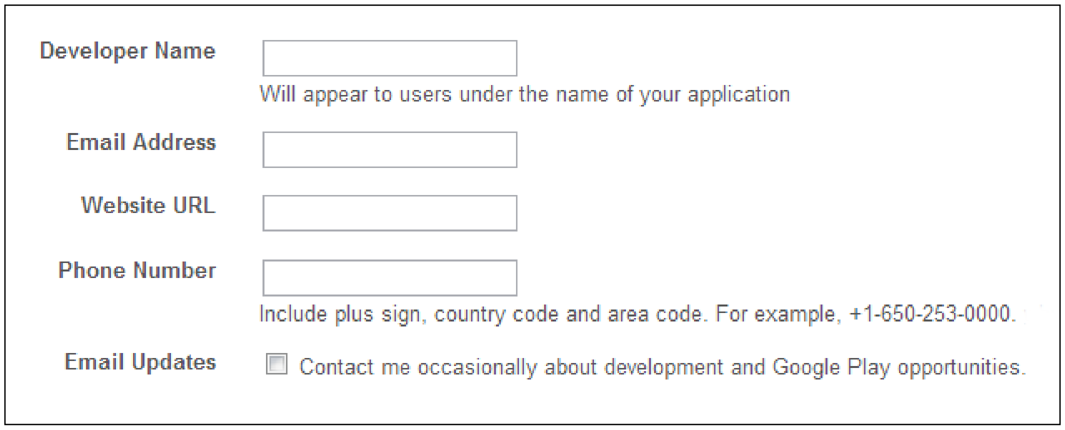
1. Review **Supported Locations** at:

   `https://support.google.com/googleplay/android-developer/table/3541286?hl=en&rd=1`

2. Go to the Developer Play Console at:

   `https://play.google.com/apps/publish/`

3. Sign in with your Google account and enter the following information:



Developer Name	Will appear to users under the name of your application
Email Address	
Website URL	
Phone Number	Include plus sign, country code and area code. For example, +1-650-253-0000.
Email Updates	☐ Contact me occasionally about development and Google Play opportunities.

4. Read and accept the **Google Play Developer Distribution Agreement**.
5. Pay the USD 25 with Google Checkout, creating an account if necessary, and that's it; you are now a registered Google Developer.

If you intend to make your apps available worldwide, then it is always worth checking the Supported Locations page, as it changes regularly. The only thing left to do is upload our app, which we will do now.

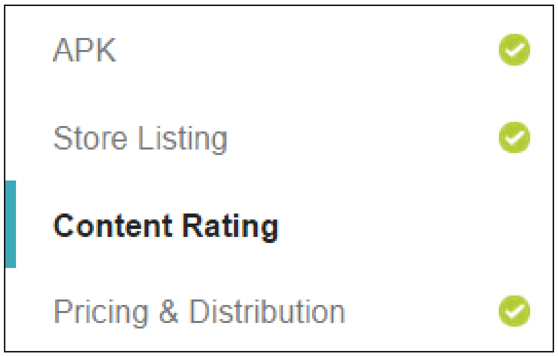
Publishing an app on the Google Play Store

Uploading and publishing our apps to the Play Store is done through the **Developer Console**. As you will see, there is a lot of information and promotional material that we could provide about our app during this process. Provided, you have followed the previous steps in this chapter and have a release-ready signed `.apk` file, complete the following instructions to publish it. Alternatively, you may just want to have a look at what is involved at this point and what form the promotional material will take. In this case, ensure you have the following four images and a signed APK, and select **Save Draft** at the end rather than **Publish app**.

- At least two screenshots of your app. These must not have any side that is shorter than 320 px or longer than 3840 px.
- If you want your app to be visible on the Play Store to users searching for apps designed for tablets, then you should prepare at least one 7-inch and one 10 inch screenshot.
- A **Hi-res icon** image of 512 x 512 px.
- A **Feature Graphic** of 1024 x 500 px, for example; with these images prepared and a signed `.apk` file, we have all we need to start. Decide how much, if anything, you wish to charge for the app and then follow these instructions:

    1. Open your **Developer Console**.
    2. Supply a **Title** and click on the **Upload APK** button.
    3. Click on **Upload your first APK to Production**.
    4. Locate your signed `app-release.apk` file. It will be in `AndroidStudioProjects\YourApp\app`.
    5. Drag-and-drop this into the space suggested.
    6. When this is completed, you will be taken to the application page.

7. Work your way through the top four sections:



8. Complete all required fields until the **Publish app** button becomes clickable.
9. If you need help, the **Why can't I publish?** link above the button will list uncompleted compulsory fields.
10. When all the required fields are completed, click on the **Publish app** (or **Save draft**) button at the top of the page.
11. Congratulations! You are now a published Android Developer.

We now know how to publish our apps on the Play Store. There are, of course, many other app markets, and they all have their own uploading procedures. Google Play however provides the widest possible audience and is the obvious choice for publication.

Before we move on, there are two other methods of distribution that we will look at: publishing on a website and distributing via e-mail.

Distributing by e-mail and websites

The first of these two methods is as simple to do as it sounds. If you attach the APK to an e-mail and it's opened on an Android device, the user will be offered the opportunity to install the app when the attachment is opened. On more recent devices, they will be able to tap an install button directly from the e-mail.

For both these methods, your users will have to allow the installation of **unknown sources** in the security settings of their devices.

Distributing your app from your website is almost as simple as e-mailing it. All you need to do is host the APK file on your site somewhere and provide a download link along the lines of `<a href="download_button.jpg" download="your_apk">`. When browsing your site from an Android device, a tap on your link will install your app on their device.

[
Distribution by e-mail provides no protection against piracy and should only be used with this in mind. The other methods are as secure as we could hope, but if you would like to take extra measures then Google offers a **Licensing Service** that can be found at `developer.android.com/google/play/licensing`.
]

Whether we have released a paid app or a free one, we want to be able to reach as many users as possible. Google provides several tools to help us with this, as well as ways to monetize our apps, as we shall see next.

Promoting and monetizing apps

Very few apps become successful without first being well promoted. There are countless ways to do this and you will, no doubt, be well ahead of the curve on how to promote your products. To help you reach a wider audience, Google provides some handy tools to assist with promotion.

After looking at promotion tools, we will explore two ways to make money from our app: in-app payments and advertising.

Promoting an app

There are two very simple methods, provided by Google, to help steer people towards our products on the Play Store; links from both websites and our apps and the **Google Play Badge,** which provides official branding to our links.

We can add links to both individual apps and our publisher page, where all our apps can be browsed. We can include these links in our apps as well as our websites.

- To include a link to a specific app's page in the Play Store, use the full package name, as found in the Manifest, in the following format:

  `http://play.google.com/store/apps/details?id=`**`com.full.package.name`**

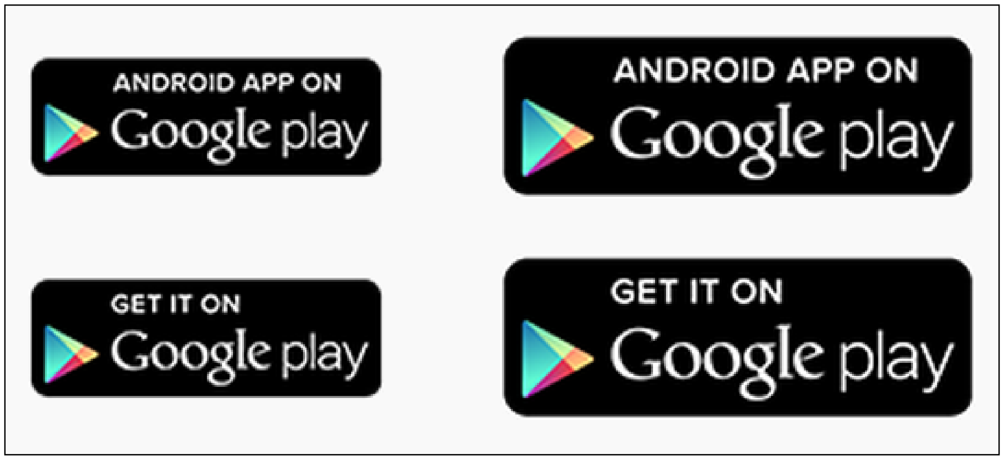
- To Include this within an Android app, use:

  `market://details?id=` **`com.my.full.package.name`**

- If you want a link to your publisher page and a list of all your products, use:
  `http://play.google.com/store/search?q=pub:`**`my publisher name`**
- Make the same changes as before when linking from an app:
  `Market://search?q=pub:`**`my publisher name`**
- To link to a specific search result, use:
  `search?q=`**`my search query`**`&c=apps.`
- To use an official Google Badge as your link, replace one of the above elements with the highlighted HTML here:

```
<a href="https://play.google.com/store/
  search?q=pub:my publisher name">
  <img alt="Get it on Google Play"
       src="https://developer.android.com/
         images/brand/en_generic_rgb_wo_60.png" />
</a>
```

The **Badge** comes in two sizes, `60.png` and `45.png`, and two styles, `"Android app on Google Play"` and `"Get it on Google Play"`. Simply change the relevant code to select the Badge that best suits your purpose.



With our app published and with well-placed links to our Play Store page, it is now time to consider how we can profit from the inevitable downloads, and so we come to how to monetize our Android app.

Monetizing an app

There are many ways to make money from an app, but two of the most popular and effective are: **in-app billing** (**IAB**) and **advertising**. In-app billing can become quite involved and perhaps deserves an entire chapter to itself. Here, we will see how to build an effective template that you can use as a foundation for an in-app product you might develop. It will include all the libraries and packages needed, along with some very useful helper classes.

Including **Google AdMob** advertisements in our apps is, in contrast, a very familiar process to us by now. An ad is in effect just another View, and can be identified and referenced just like any other Android widget. The final exercise of this chapter, and indeed this book, will be constructing a simple working AdMob demo. First, though, let us take a look at in-app billing.

In-app billing

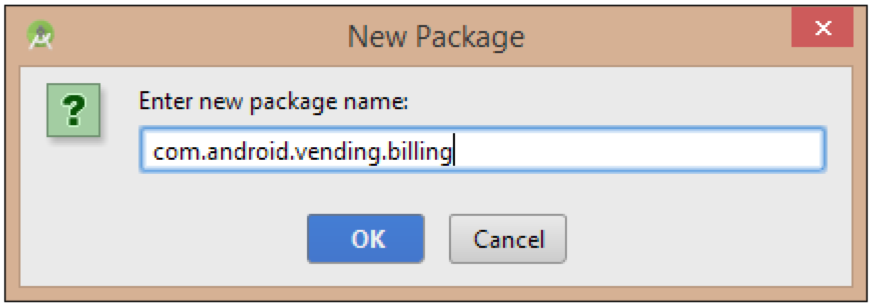
There are a large number of products that users can purchase from within an app, from upgrades and unlockables to in-game objects and currencies. Whatever the user is buying, the Google checkout process ensures they will pay in the same way as they pay for other Play Store products. From the developer's point of view, each purchase will boil down to responding to the click of a button. We will need to install the **Google Play Billing Library**, and add an **AIDL** file and some helper classes to our project. Here is how:

1. Start a new Android project or open one you want to add in-app billing to.
2. Open the SDK Manager.
3. Under **Extras**, make sure you have the **Google Play Billing Library** installed.
4. Open the manifest and apply the following permission:

   ```
   <uses-permission android:name="com.android
     .vending.BILLING" />
   ```

5. In the Project pane of the Studio, right click on **app** and select **New | Folder | AIDL Folder**.

6. From this folder as **aidl**, create a **New** | **Package**, and fill out the resultant dialog like so:



com.android.vending.billing

7. Locate and copy the `IinAppBillingService.aidl` file in the `sdk\extras\google\play_billing` directory.
8. Paste the file into the `com.android.vending.billing` package.
9. Create a **New** | **Package** in the Java folder, selecting `...\app\src\main\java` from the dialog box.
10. Name the package `com.`**`your.package.name`**`.util` and click on **Finish**.
11. From the `play_billing` directory, locate and open the `TrivialDrive\src\com\example\android\trivialdrivesample\util` folder.
12. Copy the nine Java files into the `util` package you just created.

You now have a working template for any app you wish to include on in-app purchasing. Alternatively, you can complete the above steps on a project where you have already developed your in-app products. Either way, you will no doubt be taking advantage of the `IabHelper` class, which vastly simplifies the coding, providing listeners for every step of the purchasing process. Documentation on IAB can be found at `http://developer.android.com/google/play/billing/billing_reference.html`.

Before you can start to implement in-app purchases, you will need to secure a **License Key** for your app. This can be found in the app's details in your developer console.

Paid apps and in-app products are just two ways to make money from an app, and many people choose another, and often lucrative, route for monetizing their work through advertising. **Google AdMob** allows for a great deal of flexibility and a familiar programming interface, as we shall see next.

Including an advertisement

There are many ways that we can earn money from advertising, but **AdMob** provides one of the easiest. Not only does the service allow you to select what types of product you wish to advertise but it also provides great analytical tools and seamless payment into your Checkout account.

On top of this, an **AdView** can be treated programmatically in a way that is almost identical to the methods we are used to and familiar with, as we shall see in this final exercise where we will develop a Hello World app with a demo banner AdMob ad.

Before you start this exercise, you will need to have signed up for an AdMob account at `http://www.google.com/admob/`.

1. Open a project you want to test ads on or start a new Android project.
2. Make sure you have the Google Repository installed with the SDK Manager.
3. In the `build.gradle` file, add this dependency:

```
compile 'com.google.android.gms:play-services:7.0.+'
```

4. Rebuild the project.
5. In the manifest, set these two permissions:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_
STATE" />
```

6. Within the `application` node, add this `meta-data` tag:

```
<meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />
```

7. Include this second Activity to the manifest:

```
<activity
    android:name="com.google.android.gms.ads.AdActivity"
    android:configChanges="keyboard|keyboardHidden|
       orientation|screenLayout|uiMode|screenSize|smallestScreenSi
ze"
    android:theme="@android:style/Theme.Translucent" />
```

8. Add the following string to the `res/values/strings.xml` file:

```
<string name="ad_id">ca-app-pub-3940256099942544/6300978111</
string>
```

9. Open the `main_activity.xml` layout file.

10. Add this second namespace to the root layout:

```
xmlns:ads="http://schemas.android.com/apk/res-auto"
```

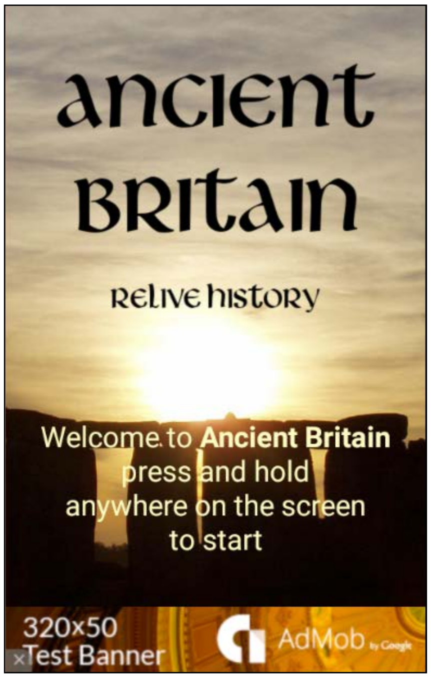
11. Add this `AdView` under the `TextView`:

```
<com.google.android.gms.ads.AdView
    android:id="@+id/ad_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_centerHorizontal="true"
    ads:adSize="BANNER"
    ads:adUnitId="@string/ad_id"></com.google
      .android.gms.ads.AdView>
```

12. In the `onCreate()` method of the MainActivity, insert these lines:

```
AdView adView = (AdView) findViewById(R.id.ad_view);
AdRequest adRequest = new AdRequest.Builder()
        .addTestDevice(AdRequest.DEVICE_ID_EMULATOR)
        .build();

adView.loadAd(adRequest);
```

13. Now test the app on a device.



More or less everything we did here resembles the way that we would program any other element, with one or two exceptions. The use of the `ACCESS_NETWORK_STATE` permission is not strictly necessary; it is used here to check for a connection prior to requesting an ad.

Any Activity that displays an ad will require a separate ID and be declared in the manifest. The ID supplied here is for testing purposes only and it is forbidden to use live IDs for testing purposes. There are only six classes in the `android.gms.ads` package and documentation for all of them can be found at `https://developers.google.com/android/reference/com/google/android/gms/ads/package-summary`.

AdMob ads come in two flavors, the banner that we saw here and the interstitial, or full screen. We only dealt with banner ads here but interstitial ads are handled in a very similar manner. With a knowledge of how to implement paid apps, in-app billing and AdMob, we are now armed to reap the rewards of our hard work and make the very most of our apps.

Summary

In this chapter, we have covered the final aspect of the app development process: packaging and deployment. We began by making our apps backward-compatible, including many of the features we had originally designed for Android 5; by doing, so we were able to reach a far wider audience. We continued by preparing and then publishing our app in the Google Play Store. Once published, we saw how easy it is to promote and then monetize an Android app.

This concludes our journey into the world of Android development. We have gone from installation to publication and hopefully covered most of the components required for the apps that you are planning. If you are new to developing or IDEs such as the Android Studio and have worked your way through this book, then what was previously a daunting set of tools will now seem like a familiar and productive place to work.

The Android platform will, no doubt, continue to flourish and develop in new and unexpected ways. Android 5 is the perfect entry point; with Material Design at its core and the most powerful set of mobile APIs available, things can only get better and brighter for Android developers.

Index