Key Concepts

Now that you have an idea of what Android is, let's take a look at how it works. Some parts of Android may be familiar, such as the Linux kernel, OpenGL, and the SQL database. Others will be completely foreign, such as Android's idea of the application life cycle.

You'll need a good understanding of these key concepts in order to write well-behaved Android applications, so if you read only one chapter in this book, read this one.

2.1 The Big Picture

Let's start by taking a look at the overall system architecture—the key layers and components that make up the Android open source software stack. In Figure 2.1, on the next page, you can see the "20,000-foot" view of Android. Study it closely—there will be a test tomorrow.

Each layer uses the services provided by the layers below it. Starting from the bottom, the following sections highlight the layers provided by Android.

Linux Kernel

Android is built on top of a solid and proven foundation: the Linux kernel. Created by Linus Torvalds in 1991, Linux can be found today in everything from wristwatches to supercomputers. Linux provides the hardware abstraction layer for Android, allowing Android to be ported to a wide variety of platforms in the future.

Internally, Android uses Linux for its memory management, process management, networking, and other operating system services. The Android phone user will never see Linux, and your programs will not

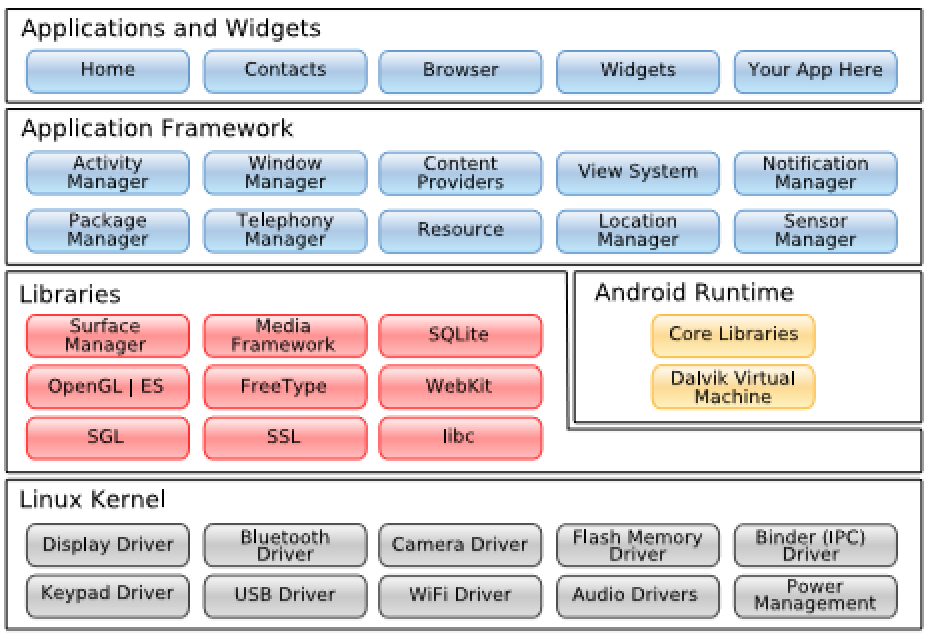


Figure 2.1: Android system architecture

make Linux calls directly. As a developer, though, you'll need to be aware it's there.

Some utilities you need during development interact with Linux. For example, the adb shell command[1] will open a Linux shell in which you can enter other commands to run on the device. From there you can examine the Linux file system, view active processes, and so forth, subject to security restrictions.

Native Libraries

The next layer above the kernel contains the Android native libraries. These shared libraries are all written in C or C++, compiled for the particular hardware architecture used by the phone, and preinstalled by the phone vendor.

Some of the most important native libraries include the following:

- *Surface Manager*: Android uses a compositing window manager similar to Vista or Compiz, but it's much simpler. Instead of drawing directly to the screen buffer, your drawing commands go into off-screen bitmaps that are then combined with other bitmaps to form the display the user sees. This lets the system create all sorts of interesting effects such as see-through windows and fancy transitions.
- *2D and 3D graphics*: Two- and three-dimensional elements can be combined in a single user interface with Android. The library will use 3D hardware if the device has it or a fast software renderer if it doesn't. See Chapter 4, *Exploring 2D Graphics*, on page 73 and Chapter 10, *3D Graphics in OpenGL*, on page 198.
- *Media codecs*: Android can play video and record and play back audio in a variety of formats including AAC, AVC (H.264), H.263, MP3, and MPEG-4. See Chapter 5, *Multimedia*, on page 105 for an example.
- *SQL database*: Android includes the lightweight SQLite database engine,[2] the same database used in Firefox and the Apple iPhone.[3] You can use this for persistent storage in your application. See Chapter 9, *Putting SQL to Work*, on page 178 for an example.
- *Browser engine*: For the fast display of HTML content, Android uses the WebKit library.[4] This is the same engine used in the Google Chrome browser, Apple's Safari browser, the Apple iPhone, and Nokia's S60 platform. See Chapter 7, *The Connected World*, on page 130 for an example.

These libraries are not applications that stand by themselves. They exist only to be called by higher-level programs. Starting in Android 1.5, you can write and deploy your own native libraries using the Native Development Toolkit (NDK). Native development is beyond the scope of this book, but if you're interested, you can read all about it online.[5]

Android Runtime

Also sitting on top of the kernel is the Android runtime, including the Dalvik virtual machine and the core Java libraries.

2. http://www.sqlite.org
3. See http://www.zdnet.com/blog/burnette/iphone-vs-android-development-day-1/682 for a comparison of iPhone and Android development.
4. http://www.webkit.org
5. http://d.android.com/sdk/ndk

Joe Asks...

What's a Dalvik?

Dalvik is a virtual machine (VM) designed and written by Dan Bornstein at Google. Your code gets compiled into machine-independent instructions called *bytecodes*, which are then executed by the Dalvik VM on the mobile device.

Although the bytecode formats are a little different, Dalvik is essentially a Java virtual machine optimized for low memory requirements. It allows multiple VM instances to run at once and takes advantage of the underlying operating system (Linux) for security and process isolation.

Bornstein named Dalvik after a fishing village in Iceland where some of his ancestors lived.

The Dalvik VM is Google's implementation of Java, optimized for mobile devices. All the code you write for Android will be written in Java and run within the VM. Dalvik differs from traditional Java in two important ways:

- The Dalvik VM runs .dex files, which are converted at compile time from standard .class and .jar files. .dex files are more compact and efficient than class files, an important consideration for the limited memory and battery-powered devices that Android targets.
- The core Java libraries that come with Android are different from both the Java Standard Edition (Java SE) libraries and the Java Mobile Edition (Java ME) libraries. There is a substantial amount of overlap, however. In Appendix A, on page 278, you'll find a comparison of Android and standard Java libraries.

Application Framework

Sitting above the native libraries and runtime, you'll find the Application Framework layer. This layer provides the high-level building blocks you will use to create your applications. The framework comes preinstalled with Android, but you can also extend it with your own components as needed.

The most important parts of the framework are as follows:

- *Activity Manager*: This controls the life cycle of applications (see Section 2.2, *It's Alive!*, on page 35) and maintains a common "backstack" for user navigation.

Embrace and Extend

One of the unique and powerful qualities of Android is that all applications have a level playing field. What I mean is that the system applications have to go through the same public API that you use. You can even tell Android to make your application replace the standard applications if you want.

- *Content providers*: These objects encapsulate data that needs to be shared between applications, such as contacts. See Section 2.3, *Content Providers*, on page 40.
- *Resource manager*: Resources are anything that goes with your program that is not code. See Section 2.4, *Using Resources*, on page 40.
- *Location manager*: An Android phone always knows where it is. See Chapter 8, *Locating and Sensing*, on page 161.
- *Notification manager*: Events such as arriving messages, appointments, proximity alerts, alien invasions, and more can be presented in an unobtrusive fashion to the user.

Applications and Widgets

The highest layer in the Android architecture diagram is the Applications and Widgets layer. Think of this as the tip of the Android iceberg. End users will see only these programs, blissfully unaware of all the action going on below the waterline. As an Android developer, however, you know better.

Applications are programs that can take over the whole screen and interact with the user. On the other hand, widgets (which are sometimes called *gadgets*), operate only in a small rectangle of the Home screen application.

The majority of this book will cover application development, because that's what most of you will be writing. Widget development is covered in Chapter 12, *There's No Place Like Home*, on page 233.

When someone buys an Android phone, it will come prepackaged with a number of standard system applications, including the following:

- Phone dialer
- Email
- Contacts
- Web browser
- Android Market

Using the Android Market, the user will be able to download new programs to run on their phone. That's where you come in. By the time you finish this book, you'll be able to write your own killer applications for Android.

Now let's take a closer look at the life cycle of an Android application. It's a little different from what you're used to seeing.

2.2 It's Alive!

On your standard Linux or Windows desktop, you can have many applications running and visible at once in different windows. One of the windows has keyboard focus, but otherwise all the programs are equal. You can easily switch between them, but it's your responsibility as the user to move the windows around so you can see what you're doing and close programs you don't need.

Android doesn't work that way.

In Android, there is one foreground application, which typically takes over the whole display except for the status line. When the user turns on their phone, the first application they see is the Home application (see Figure 2.2, on the next page).

When the user runs an application, Android starts it and brings it to the foreground. From that application, the user might invoke another application, or another screen in the same application, and then another and another. All these programs and screens are recorded on the *application stack* by the system's Activity Manager. At any time, the user can press the Back button to return to the previous screen on the stack. From the user's point of view, it works a lot like the history in a web browser. Pressing Back returns them to the previous page.

Process != Application

Internally, each user interface screen is represented by an Activity class (see Section 2.3, *Activities*, on page 39). Each activity has its own life cycle. An application is one or more activities plus a Linux process to contain them. That sounds pretty straightforward, doesn't it? But don't get comfortable yet; I'm about to throw you a curve ball.

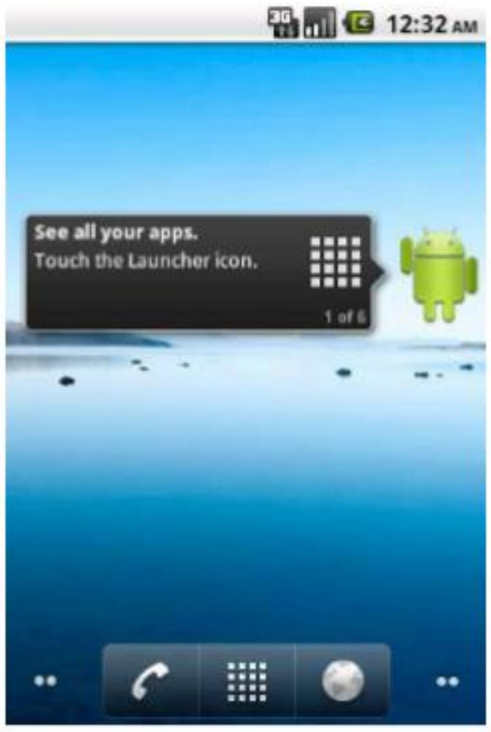


Figure 2.2: The Home application

In Android, an application can be "alive" even if its process has been killed. Put another way, the activity life cycle is not tied to the process life cycle. Processes are just disposable containers for activities. This is probably different from every other system you're familiar with, so let's take a closer look before moving on.

Life Cycles of the Rich and Famous

During its lifetime, each activity of an Android program can be in one of several states, as shown in Figure 2.3, on the next page. You, the developer, do not have control over what state your program is in. That's all managed by the system. However, you do get notified when the state is about to change through the on*XX*() method calls.

You override these methods in your Activity class, and Android will call them at the appropriate time:

- onCreate(Bundle): This is called when the activity first starts up. You can use it to perform one-time initialization such as creating

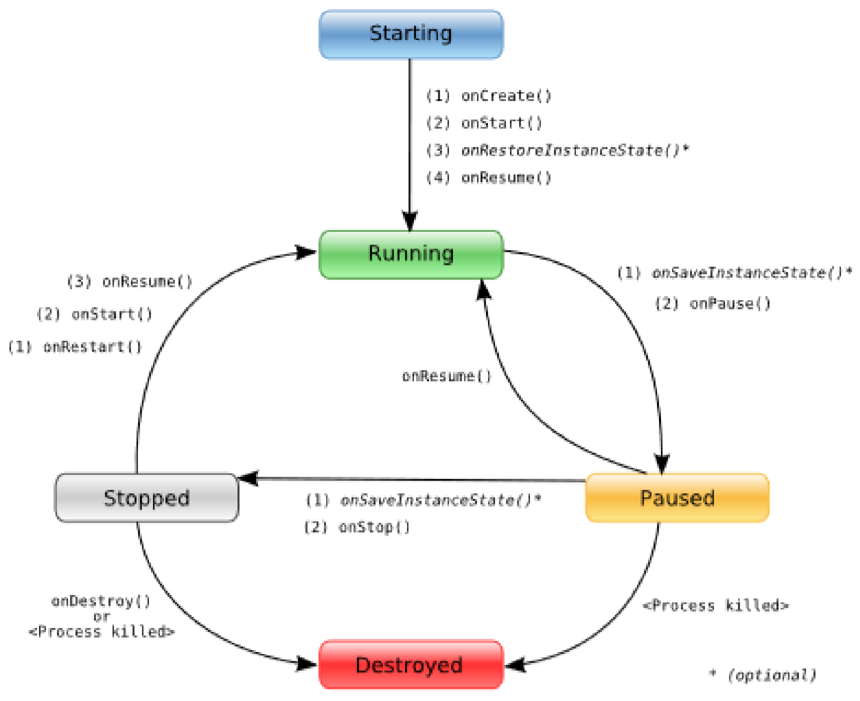


Figure 2.3: Life cycle of an Android activity

the user interface. onCreate() takes one parameter that is either **null** or some state information previously saved by the onSaveInstanceState() method.

- onStart(): This indicates the activity is about to be displayed to the user.
- onResume(): This is called when your activity can start interacting with the user. This is a good place to start animations and music.
- onPause(): This runs when the activity is about to go into the background, usually because another activity has been launched in front of it. This is where you should save your program's persistent state, such as a database record being edited.
- onStop(): This is called when your activity is no longer visible to the user and it won't be needed for a while. If memory is tight, onStop() may never be called (the system may simply terminate your process).

Flipping the Lid

Here's a quick way to test that your state-saving code is working correctly. In current versions of Android, an orientation change (between portrait and landscape modes) will cause the system to go through the process of saving instance state, pausing, stopping, destroying, and then creating a new instance of the activity with the saved state. On the T-Mobile G1 phone, for example, flipping the lid on the keyboard will trigger this, and on the Android emulator, pressing Ctrl+F11 or the 7 or 9 key on the keypad will do it.

- onRestart(): If this method is called, it indicates your activity is being redisplayed to the user from a stopped state.
- onDestroy(): This is called right before your activity is destroyed. If memory is tight, onDestroy() may never be called (the system may simply terminate your process).
- onSaveInstanceState(Bundle): Android will call this method to allow the activity to save per-instance state, such as a cursor position within a text field. Usually you won't need to override it because the default implementation saves the state for all your user interface controls automatically.
- onRestoreInstanceState(Bundle): This is called when the activity is being reinitialized from a state previously saved by the onSaveInstanceState() method. The default implementation restores the state of your user interface.

Activities that are not running in the foreground may be stopped, or the Linux process that houses them may be killed at any time in order to make room for new activities. This will be a common occurrence, so it's important that your application be designed from the beginning with this in mind. In some cases, the onPause() method may be the last method called in your activity, so that's where you should save any data you want to keep around for next time.

In addition to managing your program's life cycle, the Android framework provides a number of building blocks that you use to create your applications. Let's take a look at those next.

2.3 Building Blocks

A few objects are defined in the Android SDK that every developer needs to be familiar with. The most important ones are activities, intents, services, and content providers. You'll see several examples of them in the rest of the book, so I'd like to briefly introduce them now.

Activities

An *activity* is a user interface screen. Applications can define one or more activities to handle different phases of the program. As discussed in Section 2.2, *It's Alive!*, on page 35, each activity is responsible for saving its own state so that it can be restored later as part of the application life cycle. See Section 3.3, *Creating the Opening Screen*, on page 45 for an example.

Intents

An *intent* is a mechanism for describing a specific action, such as "pick a photo," "phone home," or "open the pod bay doors." In Android, just about everything goes through intents, so you have plenty of opportunities to replace or reuse components. See Section 3.5, *Implementing an About Box*, on page 57 for an example of an intent.

For example, there is an intent for "send an email." If your application needs to send mail, you can invoke that intent. Or if you're writing a new email application, you can register an activity to handle that intent and replace the standard mail program. The next time somebody tries to send an email, they'll get the option to use your program instead of the standard one.

Services

A *service* is a task that runs in the background without the user's direct interaction, similar to a Unix daemon. For example, consider a music player. The music may be started by an activity, but you want it to keep playing even when the user has moved on to a different program. So, the code that does the actual playing should be in a service. Later, another activity may bind to that service and tell it to switch tracks or stop playing. Android comes with many services built in, along with convenient APIs to access them. Section 12.2, *Live Wallpaper*, on page 242 uses a service to draw an animated picture behind the Home screen.

Content Providers

A *content provider* is a set of data wrapped up in a custom API to read and write it. This is the best way to share global data *between applications*. For example, Google provides a content provider for contacts. All the information there—names, addresses, phone numbers, and so forth—can be shared by any application that wants to use it. See Section 9.5, *Using a ContentProvider*, on page 192 for an example.

2.4 Using Resources

A *resource* is a localized text string, bitmap, or other small piece of noncode information that your program needs. At build time all your resources get compiled into your application. This is useful for internationalization and for supporting multiple device types (see Section 3.4, *Using Alternate Resources*, on page 55).

You will create and store your resources in the res directory inside your project. The Android resource compiler (aapt)[6] processes resources according to which subfolder they are in and the format of the file. For example, PNG and JPG format bitmaps should go in a directory starting with res/drawable, and XML files that describe screen layouts should go in a directory starting with res/layout. You can add suffixes for particular languages, screen orientations, pixel densities, and more (see Section 13.5, *All Screens Great and Small*, on page 267).

The resource compiler compresses and packs your resources and then generates a class named R that contains identifiers you use to reference those resources in your program. This is a little different from standard Java resources, which are referenced by key strings. Doing it this way allows Android to make sure all your references are valid and saves space by not having to store all those resource keys. Eclipse uses a similar method to store and reference the resources in Eclipse plug-ins.

We'll see an example of the code to access a resource in Chapter 3, *Designing the User Interface*, on page 43.

2.5 Safe and Secure

As mentioned earlier, every application runs in its own Linux process. The hardware forbids one process from accessing another process's

6. http://d.android.com/guide/developing/tools/aapt.html

memory. Furthermore, every application is assigned a specific user ID. Any files it creates cannot be read or written by other applications.

In addition, access to certain critical operations are restricted, and you must specifically ask for permission to use them in a file named AndroidManifest.xml. When the application is installed, the Package Manager either grants or doesn't grant the permissions based on certificates and, if necessary, user prompts. Here are some of the most common permissions you will need:

- INTERNET: Access the Internet.
- READ_CONTACTS: Read (but don't write) the user's contacts data.
- WRITE_CONTACTS: Write (but don't read) the user's contacts data.
- RECEIVE_SMS: Monitor incoming SMS (text) messages.
- ACCESS_COARSE_LOCATION: Use a coarse location provider such as cell towers or wifi.
- ACCESS_FINE_LOCATION: Use a more accurate location provider such as GPS.

For example, to monitor incoming SMS messages, you would specify this in the manifest file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.google.android.app.myapp" >
   <uses-permission android:name="android.permission.RECEIVE_SMS" />
</manifest>
```

Android can even restrict access to entire parts of the system. Using XML tags in AndroidManifest.xml, you can restrict who can start an activity, start or bind to a service, broadcast intents to a receiver, or access the data in a content provider. This kind of control is beyond the scope of this book, but if you want to learn more, read the online help for the Android security model.[7]

Chapter 3

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have covered the steps necessary to configure an environment suitable for the development of Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all of the required development packages are installed and functioning correctly. The best way to achieve this goal is to create an Android application and compile and run it. This chapter will cover the creation of a simple Android application project using Android Studio. Once the project has been created, a later chapter will explore the use of the Android emulator environment to perform a test run of the application.

3.1 Creating a New Android Project

The first step in the application development process is to create a new project within the Android Studio environment. Begin, therefore, by launching Android Studio so that the "Welcome to Android Studio" screen appears as illustrated in Figure 3-1:

3.1 Creating a New Android Project

The first step in the application development process is to create a new project within the Android Studio environment. Begin, therefore, by launching Android Studio so that the "Welcome to Android Studio" screen appears as illustrated in Figure 3-1:

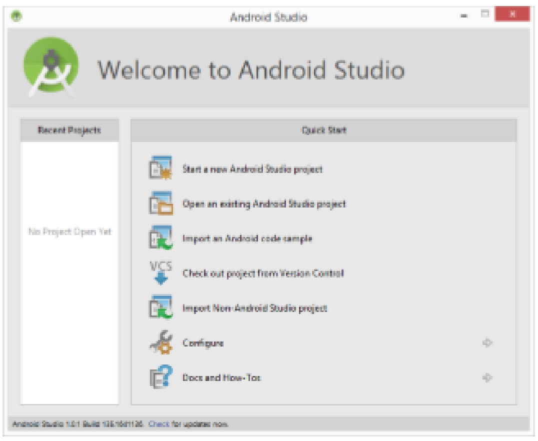
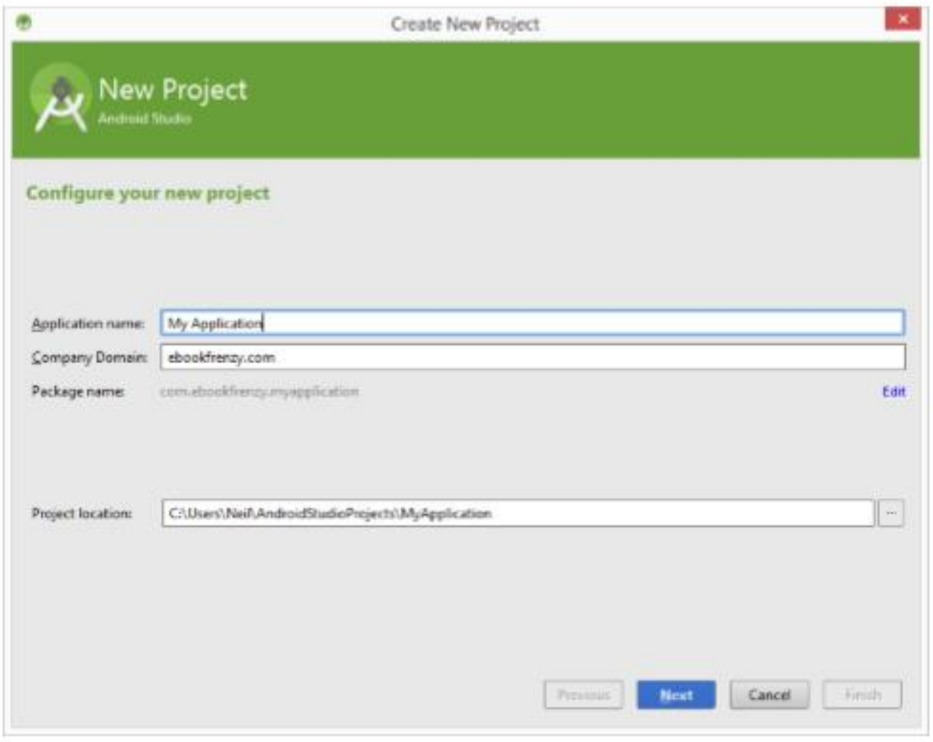

Android Studio
Welcome to Android Studio
Recent Projects
No Project Open Yet
Quick Start
Start a new Android Studio project
Open an existing Android Studio project
Import an Android code sample
Check out project from Version Control
Import Non-Android Studio project
Configure
Docs and How-Tos
Android Studio 1.0.1 Build 135.1641136. Check for updates now.

Figure 3-1

Once this window appears, Android Studio is ready for a new project to be created. To create the new project, simply click on the *Start a new Android Studio project* option to display the first screen of the *New Project* wizard as shown in Figure 3-2:



Application name: My Application
Company Domain: ebookfrenzy.com
Package name: com.ebookfrenzy.myapplication
Project location: C:\Users\Neil\AndroidStudioProjects\MyApplication

Figure 3-2

3.2 Defining the Project and SDK Settings

In the *New Project* window, set the *Application name* field to *AndroidSample*. The application name is the name by which the application will be referenced and identified within Android Studio and is also the name that will be used when the completed application goes on sale in the Google Play store.

The *Package Name* is used to uniquely identify the application within the Android application ecosystem. It should be based on the reversed URL of your domain name followed by the name of the application. For example, if your domain is *www.mycompany.com*, and the application has been named *AndroidSample*, then the package name might be specified as follows:

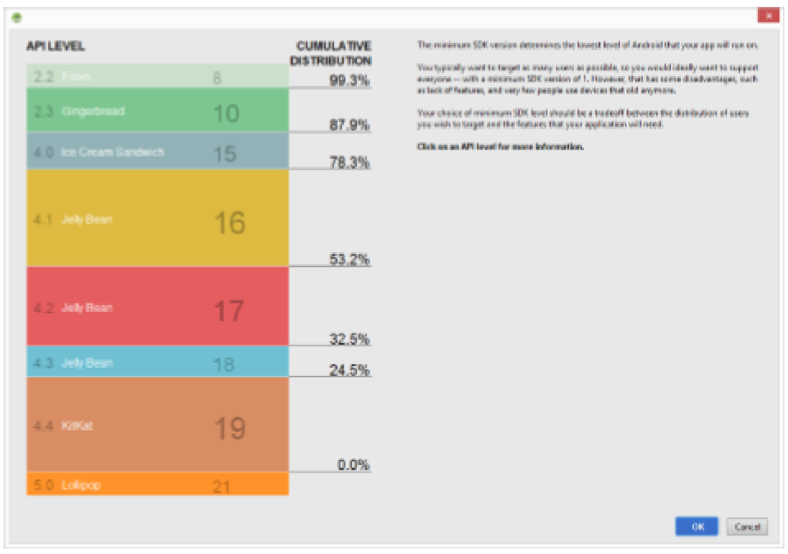
```
com.mycompany.androidsample
```

If you do not have a domain name, you may also use *ebookfrenzy.com* for the purposes of testing, though this will need to be changed before an application can be published:

```
com.ebookfrenzy.androidsample
```

The *Project location* setting will default to a location in the folder named *AndroidStudioProjects* located in your home directory and may be changed by clicking on the button to the right of the text field containing the current path setting.

Click Next to proceed. On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 8: Android 2.2 (Froyo). The reason for selecting an older SDK release is that this ensures that the finished application will be able to run on the widest possible range of Android devices. The higher the minimum SDK selection, the more the application will be restricted to newer Android devices. A useful chart (Figure 3-3) can be viewed by clicking on the *Help me choose* link. This outlines the various SDK versions and API levels available for use and the percentage of Android devices in the marketplace on which the application will run if that SDK is used as the minimum level. In general it should only be necessary to select a more recent SDK when that release contains a specific feature that is required for your application. To help in the decision process, selecting an API level from the chart will display the features that are supported at that level.



The minimum SDK version determines the lowest level of Android that your app will run on.

You typically want to target as many users as possible, so you would ideally want to support everyone -- with a minimum SDK version of 1. However, that has some disadvantages, such as lack of features, and very few people use devices that old anymore.

Your choice of minimum SDK level should be a tradeoff between the distribution of users you wish to target and the features that your application will need.

Click on an API level for more information.

Figure 3-3

Since the project is not intended for Google TV, Google Glass or wearable devices, leave the remaining options disabled before clicking *Next.*

3.3 Creating an Activity

The next step is to define the type of initial activity that is to be created for the application. A range of different activity types is available when developing Android applications. The *Empty*, *Master/Detail Flow*, *Google Maps* and *Navigation Drawer* options will be covered extensively in later chapters. For the purposes of this example, however, simply select the option to create a *Basic Activity*. The Basic Activity option creates a template user interface consisting of an app bar, menu, content area and a single floating action button.

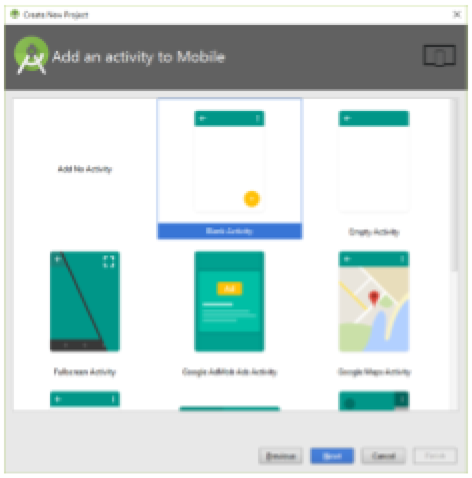
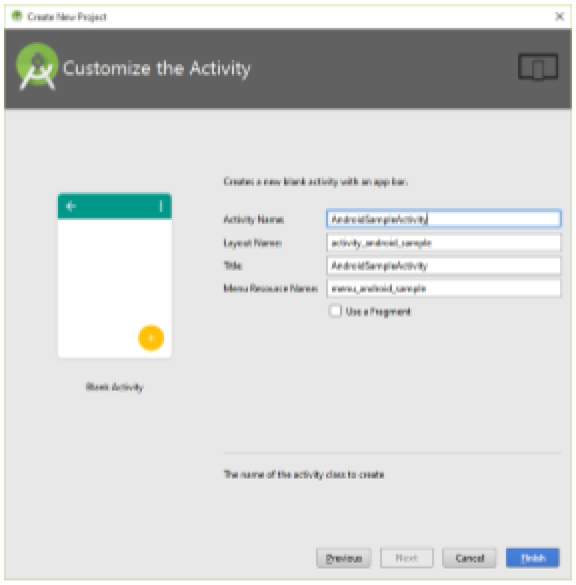


Figure 3-4

With the Basic Activity option selected, click *Next*. On the final screen (Figure 3-5) name the activity and title *AndroidSampleActivity*. The activity will consist of a single user interface screen layout which, for the purposes of this example, should be named *activity_android_sample* as shown in Figure 3-5 and with a menu resource named *menu_android_sample*:



Creates a new blank activity with an app bar.

Activity Name: AndroidSampleActivity
Layout Name: activity_android_sample
Title: AndroidSampleActivity
Menu Resource Name: menu_android_sample
Use a Fragment

The name of the activity class to create

Figure 3-5

3.4 **Modifying the Example Application**

At this point, Android Studio has created a minimal example application project and opened the main window.

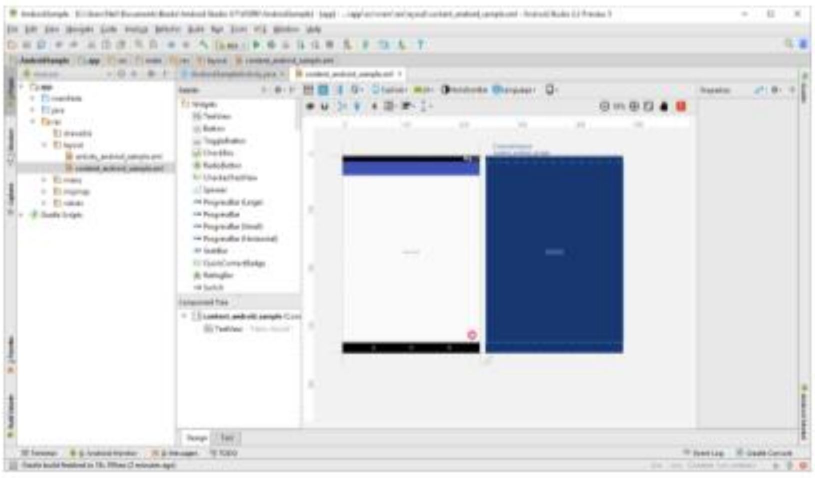


Figure 3-6

The newly created project and references to associated files are listed in the *Project* tool window located on the left hand side of the main project window. The Project tool window has a number of modes in which information can be displayed. By default, this panel will be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-7. If the panel is not currently in Android mode, use the menu to switch mode:

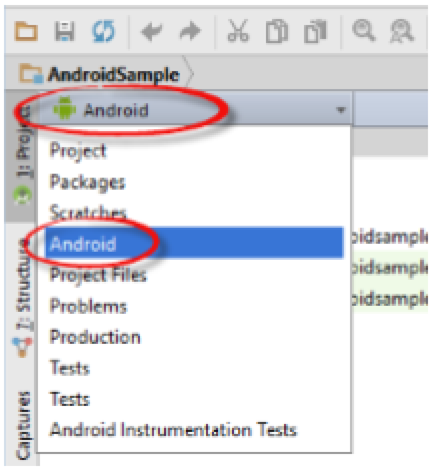


Figure 3-7

The example project created for us when we selected the option to create an activity consists of a user interface containing a label that will read "Hello World!" when the application is executed.

The next step in this tutorial is to modify the user interface of our application so that it displays a larger text view object with a different message to the one provided for us by Android Studio.

The user interface design for our activity is stored in a file named *activity_android_sample.xml* which, in turn, is located under *app -> res -> layout* in the project file hierarchy. This layout file includes the app bar (also known as an action bar) that appears across the top of the device screen (marked A in Figure 3-8) and the floating action button (the email button marked B). In addition to these items, the *activity_android_sample.xml* layout file contains a reference to a second file containing the content layout (marked C):

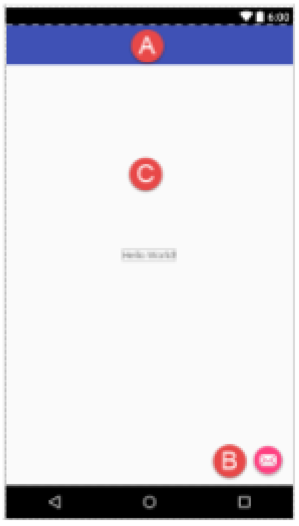


Figure 3-8

By default, the content layout is contained within a file named *content_android_sample.xml* and it is within this file that changes to the layout of the activity are made. Using the Project tool window, locate this file as illustrated in Figure 3-9:

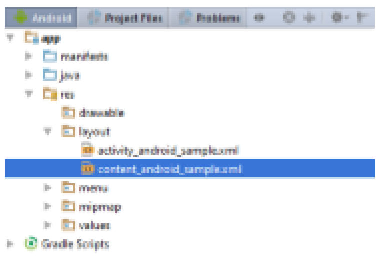


Figure 3-9

Once located, double click on the file to load it into the user interface Layout Editor tool which will appear in the center panel of the Android Studio main window:

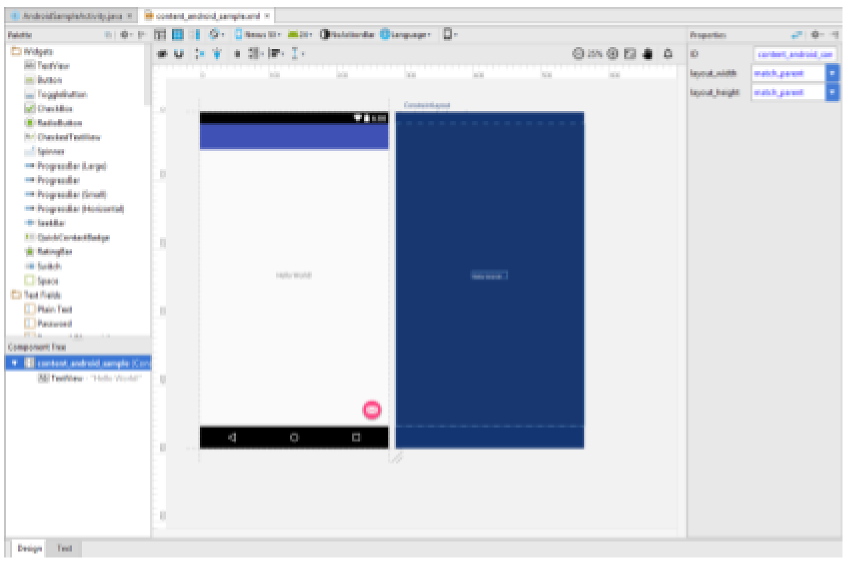


Figure 3-10

In the toolbar across the top of the Layout Editor window is a menu currently set to *Nexus 5X* in the above figure which is reflected in the visual representation of the device within the Layout Editor panel. A wide range of other device options are available for selection by clicking on this menu.

To change the orientation of the device representation between landscape and portrait simply use the drop down menu immediately to the right of the device selection menu showing the icon.

As can be seen in the device screen, the content layout already includes a label that displays a “Hello World!” message. Running down the left hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels and text fields. It should be noted, however, that not all user interface components are obviously visible to the user. One such category consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a ConstraintLayout. This can be confirmed by reviewing the information in the *Component Tree* panel which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-11:

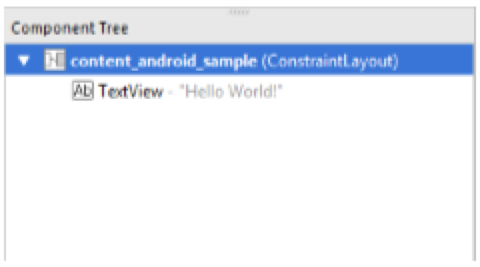

Component Tree
content_android_sample (ConstraintLayout)
TextView - "Hello World!"

Figure 3-11

As we can see from the component tree hierarchy, the user interface layout consists of a ConstraintLayout parent with a single child in the form of a TextView object.

The first step in modifying the application is to delete the TextView component from the design. Begin by clicking on the TextView object within the user interface view so that it appears with a blue border around it. Once selected, press the Delete key on the keyboard to remove the object from the layout.

In the Palette panel, locate the *Widgets* category. Click and drag the *Button* object and drop it in the center of the user interface design when the marker lines appear to indicate the center of the display:

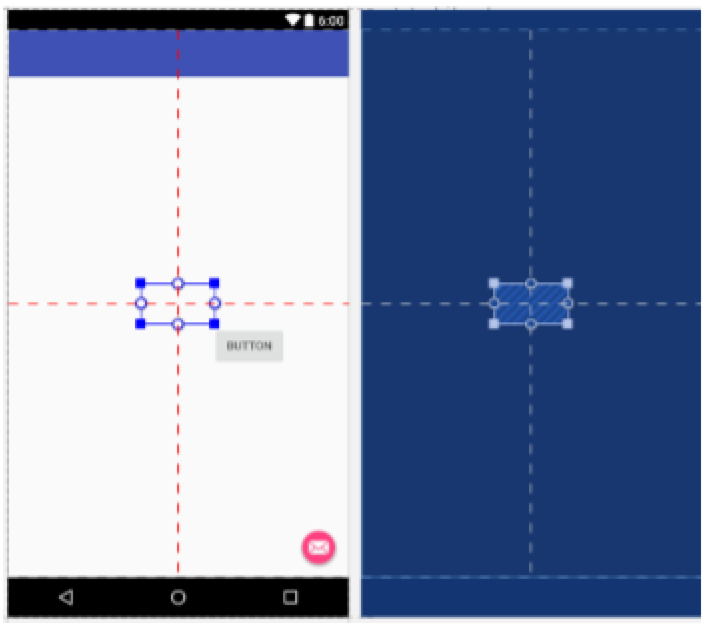
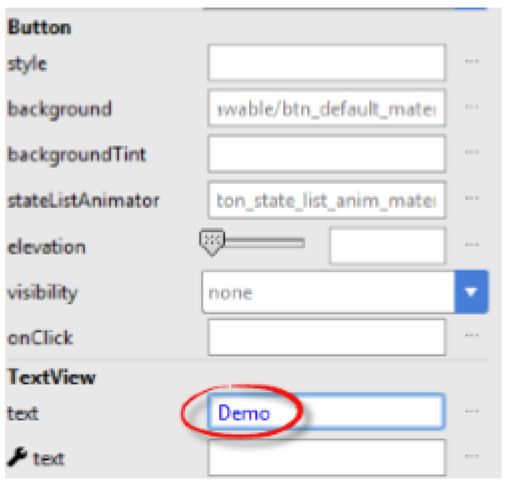


Figure 3-12

The next step is to change the text that is currently displayed by the Button component. The panel located to the right of the design area is the Properties panel. This panel displays the properties

assigned to the currently selected component in the layout. Within this panel, locate the *text* property and change the current value from "Button" to "Demo" as shown in Figure 3-13:



Button	
style	
background	awable/btn_default_mater
backgroundTint	
stateListAnimator	ton_state_list_anim_mater
elevation	
visibility	none
onClick	
TextView	
text	Demo
text	

Figure 3-13

The second text property with a wrench next to it allows a text property to be set which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing the way in which a visual component and the layout will behave with different settings without having to run the app repeatedly.

At this point it is important to explain the red button located in the top right-hand corner of the Layout Editor tool as indicated in Figure 3-14. Obviously, this is indicating potential problems with the layout. For details on any problems, click on the button:

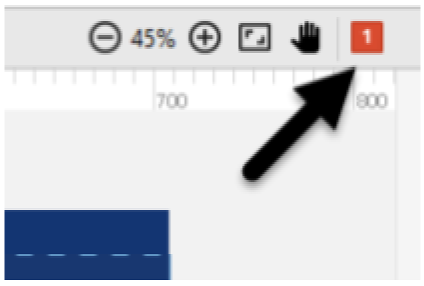
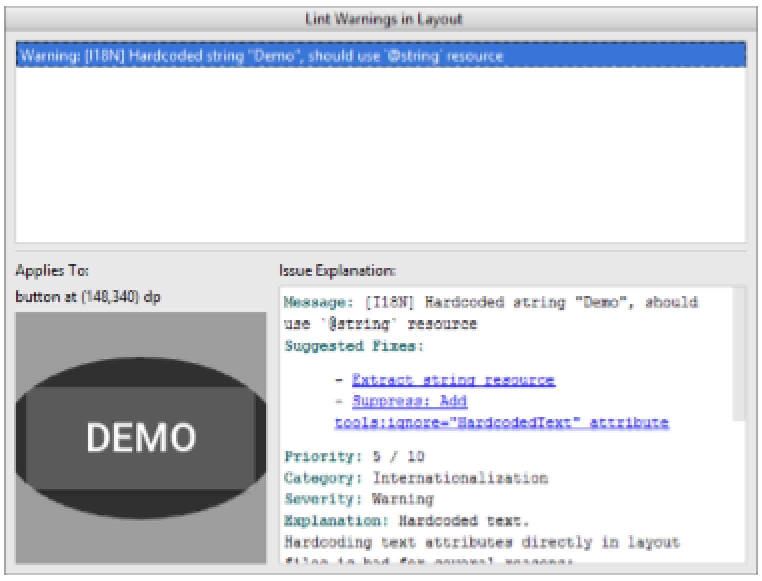


Figure 3-14

When clicked, a panel (Figure 3-15) will appear describing the nature of the problems and offering some possible corrective measures:



Lint Warnings in Layout

Warning: [I18N] Hardcoded string "Demo", should use '@string' resource

Applies To:
button at (148,340) dp

Issue Explanation:

```
Message: [I18N] Hardcoded string "Demo", should
use `@string` resource
Suggested Fixes:

      - Extract string resource
      - Suppress: Add
      tools:ignore="HardcodedText" attribute

Priority: 5 / 10
Category: Internationalization
Severity: Warning
Explanation: Hardcoded text.
Hardcoding text attributes directly in layout
```

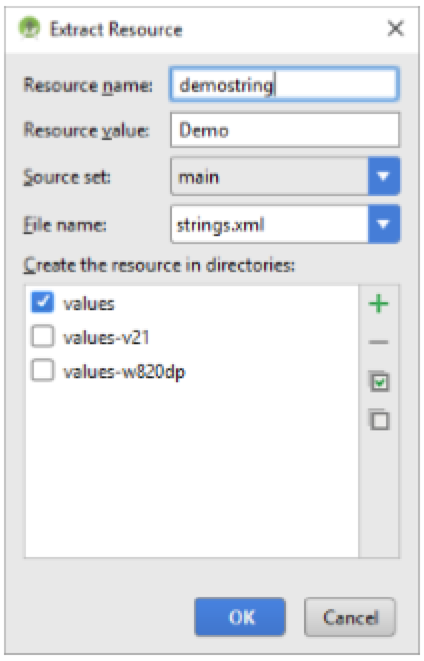
Figure 3-15

Currently, the only warning listed reads as follows:

```
Warning: [I18N] Hardcoded string "Demo", should use '@string' resource
```

This I18N message is informing us that a potential issue exists with regard to the future internationalization of the project ("I18N" comes from the fact that the word "internationalization" begins with an "I", ends with an "N" and has 18 letters in between). The warning is reminding us that when developing Android applications, attributes and values such as text strings should be stored in the form of *resources* wherever possible. Doing so enables changes to the appearance of the application to be made by modifying resource files instead of changing the application source code. This can be especially valuable when translating a user interface to a different spoken language. If all of the text in a user interface is contained in a single resource file, for example, that file can be given to a translator who will then perform the translation work and return the translated file for inclusion in the application. This enables multiple languages to be targeted without the necessity for any source code changes to be made. In this instance, we are going to create a new resource named *demostring* and assign to it the string "Demo".

Click on the *Extract string resource* link in the Issue Explanation panel to display the *Extract Resource* panel (Figure 3-16). Within this panel, change the resource name field to *demostring* and leave the resource value set to *Demo* before clicking on the OK button.



Extract Resource

Resource name: demostring
Resource value: Demo
Source set: main
File name: strings.xml
Create the resource in directories:
- [✓] values
- [ ] values-v21
- [ ] values-w820dp

Figure 3-16

It is also worth noting that the string could also have been assigned to a resource when it was entered into the Properties panel. This involves clicking on the button displaying three dots to the right of the property field in the Properties panel and selecting the *Add new resource -> New String Value…* menu option from the resulting Resources dialog. In practice, however, it is often quicker to simply set values directly into the Properties panel fields for any widgets in the layout, then work sequentially through the list in the warnings dialog to extract any necessary resources when the layout is complete.

3.5 Reviewing the Layout and Resource Files

Before moving on to the next chapter, we are going to look at some of the internal aspects of user interface design and resource handling. In the previous section, we made some changes to the user interface by modifying the *content_android_sample.xml* file using the Layout Editor tool. In fact, all that the Layout Editor was doing was providing a user-friendly way to edit the underlying XML content of the file. In practice, there is no reason why you cannot modify the XML directly in order to make user interface changes and, in some instances, this may actually be quicker than using the Layout Editor tool. At the bottom of the Layout Editor panel are two tabs labeled *Design* and *Text* respectively. To switch to the XML view simply select the *Text* tab as shown in Figure 3-17:

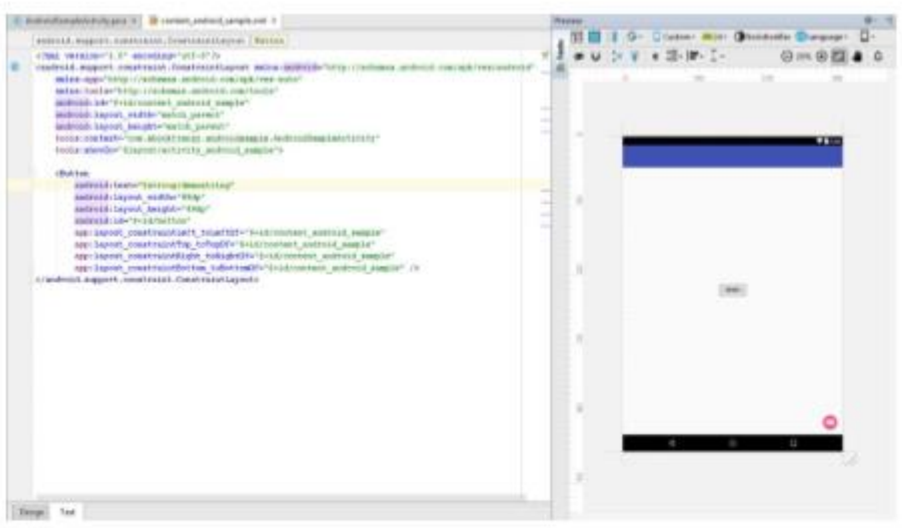


Figure 3-17

As can be seen from the structure of the XML file, the user interface consists of the ConstraintLayout component, which in turn, is the parent of the Button object. We can also see that the *text* property of the Button is set to our *demostring* resource. Although varying in complexity and content, all user interface layouts are structured in this hierarchical, XML based way.

One of the more powerful features of Android Studio can be found to the right hand side of the XML editing panel. If the panel is not visible, display it by selecting the *Preview* button located along the right hand edge of the Android Studio window. This is the Preview panel and shows the current visual state of the layout. As changes are made to the XML layout, these will be reflected in the preview panel. The layout may also be modified visually from within the Preview panel with the changes appearing in the XML listing. To see this in action, modify the XML layout to change the background color of the ConstraintLayout to a shade of red as follows:

```
<?xml version="1.0" encoding="utf-8"?>
android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/content_android_sample"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
 tools:context="com.ebookfrenzy.androidsample.AndroidSampleActivity"
    tools:showIn="@layout/activity android sample"
    android:background="#ff2438" >
.
```

```
</android.support.constraint.ConstraintLayout>
```

Note that the color of the preview changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the left-hand margin (also referred to as the *gutter*) of the XML editor next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Change the color value to #a0ff28 and note that both the small square in the margin and the preview change to green.

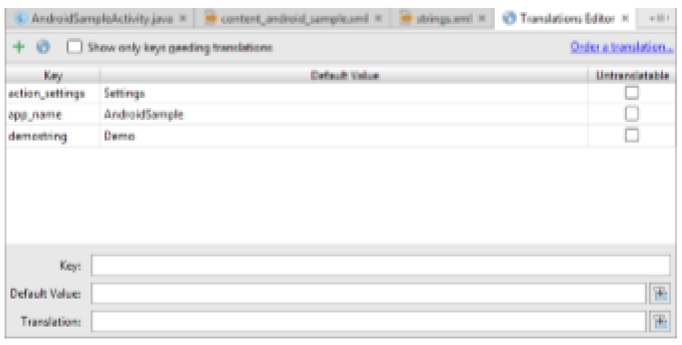
Finally, use the Project view to locate the *app -> res -> values -> strings.xml* file and double click on it to load it into the editor. Currently the XML should read as follows:

```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="action_settings">Settings</string>
    <string name="demostring">Demo</string>
</resources>
```

As a demonstration of resources in action, change the string value currently assigned to the *demostring* resource to "Hello" and then return to the Layout Editor tool by selecting the tab for the layout file in the editor panel. Note that the layout has picked up the new resource value for the welcome string.

There is also a quick way to access the value of a resource referenced in an XML file. With the Layout Editor tool in Text mode, click on the "@string/demostring" property setting so that it highlights and then press Ctrl+B on the keyboard. Android Studio will subsequently open the *strings.xml* file and take you to the line in that file where this resource is declared. Use this opportunity to revert the string resource back to the original "Demo" text.

Resource strings may also be edited using the Android Studio Translations Editor. To open this editor, right-click on the *app -> res -> values -> strings.xml* file and select the *Open Editor* menu option. This will display the Translation Editor in the main panel of the Android Studio window:


AndroidSampleActivity.java
content_android_sample.xml
strings.xml
Translations Editor
Show only keys needing translations
Order a translation...
Key:
Default Value:
Translation:

Key	Default Value	Untranslatable
action_settings	Settings	☐
app_name	AndroidSample	☐
demostring	Demo	☐

Figure 3-18

This editor allows the strings assigned to resource keys to be edited and for translations for multiple languages to be managed. The *Order a translation…* link may also be used to order a translation of the strings contained within the application to other languages. The cost of the translations will vary depending on the number of strings involved.

3.6 Previewing the Layout

So far in this chapter, the layout has only been previewed on a representation of the Nexus 4 device. As previously discussed, the layout can be tested for other devices by making selections from the device menu in the toolbar across the top edge of the Designer panel. Another useful option provided by this menu is *Preview All Screen Sizes* which, when selected, shows the layout in all currently configured device configurations as demonstrated in Figure 3-15:

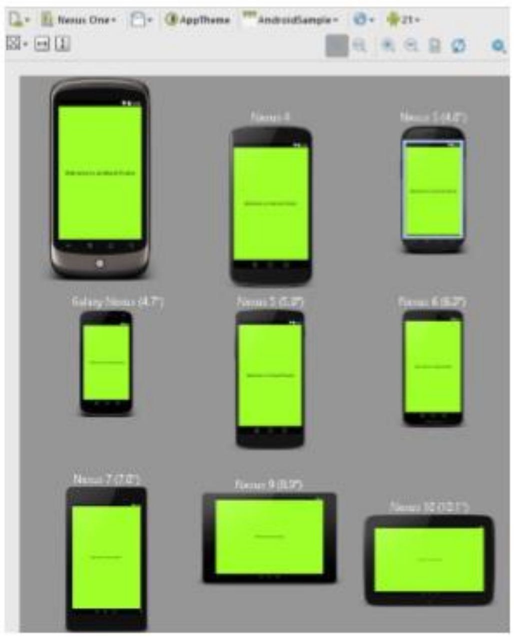


Figure 3-15

To revert to a single preview layout, select the device menu once again, this time choosing the *Remove Previews* option.

3.6 **Summary**

While not excessively complex, a number of steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through a simple example to make sure the environment is correctly installed and configured. In this chapter, we have created a simple application and then used the Android Studio Layout Editor tool to modify the user interface layout. In doing so, we explored the importance of using resources wherever possible, particularly in the case of string values, and briefly touched on the topic of layouts. Finally, we looked at the underlying XML that is used to store the user interface designs of Android applications.

While it is useful to be able to preview a layout from within the Android Studio Layout Editor tool, there is no substitute for testing an application by compiling and running it. In a later chapter entitled *Creating an Android Virtual Device (AVD) in Android Studio*, the steps necessary to set up an emulator for testing purposes will be covered in detail. Before running the application, however, the next chapter will take a small detour to provide a guided tour of the Android Studio user interface.