Android
Programming

Succinctly

by Ryan Hodson

Android Programming

Succinctly

By

Ryan Hodson

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ryan Hodson began learning ActionScript at age 14, which eventually led to a job creating Flash-based data visualizations for the National Center for Supercomputing Applications at the University of Illinois. Since then, he's worked in a diverse collection of programming fields, building everything from websites to e-publishing platforms, touch-screen thermostats, and natural language processing tools. These experiences have led to a love of exploring new software and a proficiency in several languages (HTML/CSS, JavaScript, PHP, MySQL, Python, Java, Objective-C, PDF) and many frameworks (WordPress, Django, CherryPy, and the iOS and OSX SDKs, to name a few).

In 2012, Ryan founded an independent publishing firm called RyPress and published his first book, *Ry's Friendly Guide to Git*. Since then, he has worked as a freelance technical writer for well-known software companies, including Syncfusion and Atlassian. Ryan continues to publish high-quality software tutorials via RyPress.com.

Introduction

Android is an open source operating system built on top of Linux. It powers a plethora of devices, from smart phones to tablets to gaming consoles (along with a vast array of other consumer electronics). According to the International Data Corporation, Android had over 70 percent market share for worldwide smartphones in the last quarter of 2012, so developing for this platform has the potential to reach a very large number of mobile users.

Figure 1: The Android Logo

Android is the main alternative to the iOS platform for mobile applications, but unlike iOS, Android projects can easily be created using OS X, Windows, or Linux-based computers. And since Android uses the Java programming language, developers coming from a C# background will most likely feel more comfortable than they would with iOS's Objective-C programming language.

The goal of *Android Programming Succinctly* is to guide you through the major aspects of Android development with friendly, concise examples. You should walk away with a solid understanding of the necessary design patterns, frameworks, and APIs for producing a polished Android app. If you would like to follow along with the sample code, it can be found here.

Chapter 1   Setting Up

Before we start writing any code, our first task is to set up a development environment. The major components necessary for building an Android app are as follows:

- A text editor for writing your code.

- The Android framework for linking against your application code.

- The Android command-line tools for compiling your code into a working app.

- An emulator or actual device for testing your compiled application.

While it's possible to use virtually any IDE or text editor to create apps, the easiest way to get started with the Android platform is the official Android Software Development Kit (SDK), which contains all of these components in a single convenient download.

The Android SDK

The [Android SDK](#) (available for OS X, Windows, and Linux) includes the Eclipse IDE with the Android Developer Tools (ADT) plugin, along with an emulator, a graphical layout editor, and some other useful features. This is also the development environment that we'll be using in this book, so go ahead and download it now so you can follow along.

Installation

After the download has completed, unzip the file and open the `eclipse` folder. It should contain an `Eclipse` executable that you can launch to start the IDE. You'll be prompted to select a workspace folder, and then Eclipse should be ready to go. And that's it for installation!

Creating a Project

Let's jump right in by creating a new Android project. On the **File** menu, click **New**. In the resulting wizard, select **Android Application Project**.
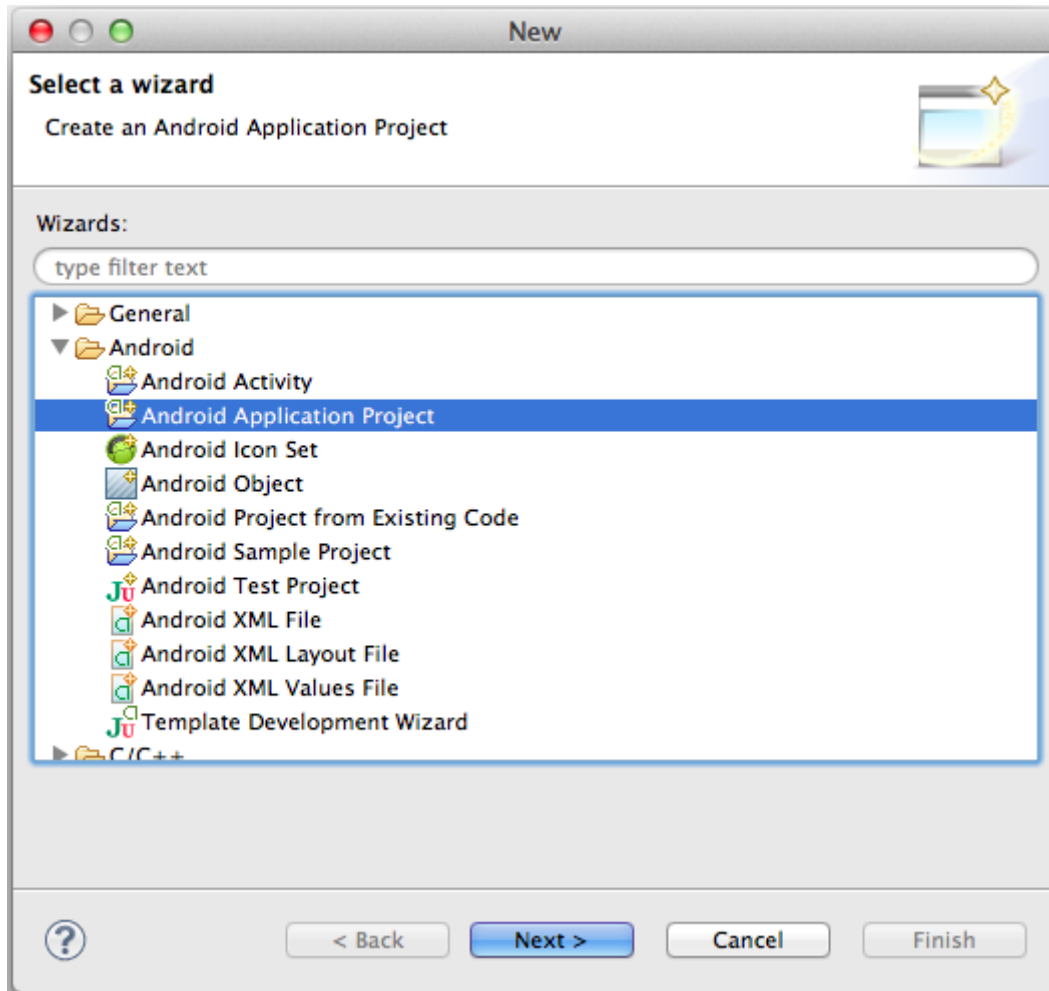
Figure 2: Creating a new Android project

This will prompt you for some information about your new project:

- **Application Name**: The name of your app. Use *Hello Android* for this field.

- **Project Name**: The name of the project directory. This should be automatically populated with *HelloAndroid*, and you can leave this value as is.

- **Package Name**: The unique namespace for the project. Since it's just an example app, you can leave the default *com.example.helloandroid*, but you should use the reverse domain name of your organization for real applications.

The remaining fields define important platform requirements, and you can leave them all at their default values. Your configuration should look like the following when you're done:
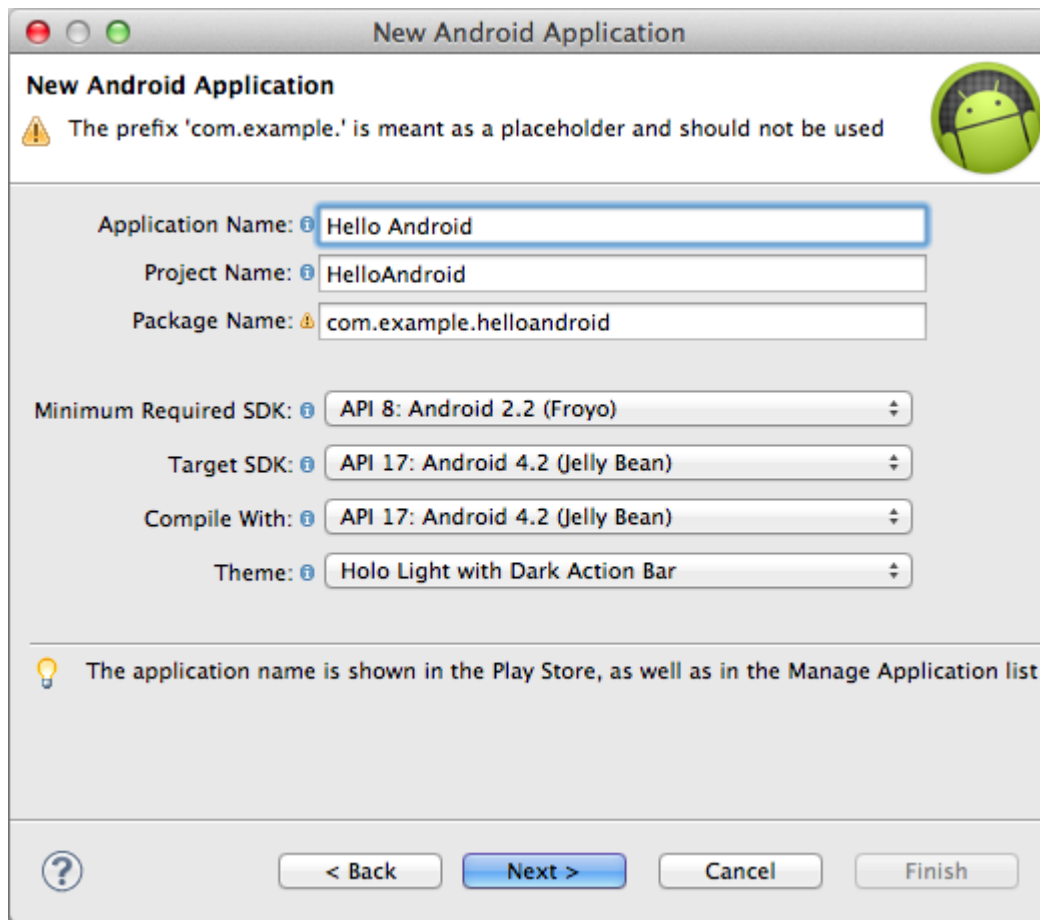
Figure 3: Configuring a new Android project

The next two windows ask you about some other miscellaneous details and the app's icon. You can leave all of them at their default values. Finally, you'll come to the following window asking if you want to create an activity:
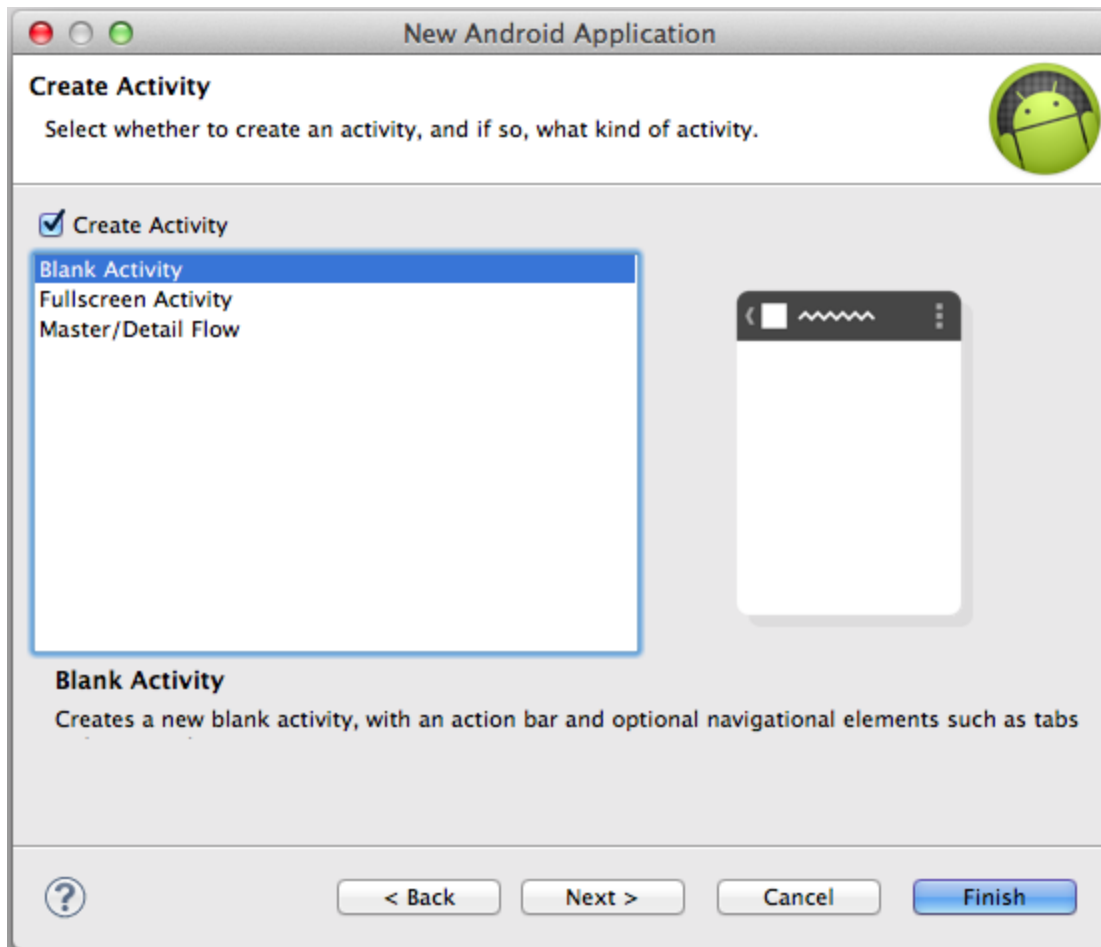
Figure 4: Creating an initial activity

We'll talk about activities in great detail next chapter, but all you need to know for now is that an activity represents a single screen of your application. We want to have something to look at initially, so make sure **Create Activity** is checked, then select **Blank Activity** to specify an empty screen. In the next window, you can use the default **MainActivity** and **activity_main** values for the **Activity Name** and **Layout Name** fields (again, we'll discuss layouts in the next chapter). Click **Finish** to create a brand new Eclipse project.

Setting Up the Emulator

Unfortunately, we can't immediately compile the template project to see what it does. First, we need to set up a device on which to test our new app. Android is designed to let you run a single application on devices of wildly differing dimensions and capabilities, making it a very efficient platform for porting apps from smartphones to tablets to anything in between. The Android Virtual Device Manager included in the SDK allows you to emulate virtually any device on the market.

To view the list of emulated devices, navigate to **Window** and select **Android Virtual Device Manager**. This window makes it easy to see how your application behaves on all sorts of Android devices, test different screen resolutions and dimensions, and experiment with various device capabilities (e.g., hardware keyboards, cameras, storage capacity).

To create an emulated device for our project, click **New...** and use **GalaxyNexus** for the **AVD Name**, then select **Galaxy Nexus** from the **Device** dropdown menu, and leave everything else as the default. For development purposes, it's also a good idea to check the **Use Host GPU** to use your computer's GPU, since emulating animations can be quite slow and clunky. Your configuration should resemble the following:
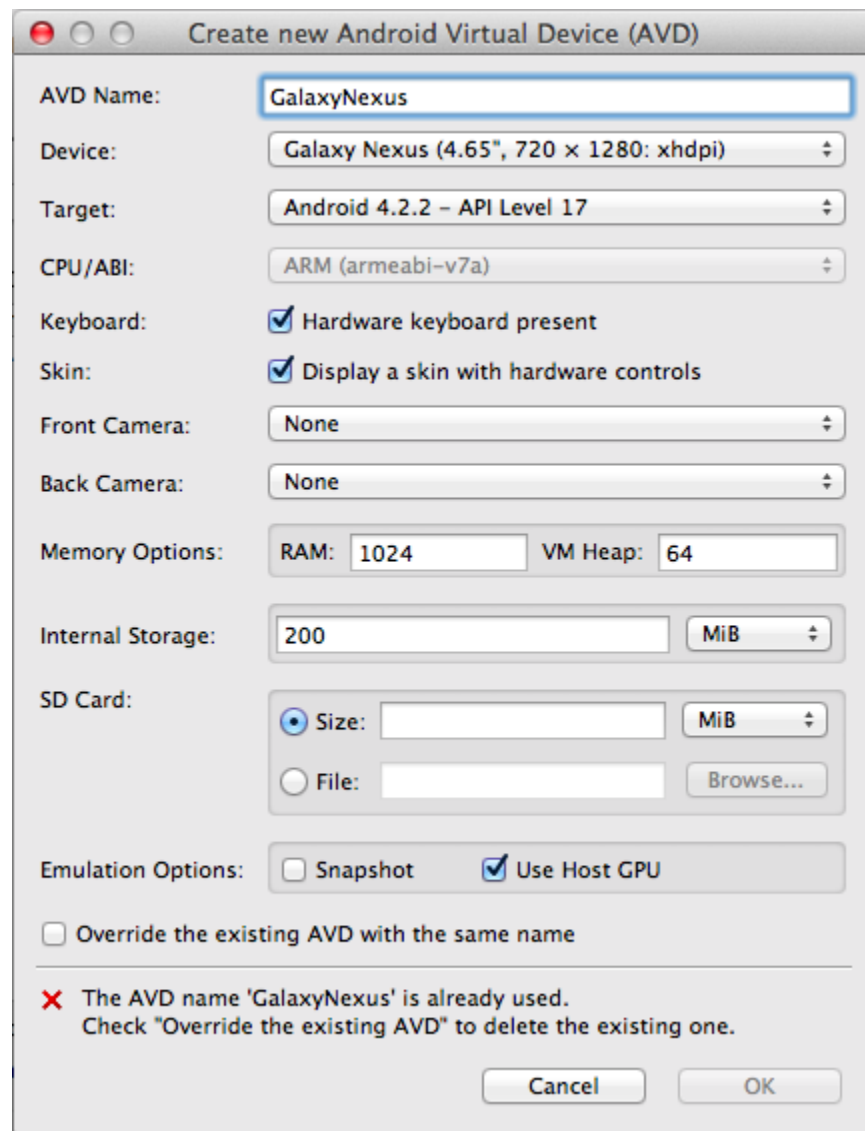
Figure 5: Creating a new emulated device

After clicking **OK**, you should find your device in the **Android Virtual Device Manager** window. To start the emulator, select the **GalaxyNexus** item, and click **Start**. Leave the launch options at their default values, and click **Launch**. This will start spinning up the emulator, which looks something like the following:

Figure 6: The Android device emulator

The emulator has to boot up the Android operating system (just like a real device), so you might be staring at that Android logo for a while before the emulator is actually ready to use. When it is finally ready, you'll see the typical Android home screen, and you can click around to launch apps and explore the emulated device:
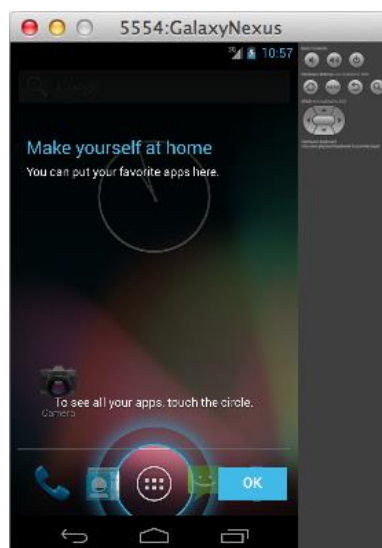
Figure 7: The emulator after it's ready to use

Since it takes so long to boot, you'll want to keep the emulator running as you start writing code (Eclipse can re-launch the application on the emulator without restarting it).

Compiling the Application

We're finally prepared to compile the sample project. Back in Eclipse, make sure one of the source files is selected in the Package Explorer, then click **Run**, select **Run as**, and choose **Android Application**. After taking a moment to compile, you should see your first Android app in the device emulator. As you can see, the default template contains a single text field that says "Hello world!"

Figure 8: The compiled template project

In the next chapter, we'll learn how to change the contents of this text field, add other UI components, and organize a simple Android application.

Chapter 2   Hello, Android

In this chapter, we'll discover the fundamental design patterns of all Android applications. We'll learn how to work with activities, display UI elements by editing XML layout files, handle button clicks, and switch between activities using intent objects. We'll also learn about best practices for representing dimension and string values for maximum portability.

Each screen in an Android app is represented by a custom subclass of Activity. The subclass defines the behavior of the activity, and it's also responsible for loading user interface from an XML layout file. Typically, this XML layout file is where the entire interface for a given activity is defined. To display text fields, buttons, images, and other widgets to the user, all you need to do is add XML elements to the layout file.

Intent objects are used to switch between the various activities that compose your app. For example, when the user clicks a button to navigate to another screen, you create an `Intent` object and pass the destination activity to it. Then, you "execute" the intent to tell Android to switch to the next page. This is the general pattern for building up a multi-screen application.

This chapter explains how all these android architecture components interact using hands-on examples. You can follow along with the empty Android project that you created in the previous chapter, or you can view the completed code in the *HelloAndroid* application included with the sample code for this book.

App Structure Overview

Let's start by taking a look at the files and directories our template project created for us. It may look like there are a lot of files in Eclipse's Package Explorer, but don't be overwhelmed—there are only three items that you need to worry about:

`AndroidManifest.xml` – This file declares everything about your app that the Android operating system needs to launch it. This includes the classes composing the application, the permissions required by your app, meta information like the minimum Android API for your app, and the libraries that your app depends on. This must be in the root directory of your project. Typically, the SDK will update this file for you automatically, so you shouldn't have to edit it directly.

`src/` – This directory contains all of your app's source files. This is where all of your Java source code will reside.

`res/` – This folder contains application **resources**, which are the images, videos, and strings that will be displayed to the user. By abstracting resources from your application code, it's trivial to add support for new screen resolutions, orientations, and languages.

The manifest and source directory are relatively straightforward, however the resource folder calls for a bit more explanation. If you expand this folder in the Package Explorer, you'll find three types of subdirectories: `drawable/`, `layout/`, and `values/`. The `drawable/` folders contain all of the graphics for your application, which can be either image files or special XML files defining shapes. The `layout/` directory stores the UI of each screen displayed by your application, and the `values/` folder contains lists of strings that are utilized by the UI.

Now, the interesting part is what comes after the hyphen. This portion of the folder name is a **qualifier** that tells the system when to use the contained resources. For example, images in `drawable-hdpi/` will be displayed when the host device has a high-density screen (~240dpi), whereas devices with low-density screens (~120dpi) will use the images in `drawable-ldpi/`.

By simply placing high-resolution images and low-resolution images in the appropriate folders, your app will be available to both types of devices with no changes to your code. Similarly, your app can be ported to other languages by appending `en`, `es`, `fr`, `ar`, or any other language code after the `values/` subdirectory. In this way, the `res/` directory makes it very easy to support new devices and reach new audiences.

Creating a User Interface

Android apps use XML to define their user interfaces. Each XML element in a layout file represents either a [ViewGroup](#) or a [View](#) object. `ViewGroups` are invisible containers for other `View` objects, and their main job is to arrange child views (or nested view groups) into a pleasing layout. `View` objects are visible UI components like text fields, buttons, and tables. To configure the properties of a view or view group, you edit the attributes of the corresponding XML element.

The template project we used comes with a default layout called `activity_main.xml`, which you should find in the `res/layout/` directory. When you double-click the file, ADT displays the graphical UI editor by default. While you can use this to visually edit the underlying XML, for this book, we'll be directly editing the XML markup to gain an in-depth understanding of how Android user interfaces work. To display the raw XML, click the **activity_main.xml** tab in the bottom-left of Eclipse's editing area, highlighted in orange as you can see in the following figure:
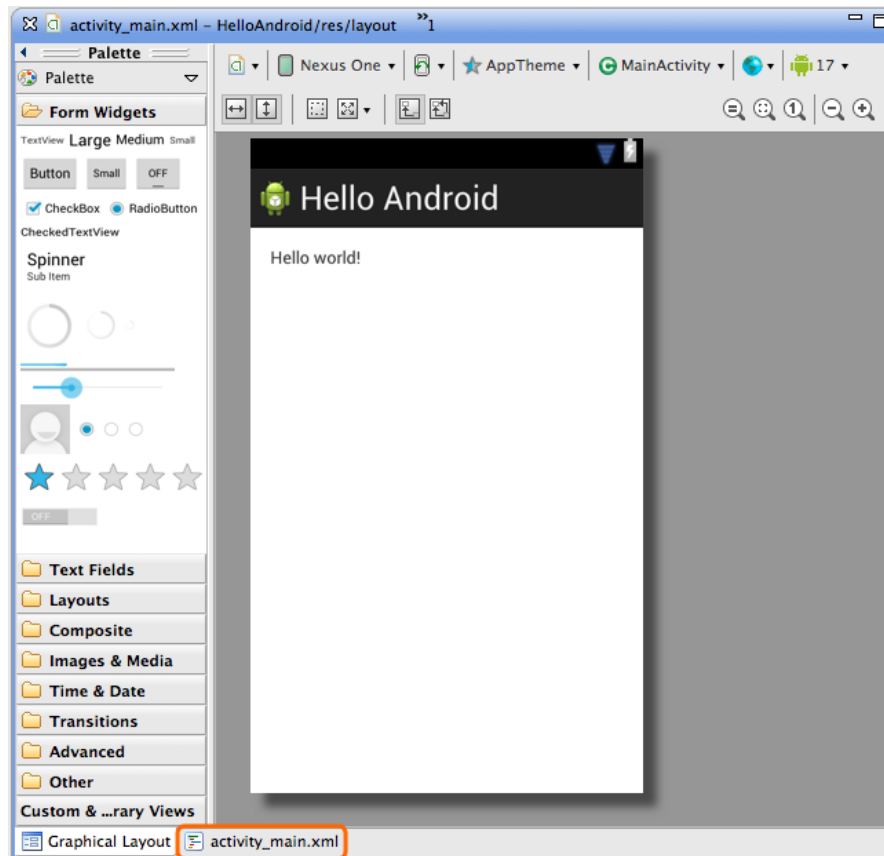
Figure 9: The raw XML tab for the main layout

After clicking this tab, you'll find the XML that defines the current layout. It should look something like the following:

```xml
<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" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

</RelativeLayout>
```

As you can see, this defines two elements: `<RelativeLayout>` and `<TextView>`. RelativeLayout is a subclass of `ViewGroup` that defines the position of its children relative to each other. TextView is a subclass of `View` that represents a standard text field component. We'll survey the most common UI elements and later in this book.

All of the attributes in the `android` namespace determine various properties of the associated element. For example, `android:layout_width="match_parent"` makes the RelativeLayout stretch to the same size as its parent (in this case, the main window of the app). The available attributes and values for each type of element are listed in the documentation for the corresponding class (e.g., View, ViewGroup, RelativeLayout, TextView). The value of `android:text` is a reference to a string resource, which we'll explain in a moment.

Adding a Button

But first, we're going to simplify the default layout a bit and change it to a single button, centered on the screen. Replace the existing `activity_main.xml` with the following:

```xml
<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:gravity="center"
    tools:context=".MainActivity" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_next" />

</RelativeLayout>
```

We've done two things here. First, we replaced the `<RelativeLayout>`'s padding attributes with `android:gravity`. This tells it to center all of its child views. Second, we changed the `<TextView>` element to a `<Button>` and gave a new value to its `android:text` attribute.

Unfortunately, you won't be able to compile the project to see the button just yet. Instead, you'll see an error message next to the `android:text` attribute saying something about a missing resource, which can be seen in the following figure:

> **NOTE: This project contains resource errors, so aapt did not succeed, which can cause rendering failures. Fix resource problems first.**
>
> Couldn't resolve resource @string/button_next

Defining String Resources

When designing layouts for Android, you'll rarely want to hardcode the value of button titles, labels, and other text fields into the layout's XML file. Instead, you define **string resources** in a separate file and link to it from within the layout. This extra layer of abstraction makes it possible to reuse the same layout file with different string resources. For instance, you can display shorter instructions when in portrait mode versus landscape mode, or display English, German, Arabic, or Chinese instructions based on the device's locale.
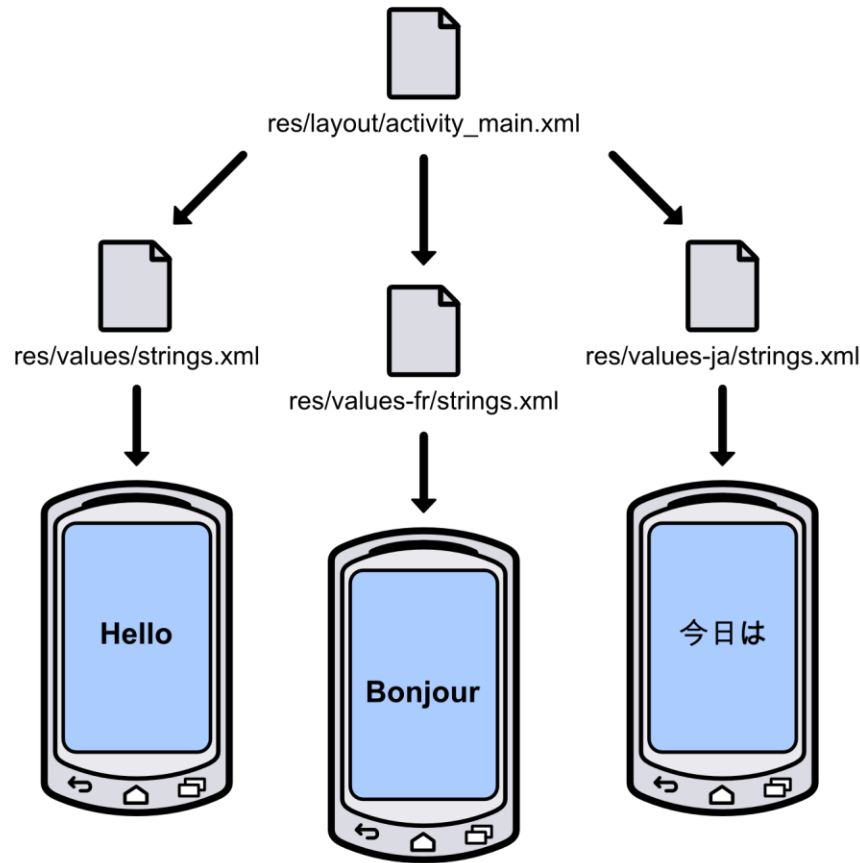
Figure 10: Displaying different string resources in the same layout

Whenever you want to link to any external resource, you use the following format: @<type>/<identifier>. For example, the android:text="@string/button_text" attribute in activity_main.xml references a string resource with an identifier of button_text. But, since we haven't created that resource yet, the compiler can't find it, so it gives us an error. Let's fix this now.

In the **res/values/** directory, open up the **strings.xml** file. This is the conventional place to define strings that will be displayed to the user (the filename is actually arbitrary, as the resource is fully defined by the contained XML). You'll find a few <string> elements inside of a <resource> element. Go ahead and add the following element:

```
<string name="button_next">Next Page</string>
```

The name attribute defines the identifier for the resource. Eclipse should stop complaining about the missing resource, and you should now be able to successfully compile your project. In the emulator, you should see a button in the center of the screen with *Next Page* as its title:
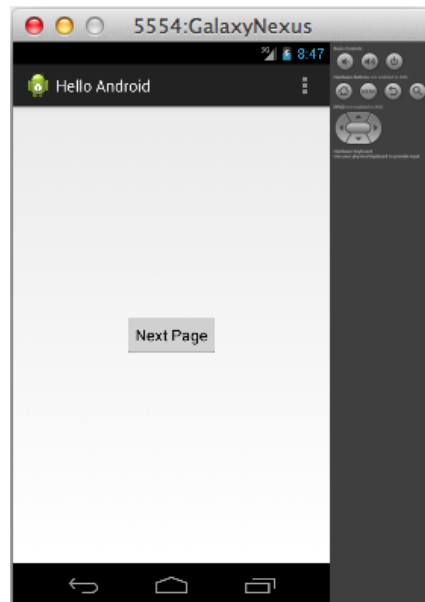
Figure 11: Adding a button to the UI

Separating UI strings from layout construction may seem like an unnecessary hassle, but once you start working with more complicated user interfaces and want to support multiple languages, you'll really appreciate the convenience of having all of your strings in one place.

Detecting Button Input

So, we have a button that the user can interact with, and it even has the iconic blue highlight when you tap it. The next step is to detect the tap. This is accomplished via the android:onClick attribute, which tells the system to call a method of the associated activity whenever the button is tapped. This binds the layout layer with the behavior one, as you can see in the following figure.
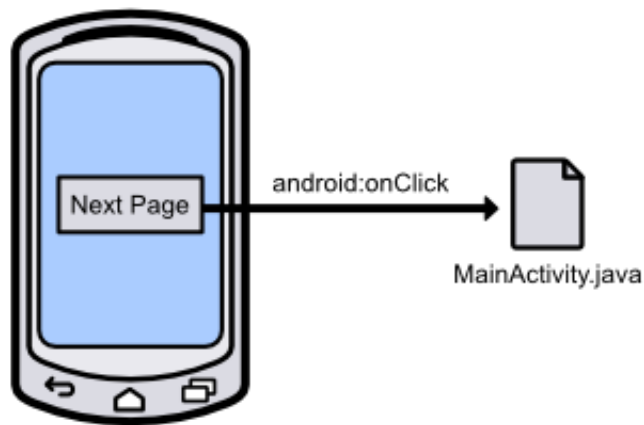
Figure 12: Detecting button clicks with the android:onClick attribute

In `activity_main.xml`, change the `<Button>` element to the following. This will hook up the button to the `nextPage()` method of `MainActivity.java` (which we still need to implement).

```xml
<Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_next"
        android:onClick="nextPage" />
```

Logging Output

To figure out if our button is working, let's log a message to the console whenever the button is tapped. Instead of Java's conventional `System.out.println()`, we'll use the [Log](#) class that is included with the Android platform. This is the preferred way to output messages to the console in Android apps due to its well-defined priority levels, which let you easily filter different types of messages. The most common `Log` methods are listed below from highest priority to lowest:

- `Log.e(String tag, String msg)` – Log an *error* message.

- `Log.w(String tag, String msg)` – Log a *warning* message.

- `Log.i(String tag, String msg)` – Log an *informational* message.

- `Log.d(String tag, String msg)` – Log a *debug* message.

Having several levels of message logging makes it possible to filter certain messages depending on your compile target. For example, debug messages are stripped at runtime from production apps, making them the best choice to use during development. Errors, warnings, and informational messages are always logged, even in production apps.

The first parameter of all these methods is a string that identifies the source of the log message. You'll typically want this to be the name of the class that's doing the logging, and the conventional way to define this value is with a private static final variable in the class itself. So, in `MainActivity.java`, add the following line at the beginning of the class definition:

```
private static final String TAG = "MainActivity";
```

And, as with any other Java class, we need to import the `Log` class before we can use it, so add the following import statements to the top of `MainActivity.java` (we'll need the `View` class, too):

```
import android.util.Log;
import android.view.View;
```

Then, implement the `nextPage()` method to output a debug message, like so:

```
public void nextPage(View view) {
      Log.d(TAG, "Switching to next page");
}
```

Remember that this is the name of the method that we hooked up in `activity_main.xml`, so it should get called every time the user clicks the button. Any method used as an `android:onClick` target needs to have the above signature (that is, it must be `public`, return `void`, and accept a single `View` object as a parameter). The parameter is a reference to the UI component that initiated the call, which in this case is the button that the user pressed.

You should now be able to compile your project, tap the button, and see the above message in the LogCat panel at the bottom of the IDE. This is where all log messages will be displayed when launching your application from within Eclipse. To display the LogCat panel, click **Window**, **Show View**, **Android**, and select **LogCat**.
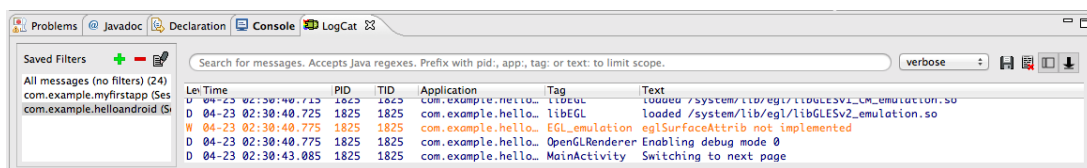
Figure 13: The LogCat panel in Eclipse

Creating Another Activity

Logging is a convenient way to make sure UI components are working properly, but you're probably going to want most of your buttons to do more than just display a message to the console. To extend our example, we'll switch to another activity when the user presses the button.

The first step is to create another class for the second activity. In Eclipse, press **Cmd+N** (or Ctrl+N if you are on a PC) to open the New Document wizard and select **Android Activity** to add a new class to the current project. Choose **Blank Activity** for the template. In the next window, use **SecondActivity** for the **Activity Name** field, **activity_second** for **Layout Name**, and **Second Activity** for the **Title** field. Your configured activity should look like the following:
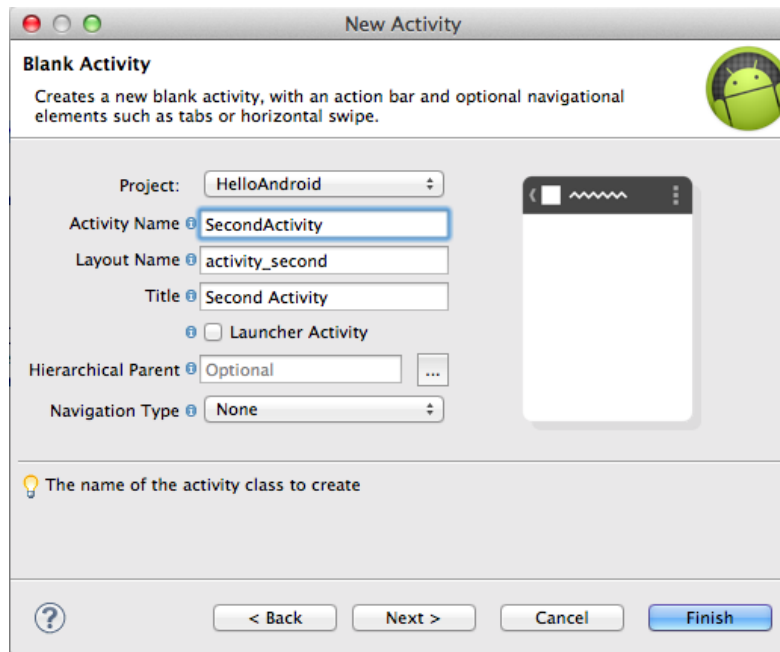
Figure 14: Configuring a new Activity class

Clicking **Finish** will create the `SecondActivity.java` and `activity_second.xml` files, add them to the manifest, and add a string resource defining the activity title in `res/values/strings.xml`. This is everything that needs to happen for a new activity to be ready to use.

Linking Activities With An Intent

The user experience of a single Android application is typically made up of several different activities (e.g., browsing the user's contact list, editing a contact, and viewing a contact are all separate activities, but are all part of the People app). Each activity is implemented as a completely independent component, even if they are part of the same application. This makes it possible for any activity to be initiated by any other activity, including ones from other applications. For example, if your application needs to add a new contact to the user's contact list, it can jump directly to the *People* app's activity for creating new entries.

To glue all of these independent activities together into a coherent app, Android provides an Intent class, which represents an arbitrary action to be performed. The general pattern is to create an `Intent` object, specify the destination class, and pass in any data required for the destination class to perform the action. The following figure shows you how the intent object links two activities.
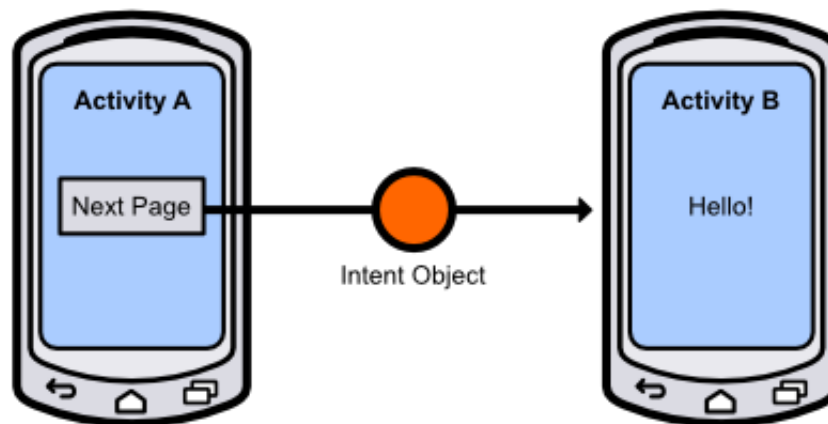
Figure 15: Switching from one activity to another using an Intent object

This loose-coupling between activities makes for a very modular application architecture with flexible code reuse opportunities.

In our example, we want to create an intent inside the `MainActivity` class and use `SecondActivity` as the destination. We'll worry about the data passing portion at the end of the chapter. First, import the `Intent` class into `MainActivity.java`:

```
import android.content.Intent;
```

Then, change the `nextPage()` method to the following:

```
public void nextPage(View view) {
      Intent intent = new Intent(this, SecondActivity.class);
      startActivity(intent);
}
```

After creating the intent, we can pass it to the built-in `startActivity()` function to execute it. The result should be a transition to the second activity when you click the Next Page button in the first activity. Right now, the second activity is just a static text field that says "Hello world!", but we'll change that in the upcoming section.

Another Button

In this section, we'll add another button to the main activity, then we'll pass the selected button to the second activity for display. Change `activity_main.xml` to the following:

```xml
<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="horizontal"
    tools:context=".MainActivity" >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_red"
        android:onClick="nextPage" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_blue"
        android:onClick="nextPage" />

</LinearLayout>
```

Note that both buttons will call the same `nextPage()` method when the user taps them. This requires two new string resources: `button_red` and `button_blue`. In `strings.xml`, go ahead and add these resources:

```xml
<string name="button_red">Red Pill</string>
<string name="button_blue">Blue Pill</string>
```

You can delete the `button_next` element since we'll no longer be using it. Before we start passing data with `Intent` objects, lets make sure we can tell which button was pressed using a simple `Log` message. In `MainActivity.java`, change the `nextPage()` method to the following:

```java
public void nextPage(View view) {
    Intent intent = new Intent(this, SecondActivity.class);
    Button button = (Button)view;
    String message = button.getText().toString();
    Log.d(TAG, message);
    startActivity(intent);
}
```

All this does is fetch the title of the button via the `getText()` method and displays it via a `Log.d()` call. You'll also need to import the `Button` class at the top of the file:

```java
import android.widget.Button;
```

Now, when you compile the app, you should see two buttons in the top-left corner.
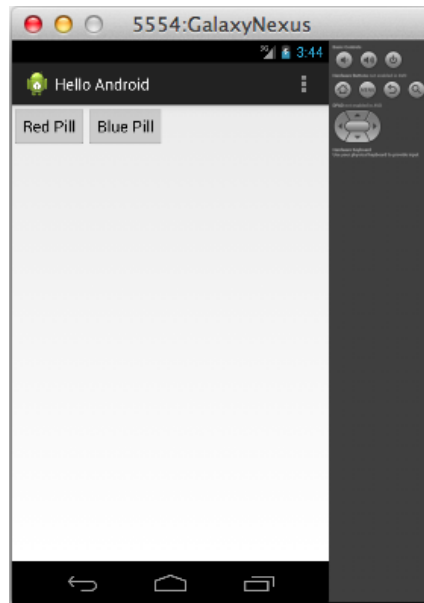
Figure 16: Adding another button to the main activity

When you tap one of the buttons, you should see the corresponding title displayed in the LogCat panel.

Passing Data with Intents

Next, we're going to pass this information along to the next activity using our existing `Intent` object. You can store data in an `Intent` instance by calling its `putExtra()` method, which takes two parameters: a name and a string. You can think of this as creating a key-value pair on the `Intent` object. For example:

```java
public void nextPage(View view) {
      Intent intent = new Intent(this, SecondActivity.class);
      Button button = (Button)view;
      String message = button.getText().toString();
      intent.putExtra(EXTRA_MESSAGE, message);
      startActivity(intent);
}
```

The `intent.putExtra(EXTRA_MESSAGE, message);` line adds the button's title to the `Intent` object, which we can later retrieve via the `EXTRA_MESSAGE` constant. We still need to define this constant, so be sure to include the following at the top of the `MainActivity` class:

```java
public static final String EXTRA_MESSAGE =
"com.example.helloandroid.MESSAGE";
```

We've now successfully encoded some data to send from the main activity to the second activity. Next, we need to allow the second activity to receive this information. This requires two steps:

1. Get the Intent instance that was sent to the second activity.

2. Use the EXTRA_MESSAGE key to return the associated value.

To do both of these, add the following method to `SecondActivity.java`:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_second);

    Intent intent = getIntent();
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
    Log.d(TAG, message);
}
```

The `onCreate()` method is a special method defined by the `Activity` class. The Android framework invokes `onCreate()` whenever an `Activity` is started, and by overriding it in subclasses, we can add custom initialization behavior. For now, you can think of it as the constructor method for an `Activity` instance, but we'll refine this concept a little bit in the next chapter.

The last three lines fetch the `Intent` data we sent from `MainActivity.java` and display it with `Log.d()`. The `getIntent()` method is another method defined by the `Activity` class which simply returns the `Intent` instance that started the activity. Then, we use the [Intent] class's `getStringExtra()` method to find the value associated with the `MainActivity.EXTRA_MESSAGE` key. Note that defining the key as a constant in `MainActivity.java` is a best practice, as it lets the compiler make sure the key is typed correctly. There are similar methods for other data types (e.g., `getFloatExtra()` returns a `float` value).

The above method still needs access to the `Intent` and `Log` classes, so add the following lines to the top of `SecondActivity.java`:

```java
import android.content.Intent;
import android.util.Log;
```

We also need to define another `TAG` variable in `SecondActivity.java`:

```java
private static final String TAG = "MainActivity";
```

You should now be able to compile the project. Click either of the buttons, and have `SecondActivity` log the selected button title:
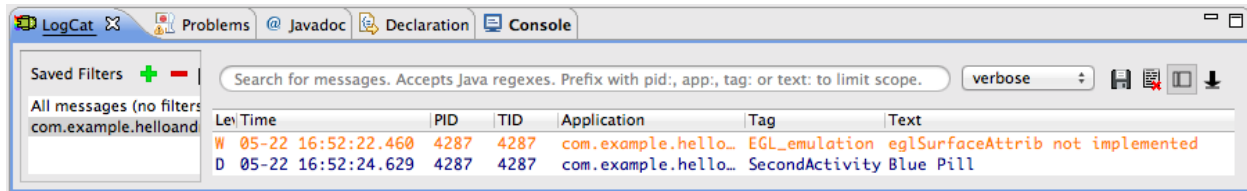
Figure 17: Logging the selected button's title

For completeness, let's make the text field in `SecondActivity` display the value instead of just displaying it in LogCat. First, add an `id` attribute to the `TextView` in `activity_second.xml`:

```xml
<TextView
        android:id="@+id/selected_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
```

The `id` attribute is necessary when you want to access a UI element from elsewhere in the application. The plus sign in `@+id` is only necessary when *first* defining the UI element (if you wanted to reference it from another UI file, you would not need it). The `selected_title` portion defines the unique ID of the element, which can be used as follows (define this in `SecondActivity.java`):

```java
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_second);

      Intent intent = getIntent();
      String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
      TextView textField = (TextView)findViewById(R.id.selected_title);
      textField.setText(message);
}
```

The global `findViewById()` method is defined by the `Activity` class, and it returns a `View` instance that has the supplied ID. Note that we have to cast the return value to a `TextView` so we can use the `setText()` method to display the selected button. The `R.id.selected_title` snippet is the conventional way to refer to UI components defined in XML files. The `R` class is generated automatically when you compile your project, and the framework populates it with constants for each element with an `android:id` attribute. This method of using static constants to refer to UI elements eliminates the possibility of referencing undefined elements (e.g., from a misspelled or omitted ID in the layout file).

Note that you'll have to import the `TextView` class for the above `onCreate()` method to compile:

```java
import android.widget.TextView;
```

One common mistake for Android beginners is to call `findViewById()` before calling `setContentView()`. The latter must be called *first*, otherwise `findViewById()` will return `null`.

The second view should now display either "Red Pill" or "Blue Pill" depending on which button you tapped in the main activity page. To switch between buttons, use the Back button on the device to return to the main page and select the one you want to test.   As you can see in the following image, selecting the Blue Pill button will display its text.

Figure 18: Displaying the selected button title in the second activity

Summary

In this chapter, we learned about the basic structure on an Android application, how to create user interfaces with XML files, define media resources, handle user input, and switch between activities using an `Intent` object. We also acquired some practical debugging skills by taking a brief look at the `Log` class.

This is almost everything you need to know to develop basic Android apps. The next chapter explains the lifecycle of an `Activity` in more detail, which is important to properly manage the memory footprint of your application. The rest of this book explores the intermediate functionality that makes apps more interactive, data-driven, and user-friendly.

Chapter 3   The Activity Lifecycle

Once you have the basic structure of an Android application down, the next step is to understand the intricacies behind constructing and destroying `Activity` objects. We already saw how `onCreate()` can be used to initialize an `Activity`, but this is only one aspect of managing an activity's lifecycle. In this chapter, we'll learn how to minimize CPU, memory, and battery usage, handle phone call interruptions, save user state, and switch between landscape and portrait orientations by properly implementing an activity's lifecycle. This is a very important aspect of Android app development, as a failure to do so will cause your application to regularly crash (and that's a very bad user experience). The following figure shows you the activity lifecycle of an application.
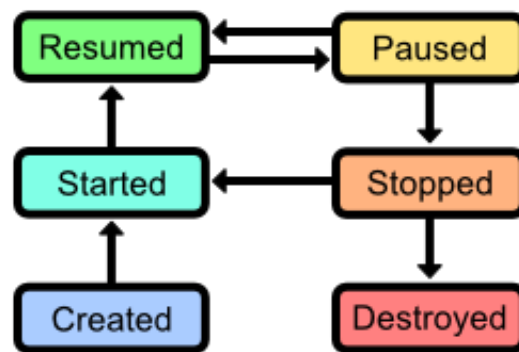
Figure 19: The activity lifecycle

An activity lifecycle consists of the following six states:

- **Created**: The activity has been created and is ready to be displayed.

- **Started**: The activity is visible, but the user cannot interact with it yet. This state is typically followed immediately by the *resumed* state.

- **Resumed**: The activity is running in the foreground and the user can interact with it.

- **Paused**: The activity has been interrupted by a phone call or dialog message (e.g., a low-battery alert). This often leads immediately into the *stopped* state. The activity is usually still visible while paused, but obscured by a dialog so the user cannot interact with it.

- **Stopped**: The activity has been moved to the background and is no longer visible, but the instance still exists in memory. An activity can be re-launched from this state without re-creating it.

- **Destroyed**: The activity has been released by the system and no longer exists. This will happen automatically when the Android operating system deems it necessary.

When an activity is being created, it passes through the first three states, and when it is being destroyed, it passes through the latter half of the states. However, this rarely happens in a strictly linear fashion. The typical application will switch between the *started*, *resumed*, *paused*, and *stopped* state as the user interacts with the other activities in your application and gets interrupted by important alerts.

To manage the transitions between these states, the `Activity` class provides several methods. To define custom startup and teardown behavior, all you have to do is override these methods in an `Activity` subclass. All of the `Activity` transition methods are listed below, along with common examples of when they should be used:

- `onCreate()` – Called when the activity is entering the *created* state. As we saw in the last chapter, this is where you want to initialize UI elements and otherwise prepare the activity for use.

- `onStart()` – Called when the activity is entering the *started* state. This is a good place to load data that needs to be displayed to the user, though this can also be done in `onResume()` depending on the type of data and how you're using it.

- `onResume()` – Called when the activity is entering the *resumed* state. This is the best place to initialize sensor input (e.g., GPS, cameras) and begin any animations required by your user interface.

- `onPause()` – Called when the activity is entering the *paused* state. This is where you should stop using scarce system resources (e.g., animations, GPS, cameras) to maximize the device's battery life and to reduce your application's memory footprint. This is the teardown counterpart to the `onResume()` method.

- `onStop()` – Called when the activity is entering the *stopped* state. This is called right before the application enters the background, so it's a good place to save user data that needs to be re-used later on (e.g., an email draft). Note that `onPause()` can also be an appropriate time to do this, as it is always called immediately before `onStop()`. Whether you want to use `onPause()` or `onStop()` largely depends on your specific application requirements. `onStop()` is the teardown counterpart to the `onStart()` method.

- `onDestroy()` – Called when the activity is entering the *destroyed* state. This is the last chance you have to clean up any resources that would otherwise leak when your application is destroyed. This is the teardown counterpart to the `onCreate()` method; however, the system will automatically release class references when the activity is destroyed, so you usually won't need to implement `onDestroy()` unless you started a background thread or created a memory buffer in `onCreate()`.

As you can see, there is a symmetry between the above transition methods. Anything that happens in `onStart()` should be undone in `onStop()`, and anything done in `onResume()` should be undone in `onPause()`.

When overriding any of the above methods, remember that you *must* call the superclass's version for your application to work correctly.

Common Activity Transition Events

If you're feeling a little confused about all of these transitions, it might help to discuss them from the perspective of the user. In this section, we've included a few of the most common events that can trigger an activity state transition below. Again, all of these occur frequently in the normal usage of an application, so it's imperative to ensure that the corresponding transition methods consume and release system resources efficiently.

Pressing the Power Button

When the user presses the device's power button to put it in standby, the current activity will be moved to the background (that is, it will *pause*, then *stop*). As you might expect, the opposite process occurs when they exit standby: the current activity will *start*, then *resume*.

If you think about how many times you've entered standby on your own Android device while in the middle of an activity, you'll quickly understand how important it is to properly manage the `onPause()`, `onStop()`, `onStart()`, and `onResume()` methods. For example, if you left the accelerometer sensor running after the device entered standby, the user would find their battery unexpectedly low when they turned their phone back on.

Rotating the Device

The way the Android system handles device orientation changes can be somewhat counterintuitive, especially for new developers. When the screen rotates, the foreground activity is actually destroyed and re-created from scratch. This is because some layouts need to load different resources for portrait vs. landscape modes, and using the same instance is potentially wasteful.

Tapping the Back Button

When the user taps the Back button, the operating system interprets this as no longer needing the current activity, so it destroys it instead of just sending it to the background. This means that if the user navigates back to the destroyed activity, it will be created from scratch. If you're trying to record the user's progress in the destroyed activity, this means that you need to store that information *outside* of the activity and reload it in `onCreate()`, `onStart()`, or `onResume()`. If you try to store any data as instance variables, it will be lost when the activity is destroyed.

Recreating Destroyed Activities

An activity that is destroyed when the user taps the Back button or when it manually terminates itself is gone forever; however, this is not the case when an activity is destroyed due to system constraints. For example, if an activity is about to be destroyed because the Android OS needs the memory, it first archives the **instance state** of the Activity in a [Bundle](#) object, saves it to disk, and associates the Bundle with the Activity subclass.

This Bundle object can then be used to create a new Activity object with the same state as the destroyed one. Essentially, this makes it appear as though the original Activity instance is always in a *stopped* state without consuming any resources whatsoever.

Restoring Instance State

If there is an associated Bundle for an Activity, it gets passed to its onCreate() method. For instance, if you were developing an email client and stored the email body in an instance variable called message, your onCreate() method might look something like the following:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    if(savedInstanceState != null) {
        // Restore the archived instance state
        this.message = savedInstanceState.getString(STATE_MESSAGE);
    } else {
        // Initialize with default values
        this.message = "Hello, World!";
    }
}
```

This would also require the following two lines at the top of the class:

```java
private static final String STATE_MESSAGE = "MainActivityMessageState";
public String message;
```

STATE_MESSAGE is a constant that defines the key to use when archiving the message instance variable in the bundle. This is similar to the key-value pattern for storing information in an Intent object.

Saving Instance State

To archive custom instance variables, all you need to do is override the onSaveInstanceState() method defined by Activity. For example, to save the above message state, you would use something like this:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putString(STATE_MESSAGE, this.message);
    super.onSaveInstanceState(savedInstanceState);
}
```

As with the `Activity` state transition methods, it's important to call the superclass version of `onSaveInstanceState()` when overriding it.

View States

The Android framework will automatically save the state of the entire view hierarchy, which means that every text field, button, and image will look exactly as it did before it was destroyed by the system. For many activities, this is the only state that really needs to be saved—one of the few times you'll actually need to implement a custom `onSaveInstanceState()` is when the `Activity` has multiple steps and the user's progress is recorded in instance variables.

Example Project

The `ActivityLifecycle-transitions` project in the resource package for this book is a simple application demonstrating all of the transition methods discussed above. It has two activities that you can switch between, and both of them use `Log.d()` to inform you when they have changed states. When you run the project in the emulator and click the buttons, you should see the following messages in LogCat:

```
05-23 12:27:41.178: D/MainActivity(4042): Created Main Activity
05-23 12:27:41.178: D/MainActivity(4042): Started Main Activity
05-23 12:27:41.178: D/MainActivity(4042): Resumed Main Activity
// Click the "Next Activity" button
05-23 12:27:44.788: D/MainActivity(4042): Paused Main Activity
05-23 12:27:45.018: D/SecondActivity(4042): Created Second Activity
05-23 12:27:45.018: D/SecondActivity(4042): Started Second Activity
05-23 12:27:45.018: D/SecondActivity(4042): Resumed Second Activity
05-23 12:27:45.728: D/MainActivity(4042): Stopped Main Activity
```

Also notice that if you press the Power button on the emulator (which emulates the physical power button on an actual device), you'll see the current activity pause and then stop. You can also see the current activity get destroyed when you press the emulator's Back button.

Of course, you'll generally want to do more than just log a message in `onCreate()`, `onStart()`, and the other activity transition methods, but this project does give you a convenient place to start experimenting with the various states of an `Activity`. We'll see some more practical versions of these methods later in the book after we learn how to create animations and save user data.

Summary

In this chapter, we introduced the lifecycle on an `Activity` object. As the user navigates an application, each `Activity` object passes through a *created*, *started*, *resumed*, *paused*, *stopped*, and *destroyed* state, often cycling between the middle four states several times before being destroyed. Since these state transitions happen so frequently, and mobile devices have such scarce system resources, properly transitioning between these states is an essential component for a satisfying user experience.

This chapter focused more on the conceptual aspects on the `Activity` lifecycle because the concrete implementation of transition methods like `onResume()` is so application-specific. What you should take away from this chapter is a high-level understanding of how the Android framework manages its `Activity` objects. Once you understand that, it's easy to adapt these concepts to your real-world application requirements.

In the next two chapters, we'll shift gears a bit and focus on configuring the front end of an Android application. First, we'll learn how to arrange UI elements into user-friendly layouts, then in the following chapter, we'll explore the API for buttons, text fields, spinners, and the other common input controls.

Chapter 4   User Interface Layouts

Laying out the UI elements in each `Activity` is one of the most important aspects of Android app development. It defines the appearance of an app, how you collect and display information to your users, and how they navigate between the various activities that compose your application.

The first chapter in *Android Programming Succinctly* provided a brief introduction to XML layouts, but this chapter will take a much deeper look at Android's built-in layout options. In this chapter, we'll learn how to arrange UI elements in linear, relative, and grid layouts, the common navigation patterns recommended by Android maintainers, and we'll even take a brief look at how to develop device-independent layouts that can seamlessly stretch and shrink to different dimensions.

This chapter goes hand-in-hand with the upcoming chapter, in which we'll learn about all of the individual UI elements. The running example for this chapter uses buttons and text fields to demonstrate different layout functionality, but remember that you can substitute any of the UI elements discussed in the next chapter to the same effect.

Loading an Android Project

You can follow along with the examples for this chapter from a fresh Android project, but if you want to see the end result, you can load the UserInterfaceLayouts project contained in the sample code for this book. To open an existing project, launch **Eclipse** (with the ADT plugin), and select **Import** in the **File** menu. This will open a dialog that looks like the following figure:
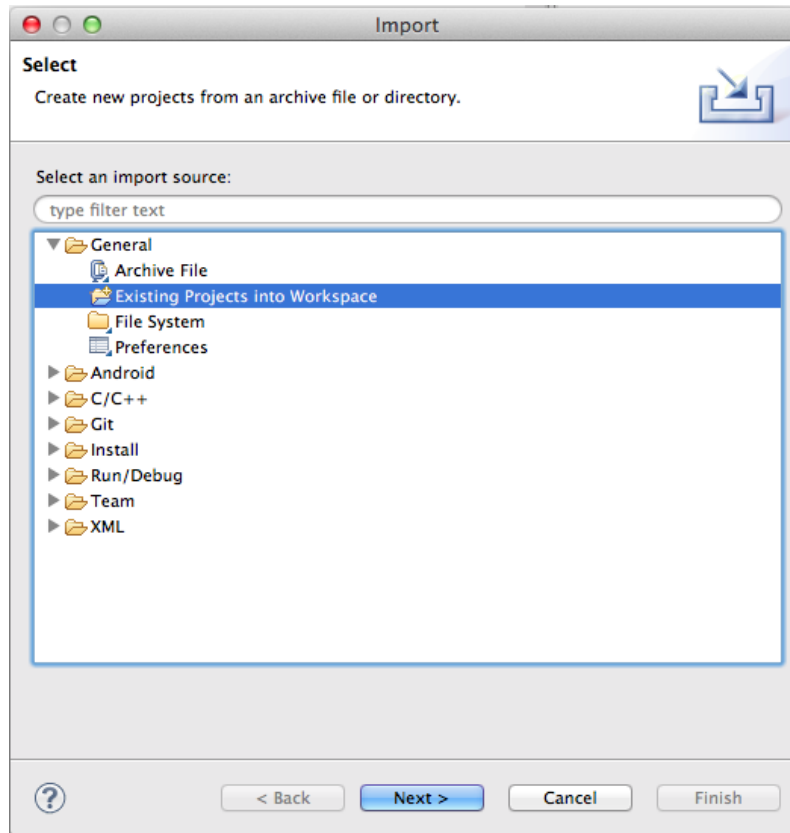
Figure 20: Opening a project in Eclipse

Open the **General** folder and select **Existing Projects into Workspace**, then click **Next.** In the next dialog, click **Browse...**, find the resource folder that came with this book, and open the **UserInterfaceLayouts** folder. After clicking **Finish**, you should see the project in the Package Explorer.

Loading Layouts

In the first chapter, we let the template load the XML layout file for us, but it's important to understand how this works under the hood. Fortunately, it's also a relatively simple process.

When you compile a project, Android automatically generates a `View` instance from each of your XML layout files. Like `String` resources, these are accessed via the special `R` class under the static `layout` variable. For example, if you wanted to access the `View` instance created from a file called `activity_main.xml`, you would use the following:

```
R.layout.activity_main
```

To display this `View` object in an `Activity`, all you have to do is call the `setContentView()` method. The `onCreate()` method of the blank `Activity` template always contains a call to `setContentView()` to load the associated view into the `Activity`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
}
```

Note that you can pass *any* View instance to setContentView(); using the generated
R.layout.activity_main object is just a useful convention for working with XML-based
layouts. If you need to replace an Activity's content dynamically, it's perfectly legal to call
setContentView() outside of onCreate().

Basic View Attributes

Before we start talking about Android's built-in layout schemes, it's important to understand how
to set the size and position of UI elements within a particular layout. The next three sections
show you how to define the dimensions, padding, and margin of a UI element using various
XML properties.

Size

To set the width of a particular UI element, all you need to do is add an
android:layout_width attribute to that element in the XML layout file. Likewise, the
android:layout_height attribute defines the height of the element. The value for either of
these parameters must be one of the following:

- wrap_content – This constant makes the element as big as it needs to be to contain its
  content.

- match_parent – This constant makes the element match the width and height of the
  parent element.

- An explicit dimension – Explicit dimensions can be measured in pixels (px), density-
  independent pixels (dp), scaled pixels based on preferred font size (sp), inches (in), or
  millimeters (mm). For example, android:layout_width="120dp" will make the element
  120 device-independent pixels wide.

- A reference to a resource – Dimension resources let you abstract reusable values into a
  resource file, just like we saw with strings.xml in the first chapter of this book.
  Dimension resources can be accessed using the @dimen/resource_id syntax, where
  resource_id is the unique ID of the resource defined in dimens.xml.

Let's start by exploring wrap_content. Change activity_main.xml to the following and
compile the project (we'll discuss the LinearLayout element later in this chapter):

```
<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"
    tools:context=".MainActivity"
    >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />

</LinearLayout>
```

You should see a single button at the top of the screen, and since we used `wrap_content` for both of the dimensions, it should be just big enough to fit the "Click me!" text (with some default padding). If you change the text, the button will expand or shrink to match. The following figure shows you the button you have created in the activity_main.xml.

Figure 21: A Button element using the wrap_content constant for both dimensions

If you change `android:layout_width` and `android:layout_height` to `match_parent`, the button will fill the entire screen, since that's how big the parent `LinearLayout` is:

```
<Button
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="Click me!" />
```

When you run this, it should look something like the following:
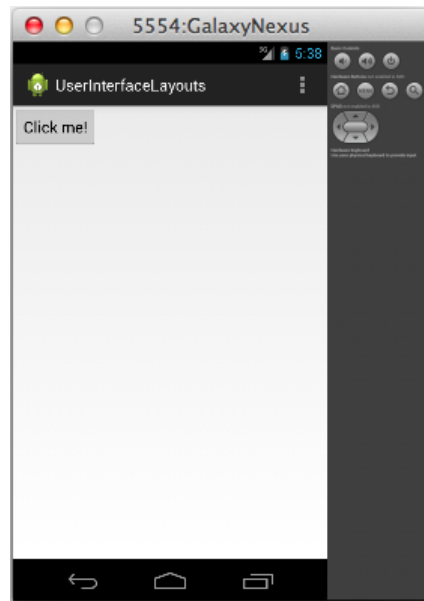
Figure 22: A Button element using the match_parent constant for both dimensions

If you need more control over the size of your UI elements, you can always manually define their width and height using explicit values, like so:

```xml
<Button
    android:layout_width="200dp"
    android:layout_height="60dp"
    android:text="Click me!" />
```

This makes the button 200 device-independent pixels wide by 60 device-independent pixels high, as you can see in the following figure:
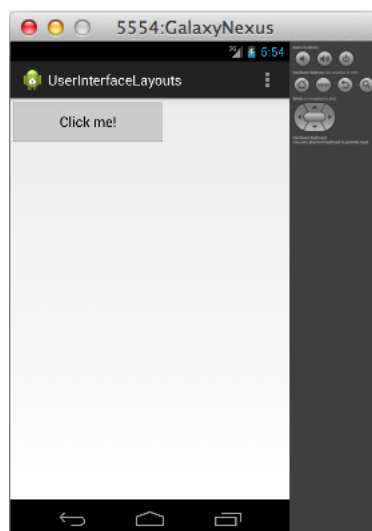
Figure 23: A Button element with an explicit width and height

The last option is to add an explicit dimension to `dimens.xml` and reference that from `activity_main.xml`. This is very useful if you have many elements that need to share the same dimension. Dimension resource references look just like string resource references, except you use `@dimen` instead of `@string`:

```xml
<Button
    android:layout_width="@dimen/button_width"
    android:layout_height="@dimen/button_height"
    android:text="Click me!" />
```

Of course, you'll also have to add these resources to `dimens.xml` before you compile the project:

```xml
<resources>

    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
    <dimen name="button_width">200dp</dimen>
    <dimen name="button_height">60dp</dimen>

</resources>
```

This will have the exact same effect as the previous snippet, but now it's possible to reuse the `button_width` and `button_height` values in other layouts (or for other elements in the same layout).

It's also worth noting that you can mix-and-match different methods for the width and height values. For example, it's perfectly legal to use `200dp` for the width and `wrap_content` for the height.

Padding

Padding is the space between an element's content and its border. It can be defined via any of the following attributes, all of which take an explicit dimension (e.g., `120dp`) or a reference to a resource (e.g., `@dimen/button_padding`):

- `android:padding` – Sets a uniform value for all sides of the element.

- `android:paddingTop` – Sets the padding for the top edge of the element.

- `android:paddingBottom` – Sets the padding for the bottom edge of the element.

- `android:paddingLeft` – Sets the padding for the left edge of the element.

- `android:paddingRight` – Sets the padding for the right edge of the element.

- `android:paddingStart` – Sets the padding for the start edge of the element.

- `android:paddingEnd` – Sets the padding for the end edge of the element.

Figure 24: Padding attributes

Padding can be added to `View` or `ViewGroup` elements. For the former, it defines the space between the element's contents (e.g., a button's title text) and its border, and for the latter it defines the space between the edge of the group and all of its child elements. For example, the following button will have 60 device-independent pixels surrounding its title text:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="60dp"
    android:text="Click me!" />
```

This should result in the following:
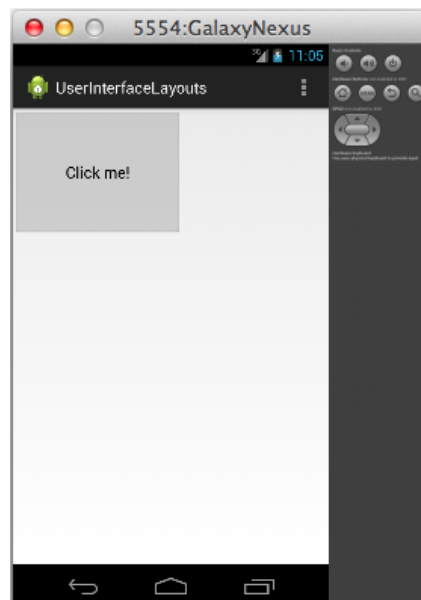
Figure 25: A Button with 60dp padding

Next, let's try adding some padding to the top and bottom of the containing `ViewGroup` (i.e., the `LinearLayout` element) and making the button match the size of its parent, like so:

```
<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"
    android:paddingTop="24dp"
    android:paddingBottom="24dp"
    tools:context=".MainActivity"
    >

    <Button
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Click me!" />

</LinearLayout>
```

This demonstrates that using `match_parent` on a child element takes the padding of the parent into account. Notice the `24dp` padding on the top and bottom of the following screenshot:
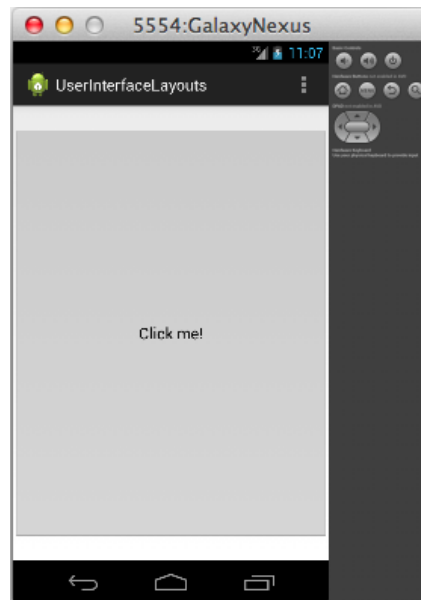
Figure 26: A Button with match_parent for its width and height and 24dp top and bottom padding in its parent ViewGroup

Margin

An element's margin is the space between its border and either the surrounding elements or the edges of the parent element. It can be specified for any `View` or `ViewGroup` using the `android:layout_margin` element. And, like `android:padding`, the top, bottom, left, right, start, and end margins can be defined individually using `android:layout_marginTop`, `android:layout_marginBottom`, etc.
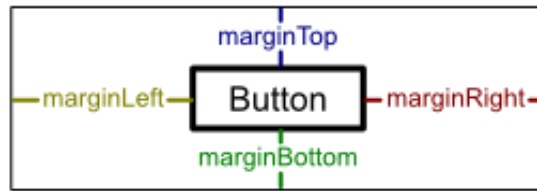
Figure 27: Margin attributes

For instance, the following code creates the same result as the example from the previous section by defining a top and bottom margin on the `Button` instead of top and bottom padding on the parent `LinearLayout`:

```xml
<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"
    tools:context=".MainActivity"
    >

    <Button
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginTop="24dp"
        android:layout_marginBottom="24dp"
        android:text="Click me!" />

</LinearLayout>
```

Common Layouts

Android provides three main ways to arrange UI elements: linear layouts, relative layouts, and grid layouts. The first two methods use classes called `LinearLayout` and `RelativeLayout`, respectively. These are subclasses of `ViewGroup` that have built-in functionality for setting the position and size of their child `View` objects. Grid layouts make it easy to display one- and two-dimensional data structures to the user, and they are slightly more complex than linear or relative layouts. The following figure shows you the available layout elements:
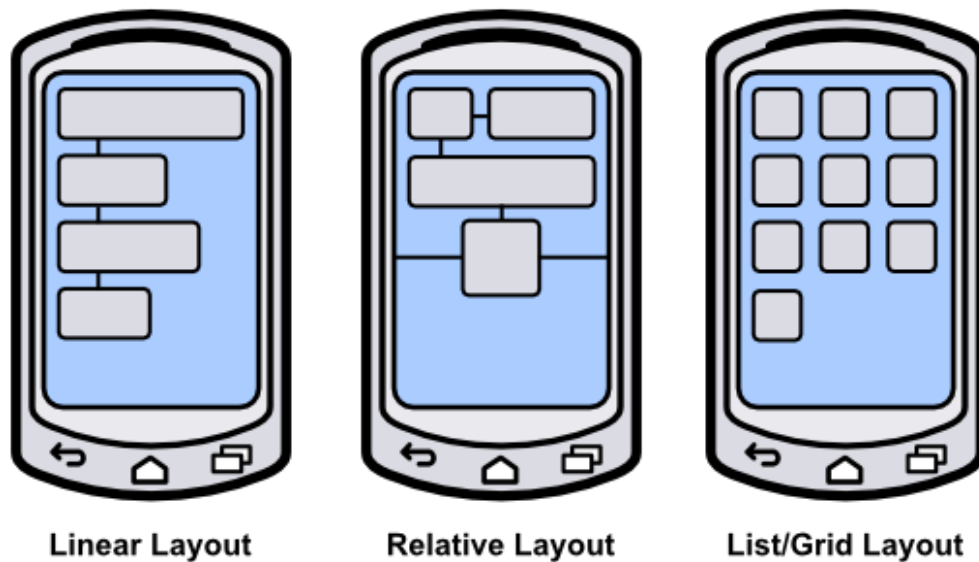
Figure 28: The three standard Android layouts

Linear Layouts

The `LinearLayout` class arranges all of the contained UI elements either horizontally or vertically. It displays each element in the same order it appears in the XML file, making it a very simple way to create complex layouts made up of many elements.

Orientation

A `LinearLayout` element needs to know in which direction it should lay out its children. This is specified via the `android:orientation` attribute, which can have a value of either `horizontal` or `vertical`. A horizontal layout will have each child element stacked left-to-right (in the same order as they appear in the source XML). For example, try changing `activity_main.xml` to the following:

```xml
<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="horizontal"
    tools:context=".MainActivity"
    >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />
```

```
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click me!" />

</LinearLayout>
```

If you compile the project and run it in the emulator, you should see a single row of three buttons, like the following:

Figure 29: A LinearLayout with horizontal orientation

But if you change the orientation to vertical, like so:

```
<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"
    tools:context=".MainActivity"
    >
...
</LinearLayout>
```

The three buttons will appear in a single column:

Figure 30: A LinearLayout with vertical orientation

Weight

You can use the same `android:layout_width` and `android:layout_height` attributes discussed earlier this chapter to define the size of each child element, but `LinearLayout` also enables another sizing option called `android:layout_weight`. An element's weight determines how much space it takes up relative to its siblings. For example, if you want three buttons to be evenly distributed over the height of the parent `LinearLayout`, you could use the following:

```xml
<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"
    tools:context=".MainActivity"
    >

    <Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="Click me!" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="Click me!" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="Click me!" />
```

```
</LinearLayout>
```

Note that for vertical orientations, you need to set each element's `android:layout_height` attribute to `0` so that it can be calculated automatically using the specified weight. For horizontal orientations, you would need to set their `android:layout_width` to `0`. Running the above code should give you the following:
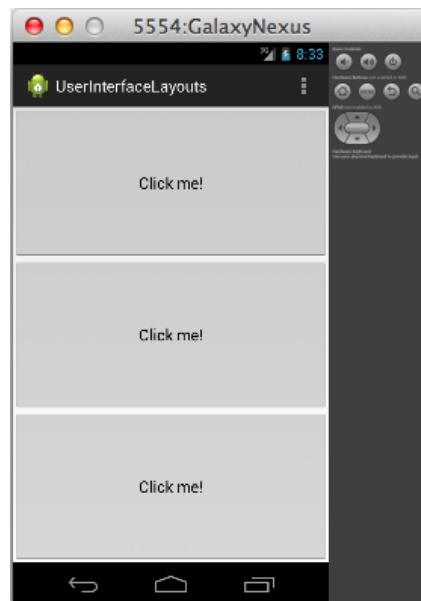
Figure 31: Evenly weighted UI elements in a vertical linear layout

You can change the weight ratios to make the buttons take up different proportions of the screen. For example, changing the button weights to 2, 1, 1 will make the first button take up half the screen and the other two a fourth of the screen:
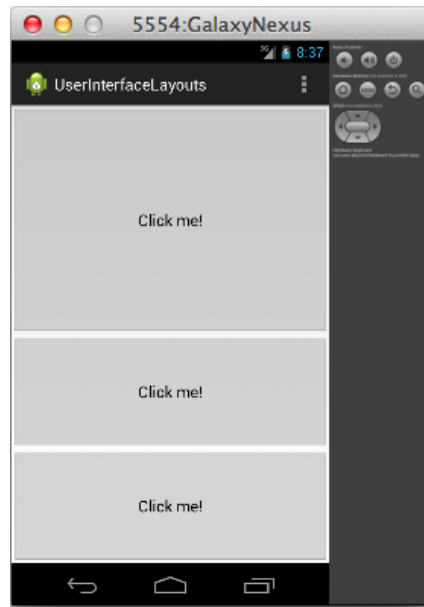
Figure 32: Unevenly weighted UI elements in a vertical linear layout

Specifying weights instead of explicit dimensions for UI elements is a flexible way to configure layouts, as it lets them automatically scale to match the size of the parent `ViewGroup`.

Relative Layouts

`RelativeLayout` is an alternative to `LinearLayout` that lets you specify where each element should be placed with respect to the other elements in the interface. Unlike `LinearLayout`, the order of the elements in the rendered interface of a `RelativeLayout` does not necessarily have to match the underlying XML file. Instead, their positions are defined by specifying a relationship to another element (e.g., "place the button to the left of the text field" or "place the button at the bottom of its parent"). The fact that positions are defined relative to other elements makes this a very powerful way to create pleasing UIs that expand and contract based on the screen size.

Relative To Parent

In a `RelativeLayout`, a UI element can be positioned with respect to its parent or with respect to its siblings. In either case, you define the position using one of the layout attributes defined by [RelativeLayouts.LayoutParams](). The attributes that position elements relative to their parent are listed below, and all of them take a Boolean value:

- `android:layout_alignParentLeft` – If `true`, aligns the left side of the element with the left side of its parent.

- `android:layout_centerHorizontal` – If `true`, centers the element horizontally in its parent.

- android:layout_alignParentRight – If true, aligns the right side of the element with the right side of its parent.

- android:layout_alignParentTop – If true, aligns the top of the element to the top of its parent.

- android:layout_centerVertical – If true, centers the element vertically in its parent.

- android:layout_alignParentBottom – If true, aligns the bottom of the element to the bottom of its parent.

- android:layout_alignParentStart – If true, aligns the start edge of the element with the start edge of its parent.

- android:layout_alignParentEnd – If true, aligns the end edge of the element with the end edge of its parent.

- android:layout_centerInParent – If true, centers the element horizontally and vertically in its parent.

For example, try changing activity_main.xml to the following:

```
<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=".MainActivity"
    >

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentBottom="true"
        android:text="Top Button" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="Middle Button" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="Bottom Button" />

</RelativeLayout>
```

This will result in a diagonal row of three buttons, as shown below:
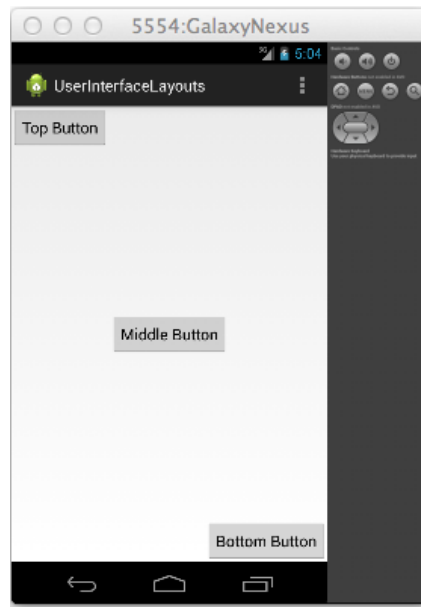
Figure 33: A RelativeLayout that positions elements with respect to their parent

Note that if you change the order of the XML elements in `activity_main.xml`, the buttons will still appear in the same locations. This is different than the behavior of `LinearLayout`, which places each element relative to the previous one. Also notice that if you change the button dimensions or rotate the emulator (Ctrl+F12), they will still appear in their respective locations relative to the screen.

Relative To Siblings

Elements can also be positioned relative to each other. The attributes listed below all specify a graphical relationship with surrounding elements, but instead of a Boolean value, they require an ID of another element in the layout:

- `android:layout_above` – Positions the bottom edge of the element above the element with the specified ID.

- `android:layout_below` – Positions the top edge of the element below the element with the specified ID.

- `android:layout_toLeftOf` – Positions the right edge of the element to the left of the element with the specified ID.

- `android:layout_toRightOf` – Positions the left edge of the element to the right of the element with the specified ID.

- `android:layout_alignBaseline` – Aligns the baseline of the element with the baseline of the element with the specified ID.

- `android:layout_alignTop`– Aligns the top edge of the element with the top of the element with the specified ID.

- android:layout_alignBottom – Aligns the bottom edge of the element with the bottom edge of the element with the specified ID.

- android:layout_alignLeft – Aligns the left edge of the element with the left edge of the element with the specified ID.

- android:layout_alignRight – Aligns the right edge of the element with the right edge of the element with the specified ID.

For example, consider the following `RelativeLayout` that uses relative-to-sibling attributes instead of relative-to-parent attributes. The second and third buttons are positioned relative to the first one, which will be located in the default top-left corner.

```
<Button
    android:id="@+id/topButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Top Button" />
<Button
    android:id="@+id/middleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/topButton"
    android:layout_toRightOf="@id/topButton"
    android:text="Middle Button" />
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/middleButton"
    android:layout_toRightOf="@id/middleButton"
    android:text="Bottom Button" />
```

Instead of a diagonal line of buttons that spans the entire screen, this code will give you a diagonal with all of the button corners touching:

Figure 34: A RelativeLayout that positions elements with respect to their siblings

Since we're referencing them by ID, we needed to include an `android:id` attribute for the top and middle buttons. Remember from the first chapter that the first time an XML element ID is used, it needs to be declared as `"@+id/foo"`. This plus sign usually occurs in the `android:id` attribute, but it doesn't have to—it should always be found in the *first* attribute that uses the ID. In the following snippet, the `android:layout_toLeftOf` attributes are the first place the `middleButton` and `bottomButton` IDs are referenced, so they need to be prefixed by a plus sign:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@+id/middleButton"
    android:layout_above="@id/middleButton"
    android:text="Top Button" />
<Button
    android:id="@id/middleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@+id/bottomButton"
    android:layout_above="@id/bottomButton"
    android:text="Middle Button" />
<Button
    android:id="@id/bottomButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentRight="true"
    android:layout_alignParentBottom="true"
    android:text="Bottom Button" />
```

This use of plus signs helps reduce the number of mistyped IDs by ensuring that new IDs are explicitly labeled as such. For example, if you were to accidentally try to reference an element with `@id/bottumButton`, the compiler would let you know that there is no such element.

The above XML positions the top and middle buttons relative to the bottom one, then puts the bottom one in the bottom-right corner of the screen. This gives you the following layout:
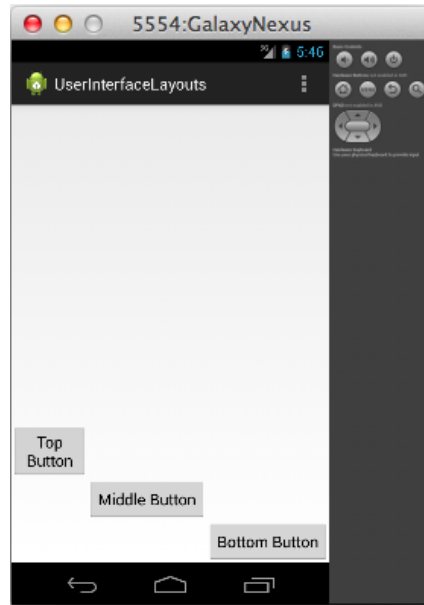
Figure 35: A RelativeLayout that positions buttons relative to elements that have not been declared yet

Also notice that you can combine the relative-to-sibling with the relative-to-parent positioning methods. The bottom button is positioned relative to its parent (e.g., `android:layout_alignParentRight`), and the others are positioned relative to a sibling (e.g., `android:layout_toLeftOf`).

List and Grid Layouts

So far, we've learned how to quickly create user interfaces with `LinearLayout` and `RelativeLayout`; however, the content of these interfaces have been entirely static—their UI elements are hardcoded into the underlying XML file. When you want to create a user interface using dynamic data, things get a little bit more complicated. Data-driven layouts require three components:

- A data set
- A subclass of [Adapter](#) for converting the data items into `View` objects.
- A subclass of [AdapterView](#) for laying out the `View` objects created by the `Adapter`

Before you can start configuring a data-driven layout, you need some data to work with. For this section, our data set will be a simple array of `String` objects, but Android also provides many built-in classes for fetching data from text files, databases, or web services.

Next, you need an `Adapter` to convert the data into `View` objects. For example, the built-in [ArrayAdapter](#) takes an array of objects, creates a `TextView` for each one, and sets its text property to the `toString()` value of each object in the array. These `TextView` instances are then sent to an `AdapterView`.

`AdapterView` is a `ViewGroup` subclass that replaces the `LinearLayout` and `RelativeLayout` classes from the previous sections. Its job is to arrange the `View` objects provided by the `Adapter`. This section explores the two most common `AdapterView` subclasses, [ListView](#) and [GridView](#), which position these views into a list or a grid, respectively.
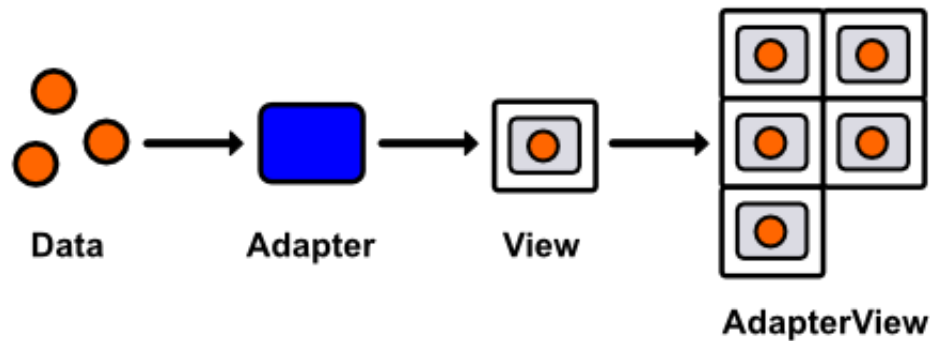
Figure 36: Displaying a data set in a GridView using an Adapter

List Layouts

Let's start by getting a simple `ListView` up and running. Since we're no longer hardcoding UI elements, our XML layout file is going to be very simple—all it needs is an empty `ListView` element. Change `activity_main.xml` to the following:

```xml
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/listView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

This `ListView` defines our entire XML layout—all of its UI elements will be populated dynamically. We do, however, need to include an `android:id` attribute so that we can access the `ListView` instance from our activity.

Next, we need to update our `MainActivity` class to define a data set, pass that data set to an `Adapter`, and then pass that adapter to the `ListView` that we defined in `activity_main.xml`. So, change `MainActivity.java` to the following:

```java
package com.example.userinterfacelayouts;

import android.os.Bundle;
import android.app.Activity;
import android.widget.ListView;
import android.widget.ArrayAdapter;

public class MainActivity extends Activity {

    @Override
```

```java
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        String[] data = new String[] { "Item 1", "Item 2", "Item 3",
                                       "Item 4" };

        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, data);
        ListView listView = (ListView) findViewById(R.id.listView);
        listView.setAdapter(adapter);
    }
}
```

First, we create an array of `Strings` to serve as our data set and assign it to the local `data` variable. Then, we create an `ArrayAdapter`, which generates a `TextView` from each `String` in the array. Its constructor takes an `Activity` context, the ID of the prototypical `TextView`, and the array of data. The `android.R.layout.simple_list_item_1` snippet is a reference to one of Android's convenient built-in layouts. You can find the complete list in the R.layout documentation. Next, we have to find the `ListView` that we added to `activity_main.xml` via `findViewById()`, and then we need to set its adapter property to the `ArrayAdapter` instance that we just configured.

You should now be able to compile the project and see the four strings in the `data` array displayed as a list of `TextView` elements:

Figure 37: A dynamic layout generated by ListView

List layouts make it incredibly easy to work with large data sets, and the fact that you can represent each item with *any* view lets you display objects that have several properties (e.g., a list of contacts that all display an image, name, and preferred phone number).

Grid Layouts

Grid layouts use the same data/Adapter/AdapterView pattern as list layouts, but instead of ListView, you use the [GridView](#) class. GridView also defines some extra configuration options for defining the number of columns and the spacing between each grid item, most of which are included in the following snippet. Try changing activity_main.xml to:

```xml
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridView"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:columnWidth="100dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="5dp"
    android:horizontalSpacing="5dp"
    android:stretchMode="columnWidth" />
```

The only change we need to make in MainActivity.java is to update the ListView references to GridView:

```java
package com.example.userinterfacelayouts;

import android.os.Bundle;
import android.app.Activity;
import android.widget.GridView;
import android.widget.ArrayAdapter;

public class MainActivity extends Activity {
    @Override
      protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_main);

         String[] data = new String[] { "Item 1", "Item 2", "Item 3",
                                         "Item 4", "Item 5", "Item 6",
                                         "Item 7", "Item 8", "Item 9"};

         ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, data);
         GridView gridView = (GridView) findViewById(R.id.gridView);
         gridView.setAdapter(adapter);
      }
}
```

This will give you a nice grid of text fields that are 100 device-independent pixels wide with 5 device-independent pixels between each one:
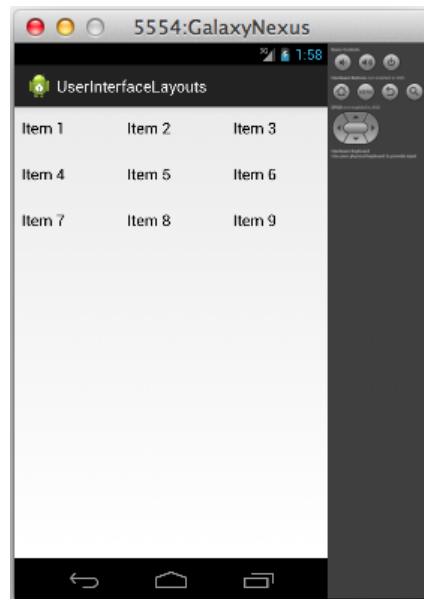
Figure 38: A dynamic layout generated by GridView

Handling Click Events

Of course, you're probably going to want to allow the user to interact with the items in a ListView or a GridView. However, because the interface generated by either of these classes is dynamic, we can't use the android:onClick XML attribute to call a method when the user clicks one of the list items. Instead, we have to define a general callback function in MainActivity.java. This can be accomplished by implementing the AdapterView.OnItemClickListener interface, like so:

```java
package com.example.userinterfacelayouts;

import android.os.Bundle;
import android.app.Activity;
import android.widget.GridView;
import android.widget.ArrayAdapter;

import android.util.Log;
import android.view.View;
import android.widget.TextView;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;

public class MainActivity extends Activity {

    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```java
            String[] data = new String[] { "Item 1", "Item 2", "Item 3",
                                           "Item 4", "Item 5", "Item 6",
                                           "Item 7", "Item 8", "Item 9"};

        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, data);
        GridView gridView = (GridView) findViewById(R.id.gridView);
        gridView.setAdapter(adapter);

        gridView.setOnItemClickListener(new OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent, View v,
                                    int position, long id) {
                TextView selectedView = (TextView) v;
                Log.d(TAG, String.format("You clicked: %s",
                                         selectedView.getText()));
            }
        });
    }
}
```

Now, the `onItemClick()` method will be called every time one of the `GridView`'s items are clicked. All of the relevant parameters are passed to this function as parameters: the parent `AdapterView`, the `View` item that was clicked, its position in the data set, and its `row` id. The above callback simply casts the clicked `View` to a `TextView` and displays whatever text it contains in LogCat.

Clicks can be handled in the exact same way for `ListView` layouts.

Editing The Data Set

When you want to change the data that is displayed to the user at runtime, all you have to do is edit the underlying data set and the built-in [BaseAdapter](#) class takes care of updating the user interface accordingly. In this section, we'll add a button to the layout so the user can add new items to the grid, and then we'll re-implement the `onItemClick()` function to remove the selected item from the list.

First, let's change `activity_main.xml` to include a button. We'll do this by making `LinearLayout` the root XML element and giving it a `Button` and a `GridView` for children:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    tools:context=".MainActivity" >

    <Button
        android:layout_width="match_parent"
        android:layout_height="80dp"
        android:text="Add Item"
        android:onClick="addItem"/>

    <GridView android:id="@+id/gridView"
```

```
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:columnWidth="100dp"
      android:numColumns="auto_fit"
      android:verticalSpacing="5dp"
      android:horizontalSpacing="5dp"
      android:stretchMode="columnWidth" />
 </LinearLayout>
```

Notice that it's perfectly legal to nest different layout schemes inside of each other (i.e., putting a GridView inside of a LinearLayout). This makes it possible to create complex, dynamic user interfaces with very little effort.

Next, we need to alter the corresponding activity to handle Add Item button clicks and remove items when they are selected in the GridView. This will require a number of changes to MainActivity.java:

```java
package com.example.userinterfacelayouts;

import android.os.Bundle;
import android.app.Activity;
import android.widget.GridView;
import android.widget.ArrayAdapter;

import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import java.util.ArrayList;

public class MainActivity extends Activity {

      private ArrayList<String> data;
      private ArrayAdapter<String> adapter;
      private int count;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.activity_main);

          this.data = new ArrayList<String>();
          this.data.add("Item 1");
          this.data.add("Item 2");
          this.data.add("Item 3");
          this.count = 3;

          adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, data);
          GridView gridView = (GridView) findViewById(R.id.gridView);
          gridView.setAdapter(adapter);

          gridView.setOnItemClickListener(new OnItemClickListener() {
              public void onItemClick(AdapterView<?> parent, View v,
                                      int position, long id) {
```

```
                data.remove(position);
                adapter.notifyDataSetChanged();
            }
        });
    }

    public void addItem(View view) {
        count++;
        String newItem = String.format("Item %d", count);
        this.data.add(newItem);
        this.adapter.notifyDataSetChanged();
    }
}
```

First, we have to change the static `String[]` array that represents our data set to a mutable `ArrayList`. This will allow us to add and remove items. We also have to change the `data` and `adapter` local variables to instance variables so that we can access them outside of `onCreate()`. We also added a `count` variable to keep track of how many items have been created.

To remove an item from the the `GridView`, all we have to do is remove it from the data set with `ArrayList`'s `remove()` method, then call `adapter.notifyDataSetChanged()`. This latter method is defined by `BaseAdapter`, and it tells the `Adapter` that it needs to synchronize its associated `AdapterView` items. It should be called whenever the underlying data set has changed.

The new `addItem()` method is called whenever the Add Item button is clicked. First it increments the `count` variable, then uses it to generate a title for the new item, adds the title to the data set, and finally calls `notifyDataSetChanged()` to refresh the `GridView`.
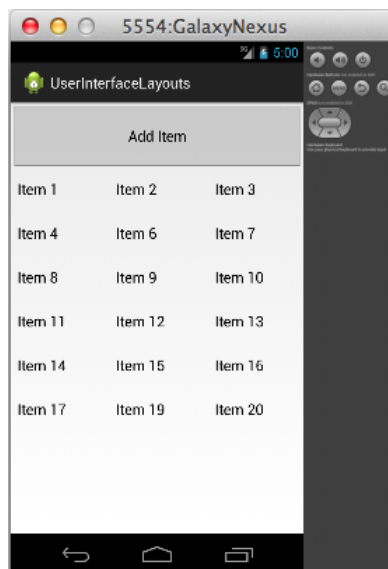
Figure 39: Adding and removing items from the data set

You should now be able to compile the project and alter the underlying data set. While this example only demonstrates adding and removing items from the `GridView`, editing existing items works the exact same way. All you have to do is change the value in the `ArrayList` and call `notifyDataSetChanged()` on the adapter.

Custom ListView (and GridView) Items

The `R.layout.simple_list_item_1 TextView` that we've been using up to this point is convenient when working with string data, but more complex data items typically call for correspondingly complex views. Fortunately, it's relatively easy to implement your own `ListView` or `GridView` items. In this section, we'll create a custom view that displays a name and a phone number.

Custom `ListView` items require three components: a class to represent the data, an XML layout file to define the `View` for each item, and a custom `Adapter` to display the data in the view. These components will be replacing the `String` data, the `android.R.layout.simple_list_item_1` view, and the `ArrayAdapter` from the previous example. The result we're aiming for looks like this:
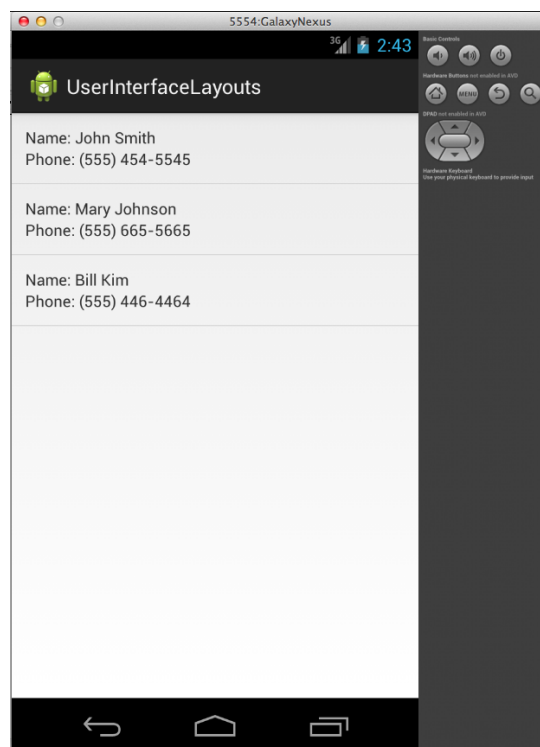
Figure 40: ListView with custom View objects for each item

First, we need to create a new class to represent a custom data item. We'll call it `DataItem`, and all it needs to store is two properties called `name` and `phoneNumber`. You can create a new class in Eclipse by pressing **Cmd+N** (or Ctrl+N if you are on a PC), and then selecting **Java**, and then **Class**. Enter **DataItem** for the **Name** field, and leave everything else at default values. The form should look like the following:
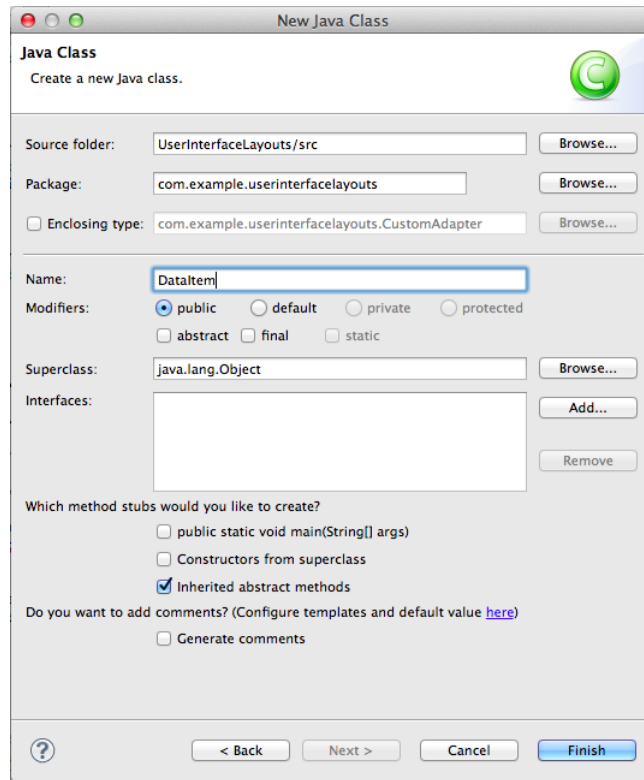
Figure 41: Creating the DataItem class

After clicking **Finish**, you should find a new file called **DataItem.java** under `src/com.example.userinterfacelayouts` in the Package Explorer. Double-click the file to open it, and change it to the following:

```java
package com.example.userinterfacelayouts;

public class DataItem {

    String name;
    String phoneNumber;

    public DataItem(String name, String phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

}
```
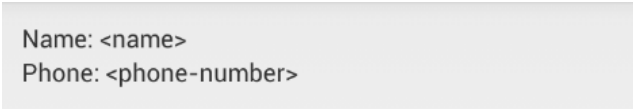
This defines two properties for each data item, along with a convenient constructor. That's all we need to represent the data, so we can move on to the XML layout file that defines the View object associated with each data item.

To create the layout XML file, press **Cmd+N** (or Ctrl+N if you are on a PC) and select **Android**, and then **Android XML Layout File**. Use `list_item.xml` for the **File** field, and leave **Root Element** as the default value (we'll change it by editing the XML). Clicking **Finish** will give you a new file called `list_item.xml` in the `res/layout` folder. We want each item to look like the following:

```
Name: <name>
Phone: <phone-number>
```

Figure 42: The view created by list_item.xml

We'll use a `RelativeLayout` and four `TextView` elements to create this layout. Change `list_item.xml` to the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="12dp">

    <TextView
        android:id="@+id/nameLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/nameLabel"/>

    <TextView
        android:id="@+id/nameValue"
        android:layout_toRightOf="@id/nameLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <TextView
        android:id="@+id/phoneLabel"
        android:layout_below="@id/nameLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/phoneLabel"/>

    <TextView
        android:id="@+id/phoneValue"
        android:layout_below="@id/nameLabel"
        android:layout_toRightOf="@id/phoneLabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</RelativeLayout>
```

The `id/nameValue` and `id/phoneValue` elements are going to be set dynamically by our custom Adapter, so they don't need an `android:text` attribute, but the `id/nameLabel` and `id/phoneLabel` elements are static labels, so they can be populated from `strings.xml`. Add the following two lines to `strings.xml`:

```xml
<string name="nameLabel">Name: </string>
<string name="phoneLabel">Phone: </string>
```

The only new part here is the ` ` entity, which is a non-breaking space. This is necessary to get our label to display correctly. That takes care of the XML layout file, so all we have left is the custom Adapter to connect it with the `DataItem` class.

Create another new class called `CustomAdapter` and use `BaseAdapter` as the subclass. BaseAdapter is a minimal implementation of the `Adapter` class. It provides some basic definitions for the inner-workings of `Adapter`, which lets us focus on higher-level functionality in our `CustomAdapter`. This is going to be a longer class definition, so let's tackle it in steps.

Let's start by changing `CustomAdapter.java` to the following:

```java
package com.example.userinterfacelayouts;

import java.util.ArrayList;

import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.content.Context;
import android.widget.BaseAdapter;
import android.widget.TextView;

public class CustomAdapter extends BaseAdapter {

    ArrayList<DataItem> data;
    Context context;
    private static LayoutInflater inflater = null;

    public CustomAdapter(Context context, ArrayList<DataItem> data) {
        this.context = context;
        this.data = data;
        inflater = (LayoutInflater) context
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    }

}
```

This starts by importing the classes that we'll need later on. Then, it defines a few properties. The `data` variable stores the data backing the adapter, which is just an `ArrayList` of `DataItem` objects. The Context class contains information about the global application environment, and we need to store a reference to it in the `context` variable. Finally, we need a `LayoutInflater` to turn the XML from `list_item.xml` into a `View` hierarchy. Without this `LayoutInflater`, it would be impossible to get to the text fields that we defined in `list_item.xml`.

Next, we define a constructor that takes a `Context` instance and the data set as parameters. Then, it uses the context to fetch a `LayoutInflater`. The [getSystemService()](#) method of a `Context` instance is the canonical way to do this.

Now we're ready to define the custom behavior of our adapter. Add the following methods to `CustomAdapter.java`:

```java
    @Override
    public int getCount() {
        return data.size();
    }

    @Override
    public Object getItem(int position) {
        return data.get(position);
    }

    @Override
    public long getItemId(int position) {
        return position;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
         // See if the view needs to be inflated
        View view = convertView;
        if (view == null) {
            view = inflater.inflate(R.layout.list_item, null);
        }
        // Extract the desired views
        TextView nameText = (TextView) view.findViewById(R.id.nameValue);
        TextView phoneText = (TextView) view.findViewById(R.id.phoneValue);

        // Get the data item
        DataItem item = data.get(position);

        // Display the data item's properties
        nameText.setText(item.name);
        phoneText.setText(item.phoneNumber);

        return view;
    }
```

The first three methods provide information that is required by all `Adapter` subclasses. The `getCount()` and `getItem()` methods must return the number of items represented by the adapter and the item at the specified position, respectively. These both just need to forward the information from the underlying `ArrayAdapter`. The `getItemId()` method should return the row ID of the item at the specified position. In this case, we can just return the position of the item in the array.

The heart of our `CustomAdapter` class is the `getView()` method, which must return a `View` object that represents the data item at the specified position. This is where the `list_view.xml` file is converted into a `View` and its `TextViews` are populated with data from the associated `DataItem`.

First, we need to see if the adapter is working with an existing view, which would be passed in via the `convertView` parameter. If it's not, we need to create a new `View` instance by **inflating** the XML layout file with the `LayoutInflater` instance that we recorded in the constructor. This parses the XML, turning each element into its corresponding view object and adding it to a view hierarchy. Finally, we need to find the `TextView` elements that we defined in the XML file and use them to display the requested `DataItem`'s `name` and `phoneNumber` properties. The following figure shows you how to inflate an XML layout file to access the contained views.
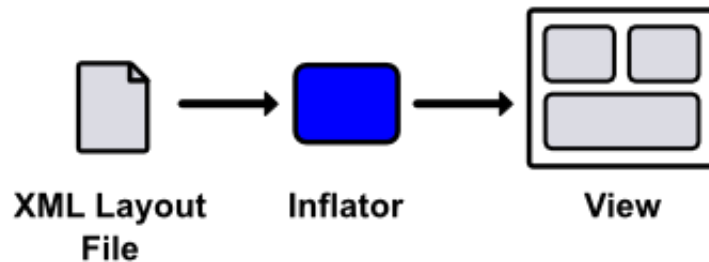
Figure 43: Inflating an XML layout file to access the contained views

This process should provide a better understanding of the purpose of an `Adapter`: its `getView()` method is where the data set is *adapted* to be displayed in a `View` hierarchy. The returned `View` instance is what gets displayed by the parent `ListView`/`GridView`.

We have our three components of a customized `ListView` set up, but we still need to put all of it together in the main activity. All that's required for `activity_main.xml` is a `ListView` that we can reference in `MainActivity.java`:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal"
    tools:context=".MainActivity" >

    <ListView
        android:id="@+id/listView"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

</LinearLayout>
```

Then, in `MainActivity.java`, we need to create the data set of `DataItem` instances and attach our `CustomAdapter` to the above `ListView`:

```java
package com.example.userinterfacelayouts;

import android.os.Bundle;
```

```java
import android.app.Activity;
import android.widget.ListView;
import java.util.ArrayList;

public class MainActivity extends Activity {

    private ArrayList<DataItem> data;
    private CustomAdapter adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        this.data = new ArrayList<DataItem>();
        this.data.add(new DataItem("John Smith", "(555) 454-5545"));
        this.data.add(new DataItem("Mary Johnson", "(555) 665-5665"));
        this.data.add(new DataItem("Bill Kim", "(555) 446-4464"));

        adapter = new CustomAdapter(this, data);
        ListView listView = (ListView) findViewById(R.id.listView);
        listView.setAdapter(adapter);
    }
}
```

You should now be able to compile the app and see our custom `list_item.xml` displayed as each item in the `ListView`. While this was a simple example using only `TextView` widgets, it's easy to extend this pattern to create arbitrarily complex `ListView` or `GridView` items with images, buttons, and other UI widgets.

The concept to take away is the interaction between lists/grids, custom views, data objects, and adapters. Together, they provide a reusable model-view-controller framework that makes it possible to display complex data sets to the user with minimal effort on the part of the developer.

Nesting Layouts

While layout optimization is an advanced topic out of the scope of this book, it's worth noting that excessively deep view hierarchies can be a potential performance bottleneck. For this reason, you should try to keep your view groups flat and wide, as opposed to narrow and deep. Consider the following layout:
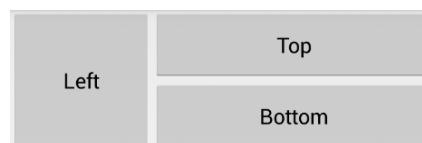

Figure 44: A simple layout consisting of three buttons

This can be created in one of two ways. First, let's see how to do it using two `LinearLayouts`: a horizontal one to separate the left button from the others, and another nested `LinearLayout` to render the top and bottom buttons.

```xml
<!-- Don't do this (it's not efficient) -->
<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="horizontal"
    tools:context=".MainActivity" >
    <Button
        android:layout_width="0dp"
        android:layout_height="120dp"
        android:layout_weight="1"
        android:text="Left" />
    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="120dp"
        android:layout_weight="2"
        android:orientation="vertical">
        <Button
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:text="Top" />
        <Button
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:text="Bottom" />
    </LinearLayout>
</LinearLayout>
```

However, the fact that both `LinearLayout` elements contain widgets with a `layout_weight` property means that their dimensions need to be calculated twice. Needless to say, this is not an optimal layout.

The other (and preferred) method of creating this layout is with a single `RelativeLayout`. This has the advantage of flattening the view hierarchy into a single layer, avoiding the inefficiencies of the nested `LinearLayouts` shown above.

```xml
<!-- Do this instead (it's more efficient) -->
<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=".MainActivity" >
    <Button
        android:id="@+id/leftButton"
        android:layout_width="120dp"
        android:layout_height="120dp"
        android:text="Left" />
    <Button
```

```
        android:id="@+id/topButton"
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:layout_toRightOf="@id/leftButton"
        android:text="Top" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:layout_toRightOf="@id/leftButton"
        android:layout_below="@id/topButton"
        android:text="Bottom" />
</RelativeLayout>
```

So, the general rule of thumb is to avoid nesting `LinearLayouts` whenever possible. This is particularly true for custom `ListView`/`GridView` items because the view gets inflated multiple times, thus any inefficiencies are multiplied as well. However, that's not to say you should *never* use `LinearLayouts`. They are very easy to configure, and they are plenty efficient when not nested. This makes them appropriate for simpler layouts (e.g., a form made up of consecutive text fields).

For more information about creating efficient layouts, please visit [Optimizing Your UI](#).

Summary

In this chapter, we explored the standard mechanisms of laying out an Android user interface. First, we discussed the basic attributes for defining the padding and margin of `View` elements. Then, we learned how to position elements using the built-in `ViewGroup` subclasses: `LinearLayout`, `RelativeLayout`, `ListView`, and `GridView`. These classes make it incredibly easy to organize UI widgets into complex layouts with minimal coding.

While it's important to know how to position elements on the screen, that knowledge is useless if you don't know *what* elements are available. The next chapter will flesh out your understanding of Android's user interface framework by surveying the most common UI widgets.

Chapter 5   User Interface Widgets

In this chapter, we'll survey Android's basic user interface widgets: buttons, text fields, checkboxes, sliders, spinners, and pickers. Understanding how to configure and query these components enables you to collect or display virtually any kind of information to a user, and that's the basis of any quality Android application. We'll also briefly talk about how to change the appearance of these UI elements and Android's built-in themes.

This chapter is meant to be a introduction to the many UI components offered by Android. You should walk away with a basic understanding of what components are available and how to get started using them in real-world applications. Links to official documentation providing more information are included throughout the chapter.

All of the examples in this chapter are available in the *UserInterfaceWidgets* project included with the example code of this book. The main activity of this project provides links to several other activities, each of which contain concrete examples of a particular type of widget.

Images

Images can be displayed using an [ImageView](#) object. An `ImageView` can display any kind of bitmap, and it takes care of basic alignment and scaling functionality. You can find several examples of various `ImageView` configurations in the `activity_image.xml` layout file:
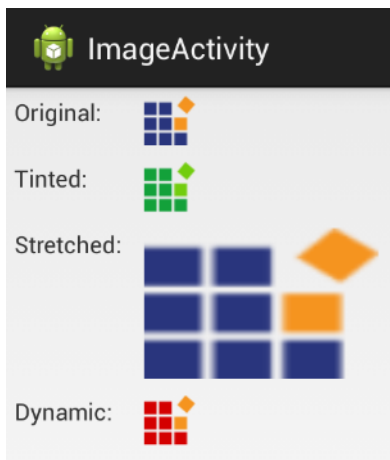

Figure 45: The ImageActivity defined in the example application for this chapter

To include an `ImageView` in an XML layout file, you can develop the following code:

```
<ImageView
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:padding="5dp"
    android:src="@drawable/syncfusion_icon" />
```

The most important attribute of an `ImageView` is `android:src`. This defines the image that it will display. In the above case, `@drawable/syncfusion_icon` refers to an image file that was included with the application source code. Note that this has a similar format to string resources (e.g., `@string/button_title`). However, the `@drawable` prefix refers to a **drawable resource**, which is any kind of graphical asset.

Adding Drawable Resources

Of course, for the above code snippet to work, we need to add an image file called `syncfusion_icon` to the Eclipse project. The *UserInterfaceWidgets* example project already contains this file, but adding your own graphical assets is as simple as copying them from your hard drive and pasting them into one or more of the `res/drawable` directories in the Eclipse Package Explorer. Remember that the `hdpi`, `ldpi`, and other suffixes correspond to different screen resolutions, so it's a good practice to include alternative images for each of the resolutions that you plan to support.
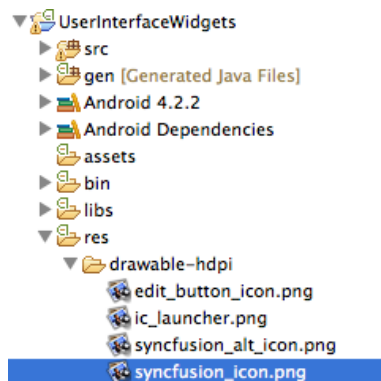
Figure 46: Adding a graphical asset to an Eclipse project

Note that drawable resources can be PNG, GIF, or JPG files—the file extension will be inferred automatically.

Scaling Images

You can scale images by defining `layout_width` and `layout_height` attributes on the `ImageView` element. `ImageView` provides several types of [scaling behaviors](). The value of an `ImageView`'s `android:scaleType` attribute determines whether it stretches the image to the specified dimensions (`fitXY`), scales proportionally to display the whole image (`centerInside`), centers the unscaled image within the specified dimensions (`center`), or maps the image dimensions to the `layout_width` and `layout_height` values in several other pre-defined ways.

```
<ImageView
    android:layout_width="150dp"
    android:layout_height="100dp"
    android:scaleType="fitXY"
    android:src="@drawable/syncfusion_icon" />
```

For example, the above code will stretch the `syncfusion_icon` image to 150x100 device-independent pixels.

Programmatically Defining the Image Source

If you want to set the source image of an `ImageView` instance dynamically, you can pass a drawable resource to its `setImageDrawable()` method. The following snippet, which can be found in `ImageActivity.java`, demonstrates how to load an image called `syncfusion_alt_icon.png` into an `ImageView` defined in an XML layout file.

```
// Dynamically load an image into an ImageView
ImageView imageView = (ImageView) findViewById(R.id.dynamicImage);
Resources resources = getResources();
Drawable image = resources.getDrawable(R.drawable.syncfusion_alt_icon);
imageView.setImageDrawable(image);
```

In the Android framework, the **resource bundle** contains all of the XML layout files, string resources, and drawable resources. An application's resource bundle is represented by the [Resources](#) class, and the active bundle can be fetched through the global `getResources()` function. The `Resources` class defines several useful methods that turn a resource ID into a useful object. In this case, the `getDrawable()` method lets us convert an image file in one of the `res/drawable` folders into a [Drawable](#) object, which can be displayed by an `ImageView` object. Note that the ID of the image file is accessed via `R.drawable`, much like an XML layout file is access via `R.layout`.

The `setImageDrawable()` method is a very flexible way to display images, as it's possible to load bitmaps from the resource bundle (as shown above) from a remote URL, or from a user-defined location.

Buttons

We've been working with `Button` views throughout this book, but there a few common modifications that are worth surveying. The buttons that we've used thus far have been purely text-based, but it's possible to create icon-based buttons, as well as ones that combine icons and text. The `activity_button.xml` layout file contained in the included example project contains text-based, icon-based, and combined buttons.

Figure 47: The ButtonActivity defined in the example application for this chapter

We've already seen several examples of text-based buttons, so we'll jump right into icon buttons. Icon buttons are created with a dedicated class called `ImageButton`. These kinds of buttons don't display a text title, and they use the same `android:src` attribute as the `ImageView` from the previous section. For example, the following snippet uses an image file called `edit_button_icon.png` as its icon:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:src="@drawable/edit_button_icon" />
```

As with `ImageView`, the `android:src` attribute must point to a drawable resource that you've added to your project.

Combining icons and text in a single button is a little bit more complicated. We have to go back to the familiar `Button` class and tell it where it should display the icon in relation to its text title using one of the following attributes:

- `android:drawableBottom` – Add the icon under the text

- `android:drawableEnd` – Add the icon after the text (varies based on text direction)

- `android:drawableLeft` – Add the icon to the left of the text

- `android:drawableRight` – Add the icon to the right of the text

- `android:drawableStart` – Add the icon before the text (varies based on text direction)

- `android:drawableTop` – Add the icon above the text

Each of these takes a drawable resource as a value, just like `ImageView`'s `android:src` attribute. For example, if you wanted to display the image file `edit_button_icon.png` to the left of the text title, you could use the following:

```
<Button
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:text="Edit"
    android:drawableLeft="@drawable/edit_button_icon"
    android:drawablePadding="10dp" />
```

Also notice the `android:drawablePadding` attribute, which lets you define the amount of space between the icon and the text.

All of these button types can use the `android:onClick` attribute to define a method to be called when they are tapped, but if you're creating them programmatically, this XML attribute is not available. Instead, you need to set the button's `onClickListener` property, just like we did to detect clicks of `ListView` and `GridView` items in the previous chapter. For instance, if you wanted to programmatically define the behavior of the `textImageButton` in the `ButtonActivity` of this chapter's example app, you would use the following:

```
Button button = (Button) findViewById(R.id.textImageButton);
button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View sender) {
        Button senderAsButton = (Button) sender;
        String title = senderAsButton.getText().toString();
        Log.d(TAG, String.format("You clicked the '%s' button", title));
    }
});
```

This method of defining button behavior opens up many more possibilities than the static `android:onClick` XML attribute. It's necessary when dynamically generating buttons, and it makes it possible to change a button's behavior depending on its context.

Text Fields

Static text fields (i.e., labels) were introduced earlier in this book. In this section, we'll learn how to change their color, size, and other properties. We'll also learn how to accept user input via editable text fields, which are one of the most basic ways to collect input from a user. Concrete examples of all the code in this section can be found in the `activity_text_field.xml` layout in the *UserInterfaceWidgets* project. A more detailed overview of text fields can be found in the Input Controls section of the developer guide, as well as in the TextView class documentation.

Styling Text Fields

Remember that text fields can be added to a layout using the `<TextView>` element and their text can be defined with the `android:text` attribute. The appearance of a particular text field can be altered by defining other attributes on it, the most common of which are listed below:

- android:textColor – The color of the text field, specified as a hex number in the form of #AARRGGBB.

- android:textSize – The size of the text. The scaled pixel (sp) is the preferred unit to use.

- android:textStyle – The style of the type. Must be either normal, bold, or italic.

- android:typeface – The typeface to use. Value must be either normal, sans, serif, or monospace.

- android:textIsSelectable – Whether or not the user can select the text. Must be either true or false.

The following <TextView> demonstrates all of these properties. Note that the scaled pixel unit (sp) is based on the user's preferred font size, which means that it will scale appropriately with the surrounding text. When explicitly defining the text size using pixel (px), device-independent pixels (dp), or inches (in), this will not be the case.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Look, a big red serif font"
    android:textColor="#ffff0000"
    android:textSize="20sp"
    android:textStyle="italic"
    android:typeface="serif"
    android:textIsSelectable="true" />
```

This will create a label that looks like this:

Figure 48: A TextView with customized appearance

And, since android:textIsSelectable is set to true, the user can tap and hold the text to select and copy it.
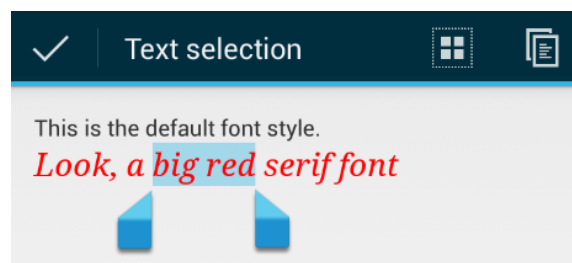
Figure 49: Selecting part of a selectable TextView

Editable Text Fields

Editable text fields have a distinct appearance and behavior from static ones. They use an underline and a hint to show the user that it is meant for collecting input, and when the user taps an editable text field, an on-screen keyboard—also called a soft keyboard (opposed to a *hard*ware keyboard)—appears.
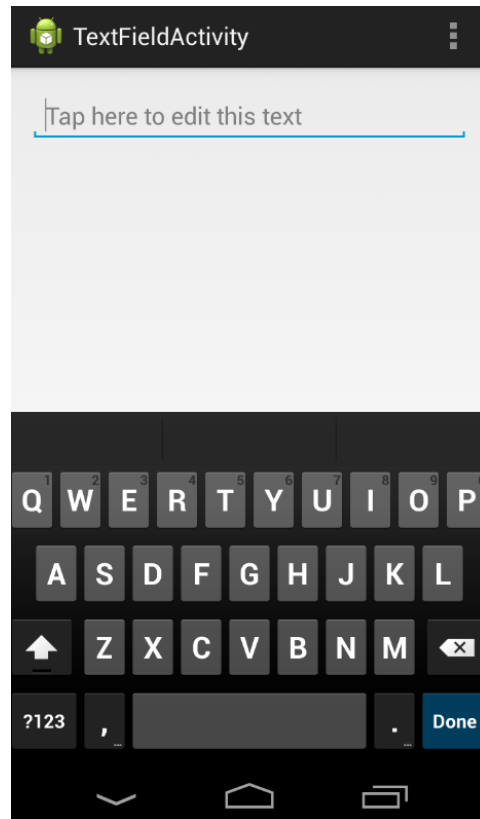
Figure 50: Editing a text field with a soft keyboard

Instead of the `<TextView>` element, editable text views are created with the `<EditText>` element. For example, the text view in the above screenshot was created from the following XML:

```
<EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="Tap here to edit this text"
    android:inputType="text" />
```

Editable text field hints are one of the most important attributes to set. They function as labels, telling the user what kind of input is expected.

The android:inputType attribute lets you specify what kind of input you're expecting. This can have a drastic impact on usability, since it determines what type of soft keyboard is displayed. For example, if you only wanted to collect a phone number, you would use phone for this value. Instead of a full keyboard, this will make Android display a dial pad, making it much easier to enter the desired input.

Figure 51: Using android:inputType to display a number pad instead of a full keyboard

There is a plethora of built-in input types, and many of them can be combined to give app developers refined control over the user experience. Some of the most common values for android:inputType values are listed below (see the [android:inputType documentation](#) for all of the available options):

- text – Use a normal text keyboard.

- textEmailAddress – Use a text keyboard with the @ character readily available.

- textUri – Use a text keyboard with the / character readily available.

- number – Use a number keypad without traditional dial pad letters.

- phone – Use a number keypad with traditional dial pad letters (e.g., the *2* key also displays *ABC*).

- textCapWords – Capitalize each word that the user types.

- textCapSentences – Capitalize the first letter of each sentence.

- TextAutoCorrect – Auto correct misspelled words using Android's built-in dictionary.

- textPassword – Hide characters after they have been typed.

- datetime – Use a number keypad with a / character readily available.

Some of these values can be combined using a bitwise operator (|). For example, if you wanted an <EditText> element to use a text input, capitalize sentences, and auto correct misspelled words, you would use the following:

```
android:inputType="text|textCapSentences|textAutoCorrect"
```

Another way to customize the soft keyboard is with the `android:imeOptions` attribute. This defines what is used as the Done button. For example, if you wanted to display *Send* as the final action after the user is finished entering input, you would add the following line to the the `<EditText>` element:

```
android:imeOptions="actionSend"
```

The resulting keyboard is shown in the following screenshot. Notice how the Done button from the previous examples turned into a Send button.

Figure 52: Changing the android:imeOptions attribute to show a Send button

The most common values for `android:imeOptions` are: `actionDone`, `actionSend`, `actionSearch`, and `actionNext`, all of which are self-explanatory. Please visit the android:imeOptions documentation for the complete list.

Collecting Text Input

Of course, if you're using an `<EditText>` element, you're probably going to want to do something with the input after the user is done entering it. This process is similar to listening for button clicks with an `OnClickListener`. In `TextFieldActivity.java`, you'll find a minimal example that shows you how to collect the input.

Let's start by looking at the classes that need to be imported for our example to work:

```
import android.view.KeyEvent;
import android.view.inputmethod.EditorInfo;
import android.widget.TextView;
import android.widget.TextView.OnEditorActionListener;
import android.widget.EditText;
import android.util.Log;
```

The KeyEvent class contains information about which key was pressed. We won't be using this for our example, but it can be very useful when working with hardware keyboards. The EditorInfo class defines several constants that let us check which kind of text the `<EditText>` element is collecting. Of course, we'll need the `TextView` and `EditText` classes, along with the OnEditorActionLister class, which is what lets us listen for the "done" action.

To figure out when the user has pressed the Done button, you first need to find the text view in question, then set its `onEditorActionListener` property, like so:

```java
EditText text = (EditText) findViewById(R.id.textField);
text.setOnEditorActionListener(new OnEditorActionListener() {
    public boolean onEditorAction(TextView textView,
                                   int actionId,
                                   KeyEvent event) {
        if (actionId == EditorInfo.IME_ACTION_SEND) {
            String input = textView.getText().toString();
            Log.d(TAG, String.format("Processing input: %s", input));
        }
        return false;
    }
});
```

The `onEditorAction()` function is called whenever the user has finished editing the specified `<EditText>` (in this case, the one with an ID of `textField`). This is where you should process the input however you see fit. If you want to check which kind of action was sent, you can check its `actionId` parameter against one of the constants in `EditorInfo`. Here, we made sure that it was a *Send* action, then we simply logged the input to LogCat.

The return value of `onEditorAction()` is very important. If it returns `true`, it means that your code has taken care of everything related to collecting the input, including hiding the on-screen keyboard, if necessary. However, if it returns `false`, it means that the default handling behavior should be executed, which will typically hide the on-screen keyboard.

The above snippet returns `false` so that the keyboard is hidden when the user is done, regardless of what the user entered. If you want to manually dismiss the keyboard based on the input (e.g., you want to keep the keyboard displayed if the user entered an invalid value), you can do so with the [InputMethodManager](#), like so:

```java
InputMethodManager imm =
(InputMethodManager)getSystemService(Context.INPUT_METHOD_SERVICE);
imm.hideSoftInputFromWindow(textView.getWindowToken(), 0);
```

You'll also need two more imports for this to work:

```java
import android.content.Context;
import android.view.inputmethod.InputMethodManager;
```

You should now understand how to include editable text fields in XML layouts, configure their keyboards, and collect that data after the user is done entering it. This is everything you need to know to start collecting text input in your own Android applications.

Checkboxes

Checkboxes are designed to let users select multiple boolean items at the same time. They can be contrasted with radio buttons, which only let the user select one item from the group. Since checkboxes can be checked individually, working with them is very similar to working with buttons.

They are represented by the [CheckBox](#) class, which implements the actual check and box, as well as a corresponding text label. Checkboxes use the same `android:onClick` attribute as buttons, which makes it easy to determine when they are toggled by the user.
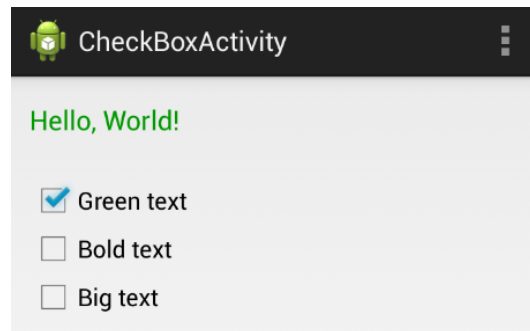
Figure 53: Using several checkboxes to change the appearance of a TextView

The `activity_check_box.xml` layout file and the `CheckBoxActivity.java` class in the example project demonstrate the most common characteristics of checkboxes (along with some programmatic `TextView` manipulation). It uses three checkboxes to alter the color, style, and size of a text field. The complete example can be seen in the above screenshot. Let's start by taking a look at the XML for one of the checkboxes:

```xml
<CheckBox
    android:id="@+id/checkBoxGreen"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Green text"
    android:onClick="checkBoxClicked" />
```

This creates a checkbox with *Green text* as its label, and it calls the `checkBoxClicked()` method on `CheckBoxActivity` whenever it's clicked. The example project contains another one for toggling whether the text field is normal or bold (*Bold text*), and a third to switch between `18sp` and `30sp` text size (*Big text*).

The activity to handle this click will be a little bit more involved than previous examples. In `CheckBoxActivity.java`, you'll find three private instance variables used to store the state of the text field:

```java
private boolean isGreen;
private boolean isBold;
```

```
private boolean isBig;
```

The onCreate() method initializes these variables, then calls two methods that we'll define shortly:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_check_box);
    this.isGreen = true;
    this.isBold = false;
    this.isBig = false;
    synchronizeCheckBoxes();
    synchronizeTextView();
}
```

These synchronization methods separate the logic for making the checkboxes and the TextView match the internal state of the activity, respectively. For activities that have many interconnected UI widgets, organizing their interaction in this way can make for a much more maintainable project. The synchronizeCheckBoxes() method fetches each of the checkboxes by their ID and makes them match their corresponding property using CheckBox's setChecked() method:

```
private void synchronizeCheckBoxes() {
    CheckBox green = (CheckBox) findViewById(R.id.checkBoxGreen);
    CheckBox bold = (CheckBox) findViewById(R.id.checkBoxBold);
    CheckBox big = (CheckBox) findViewById(R.id.checkBoxBig);
    green.setChecked(this.isGreen);
    bold.setChecked(this.isBold);
    big.setChecked(this.isBig);
}
```

The synchronizeTextView() method uses the private instance variables to toggle the appearance of the TextView:

```
private void synchronizeTextView() {
    TextView text = (TextView) findViewById(R.id.checkBoxText);
    if (this.isGreen) {
        text.setTextColor(Color.parseColor("#FF009900"));
    } else {
        text.setTextColor(Color.parseColor("#FF000000"));
    }
    if (this.isBold) {
        text.setTypeface(Typeface.create("default", Typeface.BOLD));
    } else {
        text.setTypeface(Typeface.create("default", Typeface.NORMAL));
    }
    if (this.isBig) {
        text.setTextSize(TypedValue.COMPLEX_UNIT_SP, 30);
    } else {
        text.setTextSize(TypedValue.COMPLEX_UNIT_SP, 18);
    }
```

```
    }
```

Finally, we have the checkboxes' on-click method, which uses the ID and value of the clicked checkbox to update the internal state:

```java
public void checkBoxClicked(View view) {
    CheckBox checkbox = (CheckBox) view;
    boolean isChecked = checkbox.isChecked();
    switch (view.getId()) {
        case R.id.checkBoxGreen:
            this.isGreen = isChecked;
            break;
        case R.id.checkBoxBold:
            this.isBold = isChecked;
            break;
        case R.id.checkBoxBig:
            this.isBig = isChecked;
            break;
    }
    synchronizeTextView();
}
```

These three methods provide a clear data flow that is very easy to maintain, even for larger activities: the synchronization methods make the views match the internal state of the activity, and checkBoxClicked() collects user input to alter that state.

Radio Buttons

From a UI perspective, a group of radio buttons is like a group of checkboxes; however, only one item is allowed to be selected at a time. From a developer perspective, this behavior makes their API distinct from checkboxes. Instead of managing each item individually, you have to encapsulate the radio buttons in a group so the system knows that it should only select one of them at a time.

The buttons are represented by the RadioButton class, and you group them using the RadioGroup class. Like buttons and checkboxes, you can use the android:onClick attribute on the buttons to call a method when the user makes a selection.

The `activity_radio_button.xml` layout and the `RadioButtonActivity.java` class in this chapter's example project provide a simple demonstration of radio buttons and radio groups. It uses three radio buttons to let the user choose the typeface of a `TextView`. Whereas the previous section's example let the user toggle several independent properties, the `sans serif`, `serif`, and `monospace` typeface values are mutually exclusive, so radio buttons are an appropriate choice for presenting these options.

The XML for a radio is a list of `<RadioButton>` elements surrounded by a `<RadioGroup>` element. Aside from ensuring that only one item is selection, the job of the `RadioGroup` is to arrange the radio buttons in a horizontal or vertical format. It's actually a subclass of `LinearLayout`, so you can use the same `android:orientation` attribute to set the direction of the radio buttons. Each of the contained `<RadioButton>` elements are essentially the same as the buttons and checkboxes that we've been working with:

```xml
<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    <RadioButton
        android:id="@+id/radioButtonSans"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Sans Serif"
        android:onClick="radioButtonClicked"/>
    <RadioButton
        android:id="@+id/radioButtonSerif"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Serif"
        android:onClick="radioButtonClicked"/>
    <RadioButton
        android:id="@+id/radioButtonMonospace"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Monospace"
        android:onClick="radioButtonClicked"/>
</RadioGroup>
```

In `RadioButtonActivity.java`, you'll find some code in the `onCreate()` method to set the initial selection, along with a `radioButtonClicked()` method to update the `TextView` whenever the user changes the selection. We didn't bother abstracting as much of the functionality as we did in the previous section, since there is only one property being altered:

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_radio_button);

    // Set the initial selection
    RadioButton serif = (RadioButton) findViewById(R.id.radioButtonSerif);
```

```java
        serif.setChecked(true);
        radioButtonClicked(null);
    }

    public void radioButtonClicked(View view) {
        // Use the radio group to find the selected button
        RadioGroup group = (RadioGroup) findViewById(R.id.radioGroup);
        String typeface;
        switch (group.getCheckedRadioButtonId()) {
            case R.id.radioButtonSans:
                typeface = "sans";
                break;
            case R.id.radioButtonSerif:
                typeface = "serif";
                break;
            case R.id.radioButtonMonospace:
                typeface = "monospace";
                break;
            default:
                typeface = "default";
        }
        // Update the TextView accordingly
        TextView text = (TextView) findViewById(R.id.radioButtonText);
        text.setTypeface(Typeface.create(typeface, Typeface.NORMAL));
    }
```

The only new method here is the RadioGroup's getCheckedRadioButtonId(), which returns the ID of the selected radio button. This lets you figure out which item is selected without querying each button (notice how radioButtonClicked() doesn't need to use the view parameter at all). Radio buttons can be selected programmatically using the same setChecked() method as checkboxes, which we use to set the initial selection. If you do need to individually inspect each radio button, you can use the corresponding getChecked() method.

It's also worth noting that you can clear the radio button selection with RadioGroup's clearCheck() method.

Spinners

Spinners are drop-down menus that allow the user to pick one item from a group of choices. They provide similar functionality to radio buttons, but they take up less space on the screen and make it easier to see the selected item. For these reasons, it is advised to use a spinner instead of radio buttons if you're offering more than four or five options for a single field.

While they may offer similar functionality as radio buttons, spinners require an entirely different API. Their items are populated using an Adapter, which makes working with them more like working with list views and grid views rather than radio buttons or checkboxes.
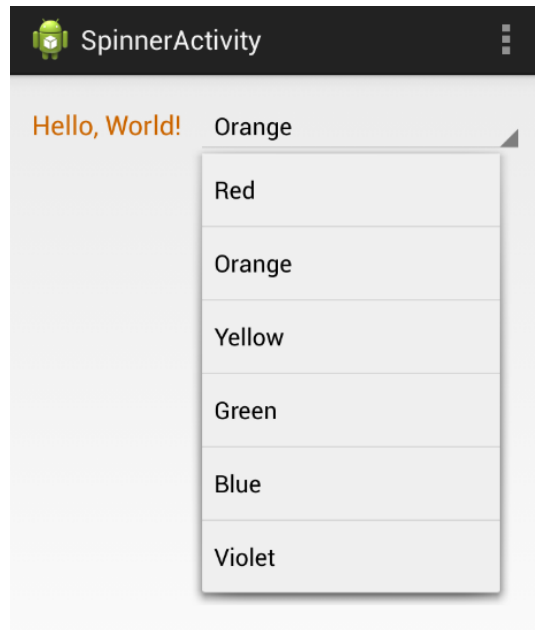
Figure 55: Using a Spinner to change the color of a TextView

In this section, we'll learn how to create a [Spinner](), populate it with an [ArrayAdapter](), and handle user input with an [OnItemSelectedListener](). This is almost the exact pattern we used to configure list views and grid views earlier in this book. The `activity_spinner.xml` layout file and the `SpinnerActivity.java` class demonstrate how to set a `TextView`'s color with a `Spinner`.

Since spinners need to be populated programmatically, the XML for adding one to a layout is very simple:

```xml
<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="horizontal"
    tools:context=".SpinnerActivity" >

    <TextView
        android:id="@+id/spinnerText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="10dp"
        android:text="Hello, World!"
        android:textSize="18sp" />

    <Spinner
        android:id="@+id/colorSpinner"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

SpinnerActivity.java shows you how to populate this spinner. First, we create an ArrayList to represent the options:

```
ArrayList<String> colors = new ArrayList<String>();
colors.add("Red");
colors.add("Orange");
colors.add("Yellow");
colors.add("Green");
colors.add("Blue");
colors.add("Violet");
```

Then, we use this ArrayList to create an ArrayAdapter. Remember from our work with list views that an ArrayAdapter is what converts the data items into View objects for display by the spinner. Android provides a built-in spinner item resource, accessible via android.R.simple_spinner_item. But, since spinners are dropdown widgets, we also need to set the adapter's dropDownViewResource property. The built-in android.R.simple_spinner_dropdown_item is the preferred resource to use for this:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    android.R.layout.simple_spinner_item,
    colors);
adapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item);
```

Then, we can give the adapter to the Spinner that we defined in the layout file:

```
Spinner spinner = (Spinner) findViewById(R.id.colorSpinner);
spinner.setAdapter(adapter);
```

If you compile the project at this point, you should be able to see the selected item in the spinner and be able to tap it to see the dropdown menu. But, to make it actually *do* something, we need to implement a selection handler. It's important not to confuse the OnItemSelectedListener class that we need for the spinner with the OnItemClickListener that we used with the ListView and GridView in the previous chapter. The former has an additional method that we need to implement (though it doesn't necessarily need to do anything):

```
spinner.setOnItemSelectedListener(new OnItemSelectedListener() {
    public void onItemSelected(AdapterView<?> parent,
                               View v,
                               int position,
                               long id) {
        String selectedColor = (String) parent.getItemAtPosition(position);
        setTextColor(selectedColor);
    }

    public void onNothingSelected(AdapterView<?> parent) {
        // Called when the selection disappears
    }
});
```

The `onItemSelected()` method is where the click is handled. The above code fetches the selected `String` using the adapter's `getItemAtPosition()` method. We then pass it off to a method called `setTextColor()`, which looks like this:

```java
private void setTextColor(String color) {
    String hexColor = "#FF000000";
    if (color.equals("Red")) {
        hexColor = "#FFAA0000";
    } else if (color.equals("Orange")) {
        hexColor = "#FFCC6600";
    } else if (color.equals("Yellow")) {
        hexColor = "#FFCCAA00";
    } else if (color.equals("Green")) {
        hexColor = "#FF00AA00";
    } else if (color.equals("Blue")) {
        hexColor = "#FF0000AA";
    } else if (color.equals("Violet")) {
        hexColor = "#FF6600AA";
    }

    TextView text = (TextView) findViewById(R.id.spinnerText);
    text.setTextColor(Color.parseColor(hexColor));
}
```

This takes the `String` values displayed in the spinner, turns them into hex values, and updates the `TextView`'s color accordingly. This is all that's required to get our spinner working.

If you don't like defining the list items programmatically, it's possible to put them in an XML resource file and load it into an adapter dynamically. This is a better practice than hardcoding values in activity classes, as it keeps all of your text values in XML files. Since it's easy to load different resource files based on the device and the user's locale, this makes it a breeze to translate your app into other languages

First we need to define a string array in `strings.xml`:

```xml
<string-array name="spinnerColors">
    <item>Red</item>
    <item>Orange</item>
    <item>Yellow</item>
    <item>Green</item>
    <item>Blue</item>
    <item>Violet</item>
</string-array>
```

To load these items into the spinner, all you have to do is replace the current `ArrayAdapter` with one created from the resource file:

```java
ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
    R.array.spinnerColors,
    android.R.layout.simple_spinner_item);
```

`R.array.spinnerColors` is the ID of the string array we just created, and the static `ArrayAdapter.createFromResource()` method takes care of everything else for us. This will have the exact same effect as hardcoding the `ArrayList`.

Date/Time Pickers

Android provides built-in UI components for selecting dates and times. Typically, this is done via a dialog instead of directly in an activity. The DatePickerDialog and TimePickerDialog classes provide reusable interfaces that ensure a valid date/time is selected by the user. They also ensure a consistent user interface across applications.
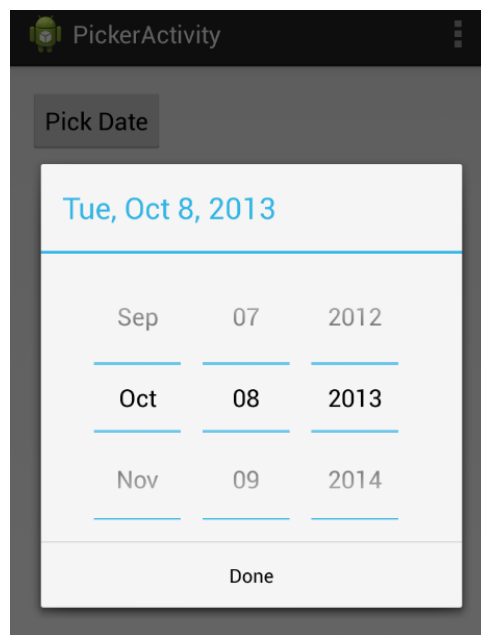
Figure 56: Selecting the date with a date picker dialog

This section explains the bare minimum required to collect date or time input from a user. Creating a date picker requires three things:

1. A DatePickerDialog object defines the appearance of the dialog.
2. A DialogFragment object hosts the `DatePickerDialog` and manages the dialog lifecycle.
3. An OnDateSetListener implementation processes the user input.

The first component is provided by the Android platform, so all we need to do is instantiate it.

The DialogFragment is a lightweight wrapper for the actual dialog, and it makes sure that the dialog is opened/closed properly, and that any interruptions are handled correctly. All we need to do is subclass DialogFragment to return a DatePickerDialog object as its hosted dialog. Fragments are introduced in the next chapter, but for now, suffice it to say that they are modular UI components. You can think of them as reusable views that can be embedded in different activities or dialogs.

To collect the input, we need to implement the OnDateSetListener interface, which defines a single method called onDateSet() that gets called whenever the user closes the dialog. Since you'll probably want to process the input in the host Activity, this is where we'll define onDateSet().

First, let's start with the XML for the example, which is just a button that lets the user pick a date:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="10dp"
    android:text="Pick Date"
    android:onClick="showDatePickerDialog" />
```

The showDatePickerDialog() method is what will open the dialog, but before we get to that, let's configure the dialog itself. This is conventionally done by subclassing DialogFragment and overriding its onCreateDialog() method to return whatever dialog you want to display. So, we need to create a new class called DatePickerFragment, and it will look like this:

```
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.app.Dialog;
import android.app.DatePickerDialog;
import java.util.Calendar;

public class DatePickerFragment extends DialogFragment {

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
            // Create and return the date picker dialog
        final Calendar c = Calendar.getInstance();
        int year = c.get(Calendar.YEAR);
        int month = c.get(Calendar.MONTH);
        int day = c.get(Calendar.DAY_OF_MONTH);

        PickerActivity context = (PickerActivity) getActivity();
        return new DatePickerDialog(context, context, year, month, day);
    }

}
```

First, we use the [Calendar](#) class to fetch the current date, then we extract the year, month, and day components to give to the date picker. Then, all we need to do is instantiate `DatePickerDialog`. The first parameter should be the host `Activity`, and the second one is the listener object, which should implement the `OnDateSetListener` interface. Since we want the host activity to be the listener, too, we use this as the second parameter (note that this will require a change to the `PickerActivity` class declaration).

Now, we can define the `showDatePickerDialog()` method in `PickerActivity.java` to display this dialog. Opening the picker entails creating the `DialogFragment` object that hosts the date picker (`DatePickerFragment`), then calling its `show()` method, like so:

```java
public void showDatePickerDialog(View view) {
    DialogFragment picker = new DatePickerFragment();
    picker.show(getSupportFragmentManager(), "datePicker");
}
```

The `getSupportFragmentManager()` method is a backwards-compatible way of displaying fragments (fragments were added in Android 3.0, but can support back to Android 1.6 using `getSupportFragmentManager()`). This method is defined in [FragmentActivity](#), which means `PickerActivity` must subclass that instead of the usual `Activity`. Remember that `PickerActivity` is also being used as the listener object, so its class declaration should look like this:

```java
public class PickerActivity extends FragmentActivity
                            implements DatePickerDialog.OnDateSetListener
```

Finally, to process the selected date, we need to define `onDateSet()` in `PickerActivity.java`. In this case, we'll just display it in a text field:

```java
@Override
public void onDateSet(DatePicker view, int year, int month, int day) {
    // Process the selected date (month is zero-indexed)
    TextView text = (TextView) findViewById(R.id.pickerText);
    String message = String.format("Selected date: %d/%d/%d",
                                   month+1, day, year);
    text.setText(message);
}
```

Note that the `month` parameter is always zero-indexed to be compatible with the `Calendar` class.

Summary

This chapter briefly introduced some of the most important widgets for creating an Android user interface. We learned how to use image views, buttons, text fields, checkboxes, radio buttons, spinners, and date pickers to display and collect information from the user. There are a few other useful UI components that we didn't discuss, including [toggle buttons](#) and [action bars](#), but we'll leave these for you to explore on your own.

By now, you should have the skills to create your own multi-screen Android apps and construct several different types of layouts. In the next section, we'll look at a more advanced aspect of user interface development called fragments. Fragments are part of a modular framework for reusing behaviors and views in several different activities. This will open up several new navigation possibilities for your Android apps.

Chapter 6   Fragments

A **fragment** is an encapsulated portion of an activity. Fragments let you configure UI components and behaviors into a reusable entity, then embed that entity in several different activities. This opens up all sorts of layout and navigation opportunities for Android applications. You can think of them as "fragments" of an activity than can be combined in different ways to form a complete activity. The following figure shows you the different fragments you can use, while developing an Android application.

Master and Detail fragments
in separate activities
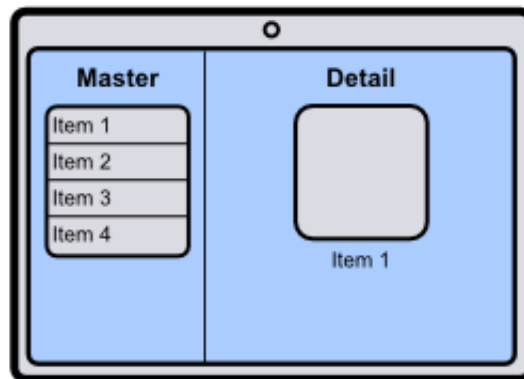
The same Master and Detail
fragments in a single activity

Figure 57: Reusing master and detail fragments in smartphone and tablet layouts

One of the most common use cases for fragments is to implement the master-detail UI pattern. For example, an email app contains a list of messages (the master interface) and a separate place to view the message body (the detail interface). By using one fragment for the list of messages and another for the message body, you can easily support multiple screen dimensions by combining the fragments in different ways. On a tablet, you might display both fragments in a single activity to make the most of your screen real estate, while on a smartphone, you would probably want to display them in dedicated activities that both take up the whole screen. Since the master and detail fragments are reusable, all you have to do is combine them differently based on the screen dimensions—all of their UI components and behaviors are reusable.

It's also possible to seamlessly swap fragments into and out of an activity. This lets you display different kinds of interfaces and behaviors within a single activity, which enables two common navigation patterns: swipe views and tabbed navigation.
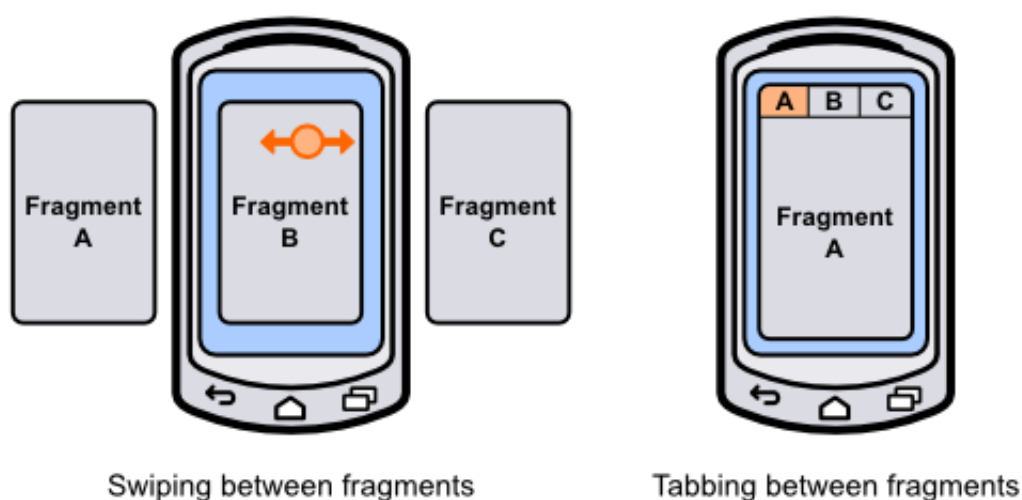
Figure 58: Swiping and tabbing between fragments

Swipe views are designed to let the user navigate between sibling detail items in a master-detail application. For example, in an email app, the user can swipe the screen to move forwards and backwards through their messages without being forced to navigate back to the master list. It's also possible to use the swipe gesture to navigate between tabs, which we'll explore later on in this chapter.

Tabbed navigation lets the user switch between fragments using labeled tabs at the top of the screen. Tabs are meant for switching between the top-level sections of your application, so there shouldn't be more than three or four tabs displayed at any given time. For example, an email app might have tabs for switching between the inbox, your starred messages, and your email settings.

This chapter provides a basic introduction to fragments. We'll learn how to create them, embed them in activities, and use them to implement swipe views and tabbed navigation. You should walk away with an understanding of how to encapsulate functionality into a fragment and reuse that functionality in various activities. The *Fragments* project included in the sample code for this book demonstrates everything that we're about to discuss.

Creating a Fragment

Fragments are made up of two parts: an XML layout file that defines what the fragment looks like, and a class to load that layout and define its behavior. Notice that these are the exact same components required for an activity.

The *Fragments* example project includes two fragments: a `HomeFragment` and an `ArticlesFragment`. Both of them display a text field with their name and a colored background so you can see their dimensions once we load them into an activity. The XML for each fragment looks exactly like that of an activity. For example, `home_view.xml` is defined as follows:

```xml
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#FFDDFFDD">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:textSize="32sp"
        android:textColor="#FF009900"
        android:text="Home Fragment" />

</RelativeLayout>
```

Next, we need to define a subclass of [Fragment](#) to load this layout. The `Fragment` class was only added in Android 3.0 (API 11). If you don't need to support anything below API 11, you can import the `Fragment` class from `android.app.Fragment`, but it's possible to support back to Android 1.6 by importing Fragment from the support library with `android.support.v4.app.Fragment`. The example code for this chapter uses the latter method.

The `HomeFragment.java` class defines the behavior of the home fragment. The `onCreateView()` is the only method that *must* be overridden in a `Fragment` subclass, as it returns the root `View` that represents the fragment. In this case, all we need to do is inflate the XML layout:

```java
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.ViewGroup;
import android.view.View;
```

```java
public class HomeFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.home_view, container, false);
    }
}
```

Fragments follow the same lifecycle pattern as activities. You can define custom `onCreate()`, `onPause()`, `onResume()`, and all of the other activity lifecycle callback methods that we discussed earlier in this book in a `Fragment` subclass. When one of the lifecycle methods is called on a host activity, it passes that along to any fragments it contains. This is part of what makes fragments so modular—they act just like activities, but are completely self-contained.

Embedding Fragments in Activities

Fragments *must* be be hosted by an activity. It's not possible to display fragments on their own, although it can be the only element in an activity. If you change `activity_main.xml` to the following, you'll see how to embed a fragment in an activity:

```xml
<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"
    tools:context=".MainActivity" >

    <fragment android:name="com.example.fragments.HomeFragment"
            android:id="@+id/homeFragment"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />

    <fragment android:name="com.example.fragments.ArticlesFragment"
            android:id="@+id/articlesFragment"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />

</LinearLayout>
```

This includes both the `HomeFragment` and the `ArticlesFragment` in the main activity, and the result is shown in the screenshot below. As you can see, fragments are embedded with the `<fragment>` tag. The most important attribute is `android:name`, which should be an absolute path to the class that defines the fragment (including the application package).

Remember that the goal of a fragment is to be a reusable piece of UI. This means that it should be able to stretch or shrink to match whatever size is defined by the host activity. This is why the root element of the fragment uses `match_parent` for its `android:layout_width` and `android:layout_height`. The above code makes each fragment half of the screen:
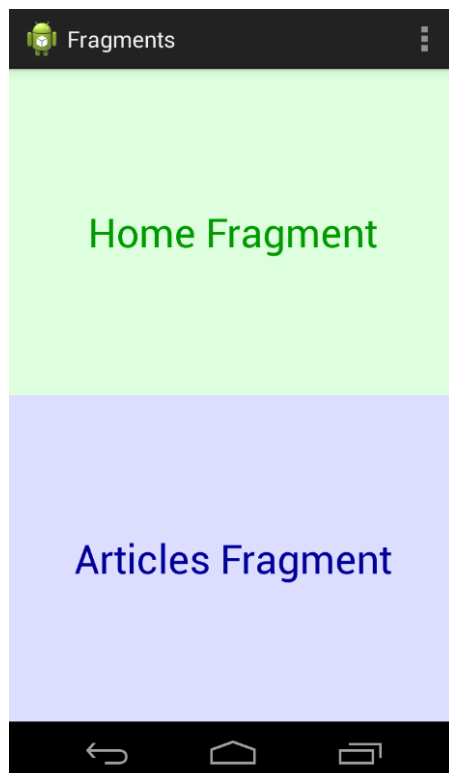
Figure 59: Displaying two fragments in a single activity

Right now, fragments might seem like just an abstraction for layout files (which they are), but they enable all sorts of other navigation options. Since they are reusable, we can display both of them in the same activity (like we've done here), display them in separate activities, paginate them with swipe views, or let the user access them via tabs.

Swipe Views

Fragments can be paginated using swipe views, which are implemented with the [ViewPager](#) class. Swipe views are recommended for only a small number of screens, and each of those screens should be self-contained. If you're letting the user swipe through a collection of data items, you should also provide a master list as an alternative means of navigation, and you shouldn't use swipe views to paginate long articles.

To add a swipe view to the main activity, replace the `<fragment>` elements in `activity_main.xml` with the following:

```
<android.support.v4.view.ViewPager
      android:id="@+id/fragmentPager"
      android:layout_width="match_parent"
      android:layout_height="match_parent" />
```

This adds a [ViewPager](#) that is supported back to Android 1.6. The `ViewPager` will be populated in `MainActivity.java` using an adapter, much like `ListViews` and `GridViews` are populated. First, it needs to define a subclass of [FragmentPagerAdapter](#), called `SimplePagerAdapter` in the following code. Its `getItem()` method returns the fragment associated with page. In this case, it returns `HomeFragment` for the first page and `ArticlesFragment` for the second one. Of course, it's possible to dynamically generate these pages from a data set if you're trying to swipe through a collection. To populate the `ViewPager`, all we need to do is set its `adapter` property to the `SimplePagerAdapter`.

```java
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
import android.support.v4.view.ViewPager;
import android.view.Menu;

public class MainActivity extends FragmentActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        SimplePagerAdapter adapter = new
SimplePagerAdapter(getSupportFragmentManager());
        ViewPager pager = (ViewPager) findViewById(R.id.fragmentPager);
        pager.setAdapter(adapter);
    }

    public static class SimplePagerAdapter extends FragmentPagerAdapter {
      public SimplePagerAdapter(FragmentManager fragmentManager) {
          super(fragmentManager);
      }

      @Override
      public int getCount() {
          return 2;
      }

      @Override
      public Fragment getItem(int position) {
          switch (position) {
          case 0:
              return new HomeFragment();
          case 1:
              return new ArticlesFragment();
          default:
              return null;
```

```
            }
        }
    }


      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
            // Inflate the menu; this adds items to the action bar if it is
present.
            getMenuInflater().inflate(R.menu.main, menu);
            return true;
      }

}
```

Each fragment should now take up the whole screen, and you should be able to swipe between them. Note that MainActivity.java extends FragmentActivity instead of Activity. This is necessary if you want to support anything before Android 3.0. If you don't need to do this, you only have to extend Activity.

Adding Tabs

Tabs work in conjunction with the ActionBar, which is the bar at the top of all the examples we've been working with thus far. You create new tabs by requesting one from the ActionBar using its newTab() method, set its title, give it a listener to respond to events, and then add it to the ActionBar with its addTab() method. This is all demonstrated in the following snippet, which should go in the onCreate() method of MainActivity.java:

```
final ActionBar actionBar = getActionBar();
actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
TabListener tabListener = new TabListener() {
    public void onTabSelected(Tab tab,
                                android.app.FragmentTransaction ft) {
        pager.setCurrentItem(tab.getPosition());
    }
    public void onTabReselected(Tab arg0,
                                    android.app.FragmentTransaction arg1) {
        // TODO Auto-generated method stub
    }
    public void onTabUnselected(Tab arg0,
                                    android.app.FragmentTransaction arg1) {
        // TODO Auto-generated method stub
    }
};

actionBar.addTab(actionBar.newTab().setText("Home").setTabListene
r));
actionBar.addTab(actionBar.newTab().setText("Articles").setTabLis
tener));
```

The onTabSelected() callback method gets called whenever a tab is selected. To display the associated fragment, all we need to do is tell the ViewPager to change its currentItem property. You should now be able to select tabs to change the fragments, but the tabs aren't updated when you swipe between them. To fix this, we need to listen for page changes and update the tabs when the user swipes, like so:

```java
pager.setOnPageChangeListener(new ViewPager.SimpleOnPageChangeListener() {
    public void onPageSelected(int position) {
        actionBar.setSelectedNavigationItem(position);
    }
});
```

Tabs and swiping should now be working properly.

Figure 60: Using tabs to navigate between fragments

Summary

In this chapter, we learned how to encapsulate reusable sections of a user interface with Fragment subclasses and display them using swipe views and tabs. This is a common use case for fragments, but they also enable other navigation patterns, like the master-detail pattern, flexible user interfaces, and complex multi-panel layouts.

The next chapter moves away from Android's user interface frameworks and explains how to save and load the data that your application collects.

Chapter 7   Application Data

The majority of *Android Programming Succinctly* has focused on developing the user interface for an Android app. Understanding how to manage the activity lifecycle, display information, collect input, and lay out screens will go a long way towards creating your first Android app. However, most apps also need a way to store the data they collect. In this chapter, we'll take a brief look at several of Android's data storage options.

First, we'll look at shared preferences, which are simple key-value pairs that persist outside of your application. Then, we'll learn how to access Android's internal storage. Finally, we'll introduce Android's SQLite API. The *ApplicationData* project provides a working example of all three of these storage mechanisms.

Shared Preferences

Android's shared preferences framework is the easiest way to store information between user sessions. It allows you to store primitive data (Booleans, floats, ints, longs, and strings) using key-value pairs, much like a persistent hashtable. An activity can have one or more `SharedPreferences` objects associated with it.

The [SharedPreferences](#) class provides access to stored values, and the [SharedPreferences.Editor](#) class lets you modify those values. To store values with `SharedPreferences`, you first need to get access to the shared preferences for the activity. If you only need one preferences file for an activity, you should use [getPreferences()](#), but if you need multiple files, you can specify the preference file names using the [getSharedPreferences()](#) method.

Both of these methods also let you specify who is allowed to access and modify the preference file. While it's possible to create preference files that are accessible from other apps using the `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE`, this can open up security vulnerabilities in your application. So, you should *always* use `MODE_PRIVATE` as the scope for preference files. If you need to share stored values with other apps, you should use something like [ContentProvider](#).

Once you have an instance of `SharedPreferences`, you can read saved values by passing the desired key to methods like `getBoolean()`, `getInt()`, `getFloat()`, and `getString()`. To record values, you first need to get a `SharedPreferences.Editor` object by calling `edit()` on the `SharedPreferences` instance. Then, you can set key-value pairs with methods like `putBoolean()`, `putFloat()`, etc. Finally, you *must* call the editor's `commit()` method to save any updated values.

The following version of `MainActivity.java` shows you how to record input from an `<EditText>` element using `SharedPreferences` and display that value when the activity loads:

```java
package com.example.applicationdata;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.KeyEvent;
import android.widget.TextView;
import android.widget.TextView.OnEditorActionListener;
import android.widget.EditText;

import android.content.SharedPreferences;

public class MainActivity extends Activity {

      private static String SHARED_PREFS_KEY = "existingInput";

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);

            // Set up the EditText
            EditText prefsText = (EditText)
findViewById(R.id.sharedPrefsText);
            prefsText.setOnEditorActionListener(new OnEditorActionListener() {
                public boolean onEditorAction(TextView textView,
                                              int actionId,
                                              KeyEvent event) {
                  String input = textView.getText().toString();
                  saveStringWithSharedPreferences(SHARED_PREFS_KEY, input);
                    return false;
                }
            });

            // Load the string from SharedPreferences
            SharedPreferences prefs = getPreferences(MODE_PRIVATE);
            String existingInput = prefs.getString(SHARED_PREFS_KEY, "");
            prefsText.setText(existingInput);
      }

      public void saveStringWithSharedPreferences(String key, String value) {
            // Get the SharedPreferences editor.
            SharedPreferences prefs = getPreferences(MODE_PRIVATE);
            SharedPreferences.Editor editor = prefs.edit();

            // Save the string.
            editor.putString(key,  value);

            // Commit the changes.
            editor.commit();
      }
}
```

Note that `SharedPreferences`'s getter methods let you specify a default value as their second parameter, as you can see in the `pref.getString(SHARED_PREFS_KEY, "")` call in the above code.

Internal Storage

While `SharedPreferences` offers a convenient abstraction for storing simple data, it's not always appropriate for more complex data structures or for recording a user's documents. As an alternative, Android apps can store information directly on the device's hard drive. However, like `SharedPreferences`, these files are private—other apps shouldn't be allowed to access them due to security vulnerabilities. Files saved to internal storage are deleted when your app in uninstalled.

To save data to a file, you first need to open the file with Context.openFileOutput(). This returns a FileOutputStream, whose `write()` method enables you to add bytes to the file. When you're done writing data to the file, you have to close it with its `close()` method. The following method shows you how to store a string in a file:

```java
public void saveStringWithInternalStorage(String filename,
                String value) throws IOException {
    FileOutputStream output = openFileOutput(filename, MODE_PRIVATE);
    byte[] data = value.getBytes();
    output.write(data);
    output.close();
}
```

Note that `FileOutputStream` works with bytes, so the string has to be converted before passing it to `write()`.

To read back this string data, you need to open the file with openFileInput(), which returns a FileInputStream object. Then you can read the file contents into a byte array. When you're done, don't forget to close the input stream. The following example loads the string saved by the previous snippet and displays it in an `EditText` widget:

```java
FileInputStream input = null;
try {
    // Open the file.
    input = openFileInput(FILENAME);
    // Read the byte data.
    int maxBytes = input.available();
    byte[] data = new byte[maxBytes];
    input.read(data, 0, maxBytes);
    while (input.read(data) != -1) {};
    // Turn it into a String and display it.
    String existingInput = new String(data);
    prefsText.setText(existingInput);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (input != null) {
    try {
        input.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
}
```

The `FileInputStream.available()` method returns the estimated number of bytes that can be read without blocking for more input. We use this to figure out the length of a new byte array, which is populated by `FileInputStream.read()`. These bytes are then converted to a string and displayed to the user.

SQLite Databases

Android also provides built-in support for local SQLite databases. This is useful for data-intensive applications that need to be able to collect large amounts of information and query it efficiently. This section shows you how to create SQLite database on an Android device, create a table, insert data into it, and query it using SQL.

Mixing raw SQL with the rest of your application code can get messy very quickly, so Android uses certain conventions to abstract the database interaction as much as possible from your application code. For example, each table should be represented by a dedicated class. This provides a single place to define your table and column names, as well as a standardized way to create, upgrade, and access the table.

Representing Databases

To create the class representation for a particular table, you need to extend the SQLiteOpenHelper class and override its `onCreate()` and `onUpgrade()` methods to create the table and upgrade it to a new version, respectively. It's also customary to define the table name and all of the columns as static final variables in this class. For example, a table called *messages* stored in the SQLite database file *messages.db* might be represented by a class called `MessageOpenHelper` that looks like this:

```java
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;

public class MessageOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "messages.db";

    public static final String TABLE_MESSAGES = "messages";
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_AUTHOR = "author";
    public static final String COLUMN_MESSAGE = "message";

    private static final String DATABASE_CREATE = "create table "
                + TABLE_MESSAGES + "("
                + COLUMN_ID + " integer primary key autoincrement, "
```

```java
                          + COLUMN_AUTHOR + " text not null"
                          + COLUMN_MESSAGE + " text not null);";

    public MessageOpenHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
            db.execSQL(DATABASE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
            // This implementation will destroy all the old data, which
            // probably isn't what you want to do in a real application.
            db.execSQL("drop table if exists " + TABLE_MESSAGES);
            onCreate(db);
    }

}
```

The entire table is defined by the final static variables at the beginning of the class. `DATABASE_VERSION` and `DATABASE_NAME` define the SQLite database version and filename, while the following four lines define a table called *messages* with three columns called `_id`, `author`, and `message`. Finally, the `DATABASE_CREATE` variable contains the raw SQL to create the *messages* table. This is one of the only places you'll be writing raw SQL, and it's neatly contained in a static final variable. Again, this structure is more of a suggested convention than a hard-and-fast rule.

The constructor for the class needs to pass the database name and version to the superclass. The job of the `onCreate()` method is to initialize the SQL table associated with the class. It is passed a [SQLiteDatabase](#) instance representing the database, and all we have to do is execute the `DATABASE_CREATE` string that we defined earlier using its `execSQL()` method. Similarly, the job of the `onUpgrade()` method is to upgrade the table to a new version. The above implementation simply drops the table and recreates it, which may or may not be desirable for your real-world application.

Accessing the Database

`SQLiteOpenHelper` methods make it very easy to create and access the underlying database. All you have to do is instantiate your custom `SQLiteOpenHelper` and request a database with either `getReadableDatabase()` or `getWritableDatabase()`. For example, to create a database called `messages.db` and connect to it, all you need is the following:

```java
MessageOpenHelper dbHelper = new MessageOpenHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();
```

The getReadableDatabase() and getWritableDatabase() methods are responsible for creating the underlying SQLite database if it doesn't already exist *and* opening it for reading and/or writing. The onCreate() method defined by MessageOpenHelper is used to create the database. The returned SQLiteDatabase object provides methods for altering and querying tables in the database.

You should always close the database by calling close() on your SQLiteOpenHelper instance when you are done interacting with the database.

Inserting Rows

Inserting rows into a SQLiteDatabase object is a two-step process. First, you need to create a [ContentValues](#) object to represent the values that you want to insert. A single ContentValues instance represents a single record, and you define it by passing each column and value to its put() method. Typically, you'll want to take the column names from the static variables defined in your custom SQLiteOpenHelper.

Second, you need to pass that ContentValues object to the SQLiteDatabase's insert() method, along with the name of the table that you want to insert into. For example, the following method (which is defined in the example project's MainActivity.java) adds a new record to the *messages* table each time it is called.

```java
public void saveStringWithDatabase(String value) {
    // Store the author and message in a ContentValues object.
    ContentValues values = new ContentValues();
    values.put(MessageOpenHelper.COLUMN_AUTHOR, AUTHOR_NAME);
    values.put(MessageOpenHelper.COLUMN_MESSAGE, value);

    // Record that ContentValues in a SQLite database
    MessageOpenHelper dbHelper = new MessageOpenHelper(this);
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    long id = db.insert(MessageOpenHelper.TABLE_MESSAGES, null, values);
    Log.d(TAG,
            String.format("Saved new record to database with ID: %d", id));
    dbHelper.close();
      }
```

Notice how we called dbHelper.close() call when we were done using the database. Also notice how we access the table name via MessageOpenHelper's static variable so that it isn't accidentally mistyped.

Querying the Database

To retrieve records from a SQLiteDatabase instance, you pass your query information to one of its query() methods. The various query() overloads provide parameters to all of the standard SQL query parameters. You can define the columns to select, the selection constraints, the grouping and ordering behavior, and selection limits.

Records are returned as [Cursor] objects, which you can use to iterate through the selected rows and cast the contained values to Java types. For example, the following snippet opens messages.db and selects the _id and message columns from rows that have AUTHOR_NAME in the author column:

```
// Load the most recent record from the SQLite database.
MessageOpenHelper dbHelper = new MessageOpenHelper(this);
SQLiteDatabase db = dbHelper.getReadableDatabase();
// Fetch the records with the appropriate author name.
String[] columns = {MessageOpenHelper.COLUMN_ID,
                    MessageOpenHelper.COLUMN_MESSAGE};
String selection = MessageOpenHelper.COLUMN_AUTHOR + " = '" + AUTHOR_NAME +
"'";
Cursor cursor = db.query(MessageOpenHelper.TABLE_MESSAGES,
                         columns, selection, null, null, null, null);
// Display the most recent record in the text field.
cursor.moveToLast();
long id = cursor.getLong(0);
String message = cursor.getString(1);
Log.d(TAG, String.format("Retrieved info from database. ID: %d Message: %s",
id, message));
prefsText.setText(message);
// Clean up.
cursor.close();
dbHelper.close();
```

The moveToLast() method moves the cursor to the last selected row, which in this case should be the most recent record. To extract the values, you use methods like getLong() and getString(), passing in the column position. Note that this position is defined by the columns array that we passed to query(), not the order of the columns in the database. When we were done with the results, we cleaned up by calling close() on both cursor and the SQLiteOpenHelper.

While this section only covered the basics of Android's SQLite API, keep in mind that Android also provides more advanced SQL functionality, including database locking and transactions.

Summary

This chapter discussed three of the most common ways to store data on an Android device. We began with shared preferences, which provide a convenient way to store key-value pairs. Then, we learned how to store data in files on the device's internal storage, which is more flexible than shared preferences. Finally, we took a brief look at Android's built-in SQLite capabilities by creating a SQLite database, inserting some rows, and reading them back out.

The majority of this book discussed how to display and collect information from the user. Combined with this chapter, you should now be able to collect and store almost any kind of user data you could possibly need. I hope that, armed with these skills, you're feeling ready to venture out into the Android ecosystem and start building your own Android applications. Good luck!