Android Programming: The Big Nerd Ranch Guide

by Bill Phillips, Chris Stewart, Brian Hardy and Kristin Marsicano

Big Nerd Ranch, LLC.
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
http://www.bignerdranch.com/
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, Inc.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0134171454
ISBN-13 978-0134171456

Second edition, second printing, February 2016

Changes in the second printing

This is the second printing of the second edition of this book. Since the first printing we have made some minor corrections, and updated how new activities are created in Android Studio. When the first printing was made, the best template to start with was called the Blank Activity template. It is now called the Empty Activity template, so we have changed its name throughout the book.

The tools will continue to change, but do not worry, the fundamentals in this book will stand the test of time. You can keep an eye on the forums (`http://forums.bignerdranch.com`) for adjustments based on those future updates.

How to Use This Book

This book is not a reference book. Its goal is to get you over the initial hump to where you can get the most out of the reference and recipe books available. It is based on our five-day class at Big Nerd Ranch. As such, it is meant to be worked through from the beginning. Chapters build on each other and skipping around is unproductive.

In our classes, students work through these materials, but they also benefit from the right environment – a dedicated classroom, good food and comfortable board, a group of motivated peers, and an instructor to answer questions.

As a reader, you want your environment to be similar. That means getting a good night's rest and finding a quiet place to work. These things can help, too:

- Start a reading group with your friends or coworkers.
- Arrange to have blocks of focused time to work on chapters.
- Participate in the forum for this book at `http://forums.bignerdranch.com`.
- Find someone who knows Android to help you out.

How This Book is Organized

As you work through this book, you will write eight Android apps. A couple are very simple and take only a chapter to create. Others are more complex. The longest app spans 11 chapters. All are designed to teach you important concepts and techniques and give you direct experience using them.

GeoQuiz	In your first app, you will explore the fundamentals of Android projects, activities, layouts, and explicit intents.
CriminalIntent	The largest app in the book, CriminalIntent lets you keep a record of your colleagues' lapses around the office. You will learn to use fragments, master-detail interfaces, list-backed interfaces, menus, the camera, implicit intents, and more.
BeatBox	Intimidate your foes with this app while you learn more about fragments, media playback, themes, and drawables.
NerdLauncher	Building this custom launcher will give you insight into the intent system and tasks.

PhotoGallery	A Flickr client that downloads and displays photos from Flickr's public feed, this app will take you through services, multithreading, accessing web services, and more.
DragAndDraw	In this simple drawing app, you will learn about handling touch events and creating custom views.
Sunset	In this toy app, you will create a beautiful representation of a sunset over open water while learning about animations.
Locatr	This app lets you query Flickr for pictures around your current location and display them on a map. In it, you will learn how to use location services and maps.

Challenges

Most chapters have a section at the end with exercises for you to work through. This is your opportunity to use what you have learned, explore the documentation, and do some problem solving on your own.

We strongly recommend that you do the challenges. Going off the beaten path and finding your way will solidify your learning and give you confidence with your own projects.

If you get lost, you can always visit `http://forums.bignerdranch.com` for some assistance.

Are you more curious?

There are also sections at the ends of chapters labeled "For the More Curious." These sections offer deeper explanations or additional information about topics presented in the chapter. The information in these sections is not absolutely essential, but we hope you will find it interesting and useful.

Code Style

There are two areas where our choices differ from what you might see elsewhere in the Android community:

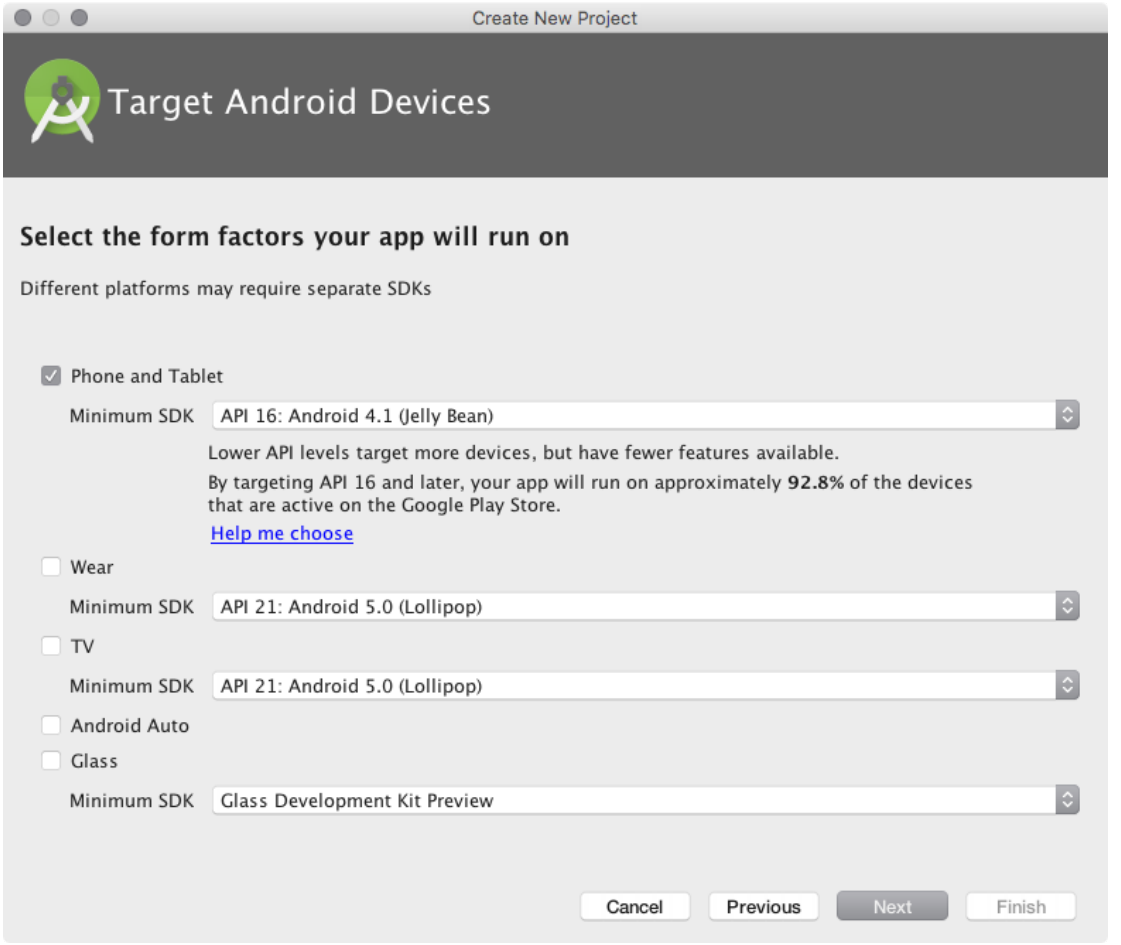
We use anonymous inner classes for listeners.

This is mostly a matter of opinion. We find it makes for cleaner code in the applications in this book because it puts the listener's method implementations right where you want to see them. In high-performance contexts or large applications, anonymous inner classes may cause problems, but for most circumstances they work fine.

After we introduce fragments in Chapter 7, we use them for all user interfaces.

Fragments are not an absolutely necessary tool but we find that, when used correctly, they are a valuable tool in any Android developer's toolkit. Once you get comfortable with fragments, they are not that difficult to work with. Fragments have clear advantages over activities that make them worth the effort, including flexibility in building and presenting your user interfaces.

Figure 1.5 Specifying device support

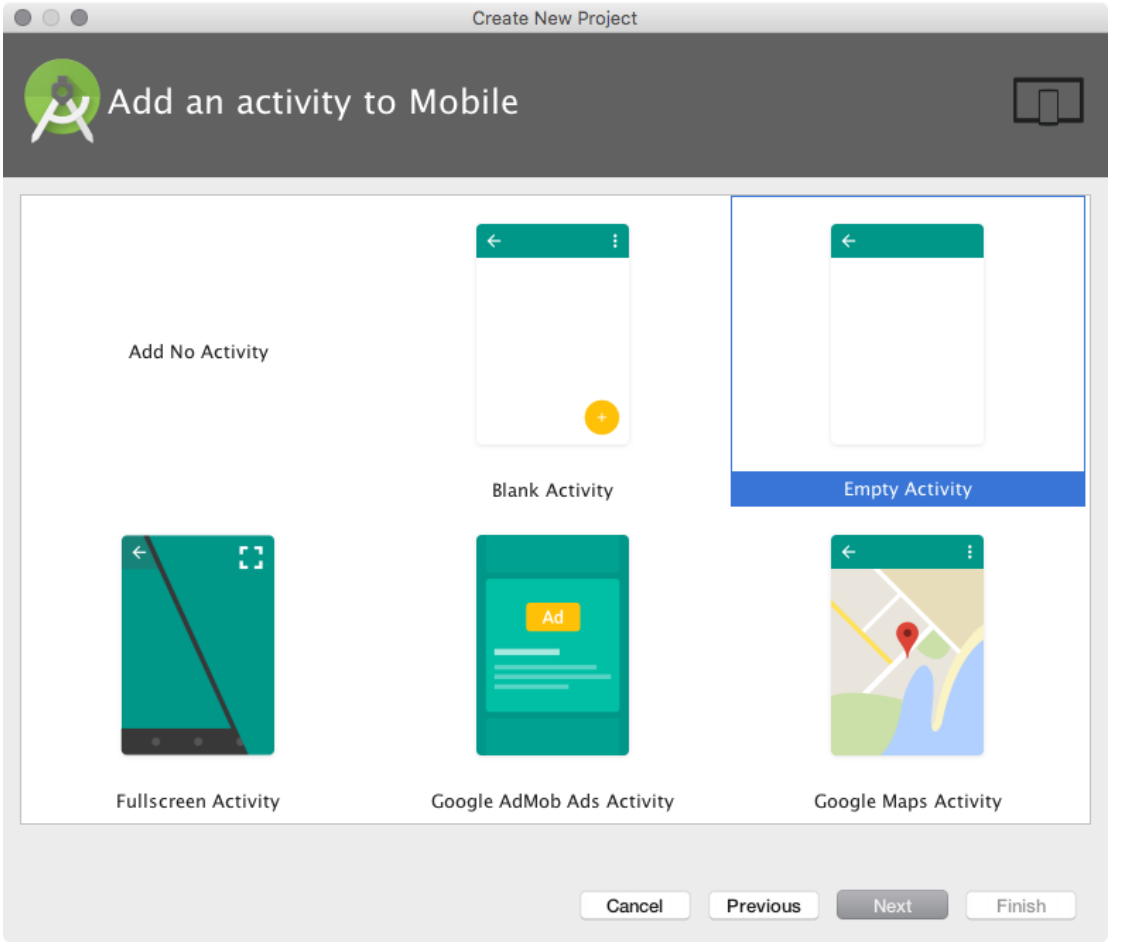


Click Next.

In the next screen, you are prompted to choose a template for the first screen of GeoQuiz (Figure 1.6). You want the most basic template available. Choose Empty Activity and click Next.

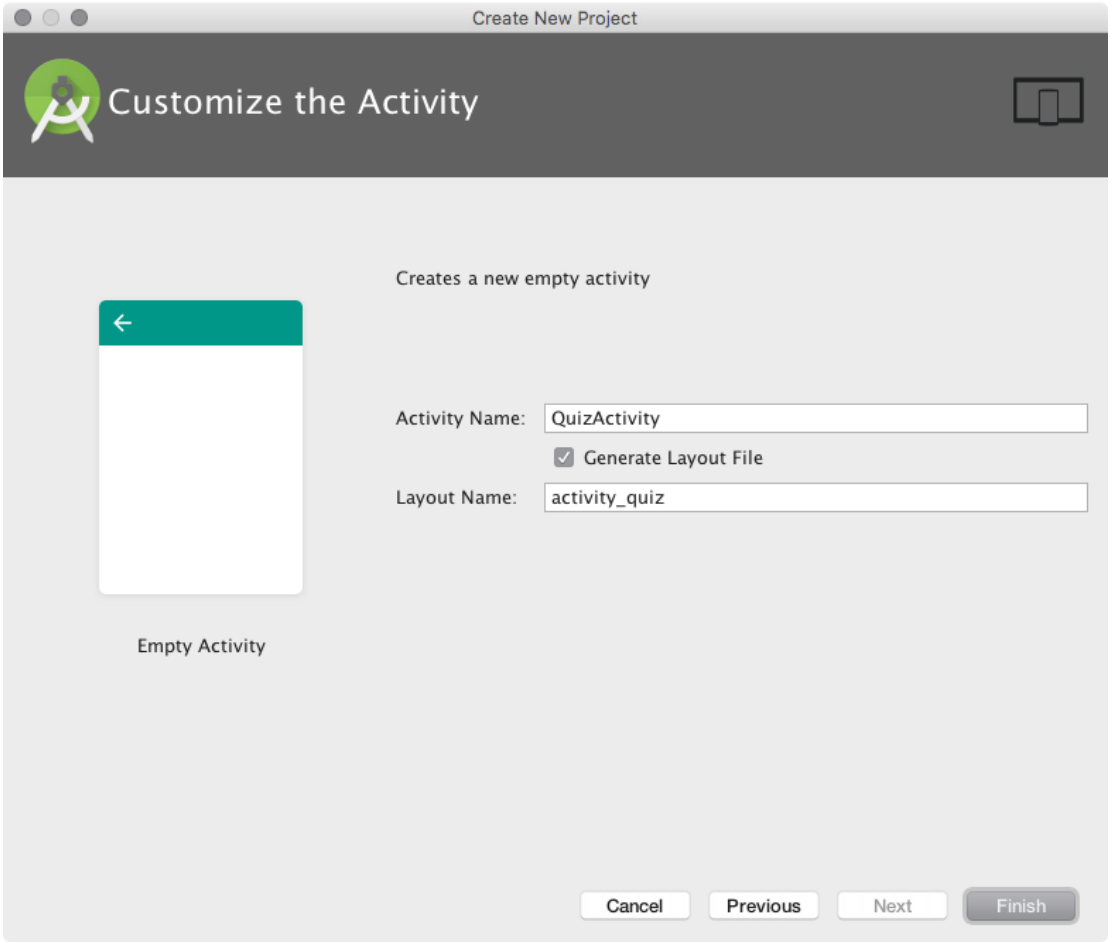
(Android Studio updates regularly, so your wizard may look slightly different from what we are showing you. This is usually not a problem; the choices should be similar. If your wizard looks very different, then the tools have changed more drastically. Do not panic. Head to this book's forum at `forums.bignerdranch.com` and we will help you navigate the latest version.)

Figure 1.6 Choosing a type of Activity



In the final dialog of this wizard, name the activity subclass **`QuizActivity`** (Figure 1.7). Notice the **`Activity`** suffix on the class name. This is not required, but it is an excellent convention to follow.

Figure 1.7 Configuring the new activity



Leave Generate Layout File checked. The layout name will automatically update to `activity_quiz` to reflect the activity's new name. The layout name reverses the order of the activity name, is all lowercase, and has underscores between words. This naming style is recommended for layouts as well as other resources that you will learn about later.

If your version of Android Studio has other options on this screen, leave them as is. Click Finish. Android Studio will create and open your new project.

Creating string resources

Every project includes a default strings file named `strings.xml`.

In the Project tool window, find the `app/res/values` directory, reveal its contents, and open `strings.xml`.

The template has already added a few string resources for you. Remove the unused string named `hello_world` and add the three new strings that your layout requires.

Listing 1.3 Adding string resources (`strings.xml`)

```
<resources>
    <string name="app_name">GeoQuiz</string>

    <string name="question_text">
        Constantinople is the largest city in Turkey.
    </string>
    <string name="true_button">True</string>
    <string name="false_button">False</string>
</resources>
```

(Depending on your version of Android Studio, you may have additional strings. Do not delete them. Deleting them could cause cascading errors in other files.)

Now, whenever you refer to `@string/false_button` in any XML file in the GeoQuiz project, you will get the literal string “False” at runtime.

Save `strings.xml`. If you had errors in `activity_quiz.xml` about the missing string resources, they should now be gone. (If you still have errors, check both files for typos.)

Although the default strings file is named `strings.xml`, you can name a strings file anything you want. You can also have multiple strings files in a project. As long as the file is located in `res/values/`, has a `resources` root element, and contains child `string` elements, your strings will be found and used appropriately.

Previewing the layout

Your layout is now complete, and you can preview the layout in the graphical layout tool (Figure 1.12). First, make sure that your files are saved and error free. Then return to `activity_quiz.xml` and open the Preview tool window (if it is not already open) using the tab to the right of the editor.

Listing 1.4 Default class file for **QuizActivity** (QuizActivity.java)

```
package com.bignerdranch.android.geoquiz;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;


public class QuizActivity extends AppCompatActivity {

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

}
```

(Wondering what **AppCompatActivity** is? It is a subclass of Android's **Activity** class that provides compatibility support for older versions of Android. You will learn much more about **AppCompatActivity** in Chapter 13.)

If you are not seeing all of the import statements, click the symbol to the left of the first import statement to reveal the others.

This file has one **Activity** method: **onCreate(Bundle)**.

(If your file has **onCreateOptionsMenu(Menu)** and **onOptionsItemSelected(MenuItem)** methods, ignore them for now. You will return to menus in detail in Chapter 13.)

The **onCreate(Bundle)** method is called when an instance of the activity subclass is created. When an activity is created, it needs a user interface to manage. To get the activity its user interface, you call the following **Activity** method:

```
public void setContentView(int layoutResID)
```

This method *inflates* a layout and puts it on screen. When a layout is *inflated*, each widget in the layout file is instantiated as defined by its attributes. You specify which layout to inflate by passing in the layout's resource ID.

Listing 1.6 Adding IDs to **`Button`**s (`activity_quiz.xml`)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
... >

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:padding="24dp"
    android:text="@string/question_text" />

  <LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <Button
      android:id="@+id/true_button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/true_button" />

    <Button
      android:id="@+id/false_button"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/false_button" />

  </LinearLayout>

</LinearLayout>
```

Notice that there is a + sign in the values for `android:id` but not in the values for `android:text`. This is because you are *creating* the IDs and only *referencing* the strings.

Wiring Up Widgets

Now that the buttons have resource IDs, you can access them in **`QuizActivity`**. The first step is to add two member variables.

Type the following code into `QuizActivity.java`. (Do not use code completion; type it in yourself.) After you save the file, it will report two errors.

Listing 1.7 Adding member variables (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    private Button mTrueButton;
    private Button mFalseButton;

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

}
```

You will fix the errors in just a second. First, notice the `m` prefix on the two member (instance) variable names. This prefix is an Android naming convention that we will follow throughout this book.

Now mouse over the red error indicators. They report the same problem: Cannot resolve symbol 'Button'.

These errors are telling you that you need to import the **`android.widget.Button`** class into `QuizActivity.java`. You could type the following import statement at the top of the file:

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

Or you can do it the easy way and let Android Studio do it for you. Just press Option+Return (or Alt +Enter) to let the IntelliJ magic under the hood amaze you. The new import statement now appears with the others at the top of the file. This shortcut is generally useful when something is not correct with your code. Try it often!

This should get rid of the errors. (If you still have errors, check for typos in your code and XML.)

Now you can wire up your button widgets. This is a two-step process:

- get references to the inflated **`View`** objects
- set listeners on those objects to respond to user actions

Getting references to widgets

In an activity, you can get a reference to an inflated widget by calling the following **`Activity`** method:

```
public View findViewById(int id)
```

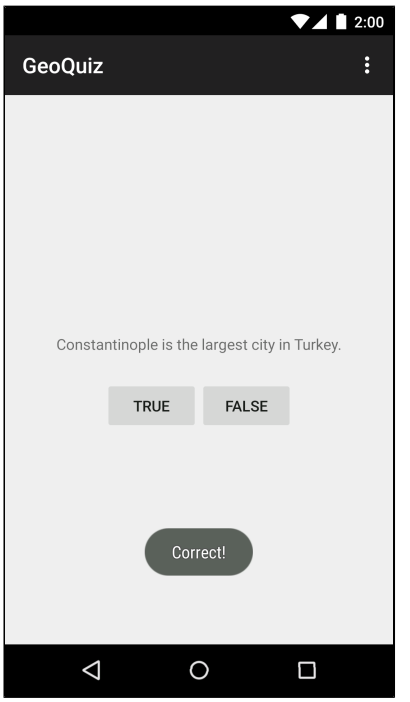
This method accepts a resource ID of a widget and returns a **`View`** object.

In `QuizActivity.java`, use the resource IDs of your buttons to retrieve the inflated objects and assign them to your member variables. Note that you must cast the returned **`View`** to **`Button`** before assigning it.

Listing 1.8 Getting references to widgets (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    private Button mTrueButton;
    private Button mFalseButton;

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

        mTrueButton = (Button) findViewById(R.id.true_button);
        mFalseButton = (Button) findViewById(R.id.false_button);
    }

}
```

Figure 1.14 A toast providing feedback



First, return to `strings.xml` and add the string resources that your toasts will display.

Listing 1.11 Adding toast strings (`strings.xml`)

```
<resources>
  <string name="app_name">GeoQuiz</string>

  <string name="question_text">Constantinople is the largest city in Turkey.</string>
  <string name="true_button">True</string>
  <string name="false_button">False</string>
  <string name="correct_toast">Correct!</string>
  <string name="incorrect_toast">Incorrect!</string>
</resources>
```

To create a toast, you call the following method from the **`Toast`** class:

```
    public static Toast makeText(Context context, int resId, int duration)
```

The **`Context`** parameter is typically an instance of **`Activity`** (**`Activity`** is a subclass of **`Context`**). The second parameter is the resource ID of the string that the toast should display. The **`Context`** is needed by the **`Toast`** class to be able to find and use the string's resource ID. The third parameter is one of two **`Toast`** constants that specify how long the toast should be visible.

After you have created a toast, you call **`Toast.show()`** on it to get it on screen.

In **`QuizActivity`**, you are going to call **`makeText(…)`** in each button's listener (Listing 1.12). Instead of typing everything in, try using Android Studio's code completion feature to add these calls.

Keep the emulator running; you do not want to wait for it to launch on every run.

You can stop the app by pressing Android's Back button on the emulator. The Back button is shaped like a left-pointing triangle (on older versions of Android, it looks like an arrow that is making a U-turn). Then re-run the app from Android Studio to test changes.

The emulator is useful, but testing on a real device gives more accurate results. In Chapter 2, you will run GeoQuiz on a hardware device. You will also give GeoQuiz more geography questions with which to test the user.

For the More Curious: Android Build Process

By now, you probably have some burning questions about how the Android build process works. You have already seen that Android Studio builds your project automatically as you modify it rather than on command. During the build process, the Android tools take your resources, code, and the `AndroidManifest.xml` file (which contains meta-data about the application) and turn them into an `.apk` file. This file is then signed with a debug key, which allows it to run on the emulator. (To distribute your `.apk` to the masses, you have to sign it with a release key. There is more information about this process in the Android developer documentation at `http://developer.android.com/tools/publishing/preparing.html`.)

Figure 1.19 shows the complete build process.

Listing 2.5 Adding question strings in advance (`strings.xml`)

```
...

  <string name="incorrect_toast">Incorrect!</string>
  <string name="question_oceans">The Pacific Ocean is larger than
    the Atlantic Ocean.</string>
  <string name="question_mideast">The Suez Canal connects the Red Sea
    and the Indian Ocean.</string>
  <string name="question_africa">The source of the Nile River is in Egypt.</string>
  <string name="question_americas">The Amazon River is the longest river
    in the Americas.</string>
  <string name="question_asia">Lake Baikal is the world\'s oldest and deepest
    freshwater lake.</string>
  ...
```

Notice that you use the escape sequence `\'` in the last value to get an apostrophe in your string. You can use all the usual escape sequences in your string resources, such as `\n` for a new line.

Save your files. Then return to `activity_quiz.xml` and preview your layout changes in the graphical layout tool.

That is all for now for GeoQuiz's view layer. Time to wire everything up in your controller class, **`QuizActivity`**.

Updating the Controller Layer

In the previous chapter, there was not much happening in GeoQuiz's one controller, **`QuizActivity`**. It displayed the layout defined in `activity_quiz.xml`. It set listeners on two buttons and wired them to make toasts.

Now that you have multiple questions to retrieve and display, **`QuizActivity`** will have to work harder to tie GeoQuiz's model and view layers together.

Open `QuizActivity.java`. Add variables for the **`TextView`** and the new **`Button`**. Also, create an array of **`Question`** objects and an index for the array.

Listing 2.6 Adding variables and a **`Question`** array (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    private Button mTrueButton;
    private Button mFalseButton;
    private Button mNextButton;
    private TextView mQuestionTextView;

    private Question[] mQuestionBank = new Question[] {
        new Question(R.string.question_oceans, true),
        new Question(R.string.question_mideast, false),
        new Question(R.string.question_africa, false),
        new Question(R.string.question_americas, true),
        new Question(R.string.question_asia, true),
    };

    private int mCurrentIndex = 0;
    ...
```

Download this file and open the `02_MVC/GeoQuiz/app/src/main/res` directory. Within this directory, locate the `drawable-hdpi`, `drawable-mdpi`, `drawable-xhdpi`, and `drawable-xxhdpi` directories.

The suffixes on these directory names refer to the screen pixel density of a device:

`mdpi` medium-density screens (~160dpi)

`hdpi` high-density screens (~240dpi)

`xhdpi` extra-high-density screens (~320dpi)

`xxhdpi` extra-extra-high-density screens (~480dpi)

(There are a few other density categories that are omitted from the solutions, including ldpi and xxxhdpi.)

Within each directory, you will find two image files – `arrow_right.png` and `arrow_left.png`. These files have been customized for the screen pixel density specified in the directory's name.

You are going to include all the image files from the solutions in GeoQuiz. When it runs, the OS will choose the best image file for the specific device running the app. Note that by duplicating the images multiple times, you increase the size of your application. In this case, this is not a problem because GeoQuiz is a simple app.

If an app runs on a device that has a screen density not included in any of the application's screen density qualifiers, Android will automatically scale the available image to the appropriate size for the device. Thanks to this feature, it is not necessary to provide images for all of the pixel density buckets. To reduce the size of your application, you can focus on one or a few of the higher resolution buckets and selectively optimize for lower resolutions when Android's automatic scaling provides an image with artifacts on those lower resolution devices.

(For alternatives to duplicating images at different densities and an explanation of your `mipmap` directory, see Chapter 21.)

Adding resources to a project

The next step is to add the image files to GeoQuiz's resources.

First, confirm that you have the necessary drawable folders. Make sure the project tools window is displaying the Project view (select Project from the dropdown at the top of the project tools window, as shown in Figure 1.13 in Chapter 1). Expand the contents of `GeoQuiz/app/src/main/res`. You should see folders named `drawable-hdpi`, `drawable-mdpi`, `drawable-xhdpi`, and `drawable-xxhdpi`, as shown in Figure 2.10. (You will likely see other folders as well. Ignore those for now.)

Listing 3.2 Adding log statement to **`onCreate(…)`** (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate(Bundle) called");
        setContentView(R.layout.activity_quiz);

        ...
    }
}
```

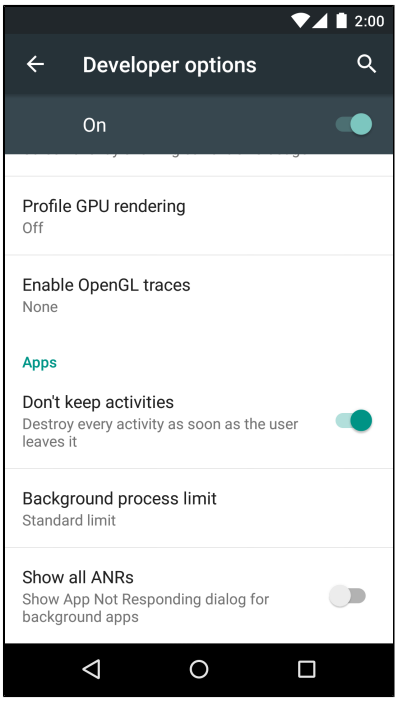
Now override five more methods in **`QuizActivity`** by adding the following after **`onCreate(Bundle)`**:

Listing 3.3 Overriding more lifecycle methods (`QuizActivity.java`)

```
    @Override
    public void onStart() {
        super.onStart();
        Log.d(TAG, "onStart() called");
    }

    @Override
    public void onPause() {
        super.onPause();
        Log.d(TAG, "onPause() called");
    }

    @Override
    public void onResume() {
        super.onResume();
        Log.d(TAG, "onResume() called");
    }

    @Override
    public void onStop() {
        super.onStop();
        Log.d(TAG, "onStop() called");
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy() called");
    }

    ...
}
```

Notice that you call the superclass implementations before you log your messages. These superclass calls are required. Calling the superclass implementation before you do anything else is critical in **`onCreate(…)`**; the order is less important in the other methods.

Figure 3.15 Don’t keep activities selected



Now run your app and press the Home button. Pressing Home causes the activity to be paused and stopped. Then the stopped activity will be destroyed just as if the Android OS had reclaimed it for its memory. Then you can restore the app to see if your state was saved as you expected. *Be sure to turn this setting off when you are done testing, as it will cause a performance decrease and some apps will perform poorly.*

Pressing the Back button instead of the Home button will always destroy the activity, regardless of whether you have this development setting on. Pressing the Back button tells the OS that the user is done with the activity.

For the More Curious: Logging Levels and Methods

When you use the **`android.util.Log`** class to send log messages, you control not only the content of a message, but also a *level* that specifies how important the message is. Android supports five log levels, shown in Figure 3.16. Each level has a corresponding method in the **`Log`** class. Sending output to the log is as simple as calling the corresponding **`Log`** method.

Listing 5.1 Adding strings (`strings.xml`)

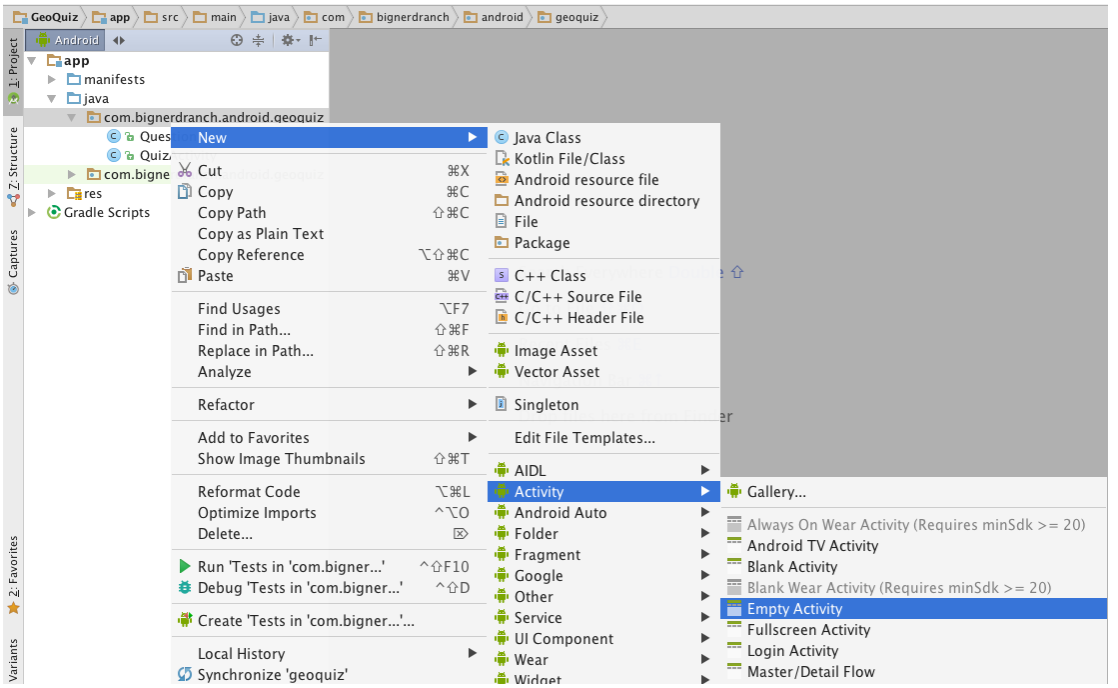
```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    ...
    <string name="question_asia">Lake Baikal is the world\'s oldest and deepest
      freshwater lake.</string>
    <string name="warning_text">Are you sure you want to do this?</string>
    <string name="show_answer_button">Show Answer</string>
    <string name="cheat_button">Cheat!</string>
    <string name="judgment_toast">Cheating is wrong.</string>

</resources>
```

Creating a new activity

Creating an activity typically involves touching at least three files: the Java class file, an XML layout, and the application manifest. If you touch those files in the wrong ways, Android can get mad. To ensure that you do it right, you should use Android Studio's New Activity wizard.

Launch the New Activity wizard by right-clicking on your `com.bignerdranch.android.geoquiz` package in the Project Tool Window. Choose New → Activity → Empty Activity as shown in Figure 5.3.

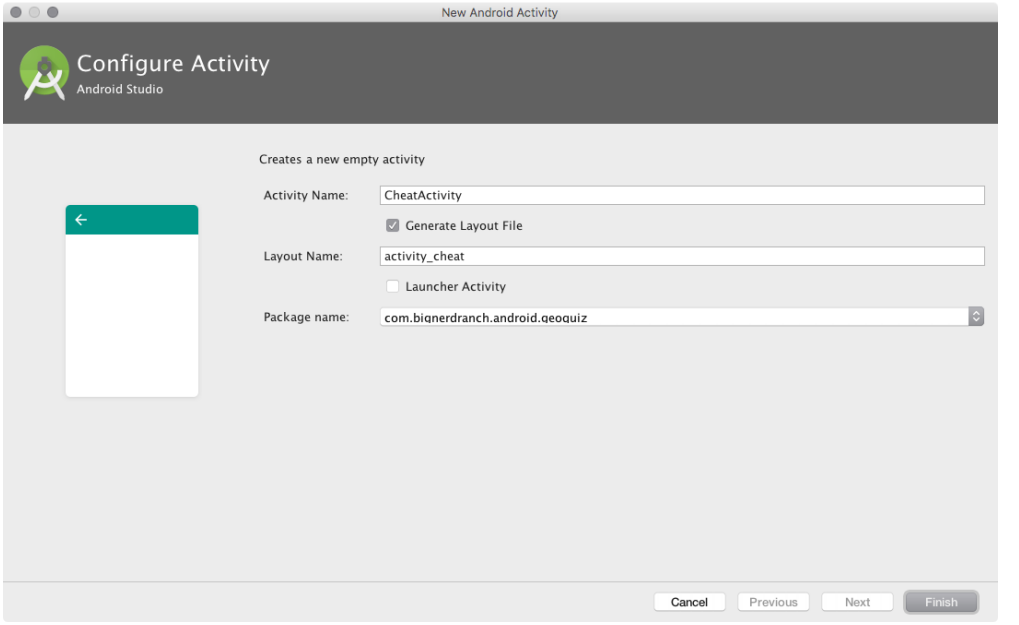
Figure 5.3 The New Activity Wizard menu



You should then see a dialog like Figure 5.4. Set Activity Name to **`CheatActivity`**. This is the name of your **`Activity`** subclass. Layout Name should be automatically set to `activity_cheat`. This will be the base name of the layout file the wizard creates.

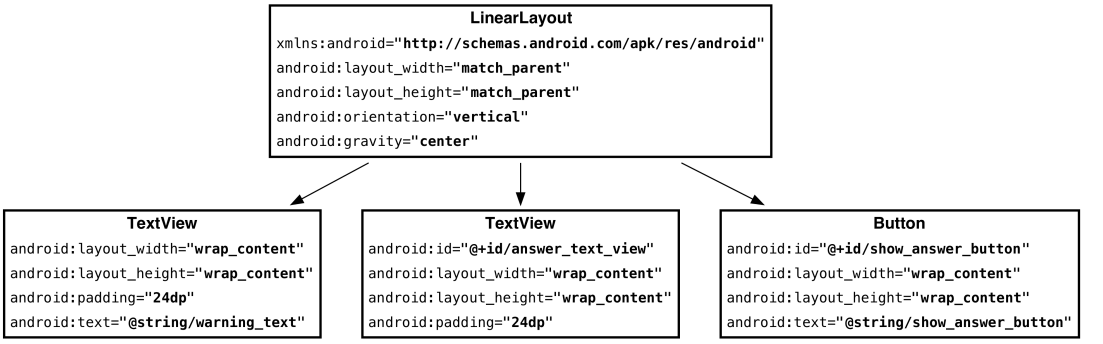
The defaults for the remaining fields should be fine, but take care to ensure that the package name is what you expect. This determines where `CheatActivity.java` will live on the filesystem. Click the Finish button to make the magic happen.

Figure 5.4 The New Empty Activity wizard



Now it is time to make the user interface look good. The screenshot at the beginning of the chapter shows you what **`CheatActivity`**'s view should look like. Figure 5.5 shows the widget definitions.

Figure 5.5 Diagram of layout for **`CheatActivity`**



Listing 5.3 Declaring **CheatActivity** in the manifest (AndroidManifest.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.geoquiz" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".QuizActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".CheatActivity" >
        </activity>
    </application>

</manifest>
```

The `android:name` attribute is required, and the dot at the start of this attribute's value tells the OS that this activity's class is in the package specified in the `package` attribute in the `manifest` element at the top of the file.

You will sometimes see a fully qualified `android:name` attribute: `android:name="com.bignerdranch.android.geoquiz.CheatActivity"`. The long-form notation is identical to the version in Listing 5.3.

There are many interesting things in the manifest, but for now, let's stay focused on getting **CheatActivity** up and running. You will learn about the different parts of the manifest in later chapters.

Adding a Cheat! button to QuizActivity

The plan is for the user to press a button in **QuizActivity** to get an instance of **CheatActivity** on screen. So you need new buttons in `layout/activity_quiz.xml` and `layout-land/activity_quiz.xml`.

In the default layout, add the new button as a direct child of the root **LinearLayout**. Its definition should come right before the Next button.

Listing 5.6 Wiring up the Cheat! button (`QuizActivity.java`)

```
public class QuizActivity extends AppCompatActivity {

    ...

    private Button mNextButton;
    private Button mCheatButton;

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        ...

        mCheatButton = (Button)findViewById(R.id.cheat_button);
        mCheatButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Start CheatActivity
            }
        });

        if (savedInstanceState != null) {
           mCurrentIndex = savedInstanceState.getInt(KEY_INDEX, 0);
        }

        updateQuestion();
    }

 ...

}
```

Now you can get to the business of starting **`CheatActivity`**.

Starting an Activity

The simplest way one activity can start another is with the **`startActivity`** method:

```
public void startActivity(Intent intent)
```

You might guess that **`startActivity(…)`** is a static method that you call on the **`Activity`** subclass that you want to start. But it is not. When an activity calls **`startActivity(…)`**, this call is sent to the OS.

In particular, it is sent to a part of the OS called the **`ActivityManager`**. The **`ActivityManager`** then creates the **`Activity`** instance and calls its **`onCreate(…)`** method, as shown in Figure 5.7.

An activity may be started from several different places, so you should define keys for extras on the activities that retrieve and use them. Using your package name as a qualifier for your extra, as shown in Listing 5.8, prevents name collisions with extras from other apps.

Now you could return to **`QuizActivity`** and put the extra on the intent, but there is a better approach. There is no reason for **`QuizActivity`**, or any other code in your app, to know the implementation details of what **`CheatActivity`** expects as extras on its **`Intent`**. Instead, you can encapsulate that work into a **`newIntent(…)`** method.

Create this method in **`CheatActivity`** now:

Listing 5.9 A **`newIntent(…)`** method for **`CheatActivity`** (`CheatActivity.java`)

```
public class CheatActivity extends AppCompatActivity {

    private static final String EXTRA_ANSWER_IS_TRUE =
            "com.bignerdranch.android.geoquiz.answer_is_true";

    public static Intent newIntent(Context packageContext, boolean answerIsTrue) {
        Intent i = new Intent(packageContext, CheatActivity.class);
        i.putExtra(EXTRA_ANSWER_IS_TRUE, answerIsTrue);
        return i;
    }

    ...
```

This static method allows us to create an **`Intent`** properly configured with the extras **`CheatActivity`** will need. The `answerIsTrue` argument, a `boolean`, is put into the intent with a private name using the `EXTRA_ANSWER_IS_TRUE` constant. You will extract this value momentarily. Using a **`newIntent(…)`** method like this for your activity subclasses will make it easy for other code to properly configure their launching intents.

Speaking of other code, use this new method in **`QuizActivity`**'s cheat button listener now.

Listing 5.10 Launching **`CheatActivity`** with an extra (`QuizActivity.java`)

```
        ...
        mCheatButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Start CheatActivity
                Intent i = new Intent(QuizActivity.this, CheatActivity.class);
                boolean answerIsTrue = mQuestionBank[mCurrentIndex].isAnswerTrue();
                Intent i = CheatActivity.newIntent(QuizActivity.this, answerIsTrue);
                startActivity(i);
            }
        });

        updateQuestion();
    }
```

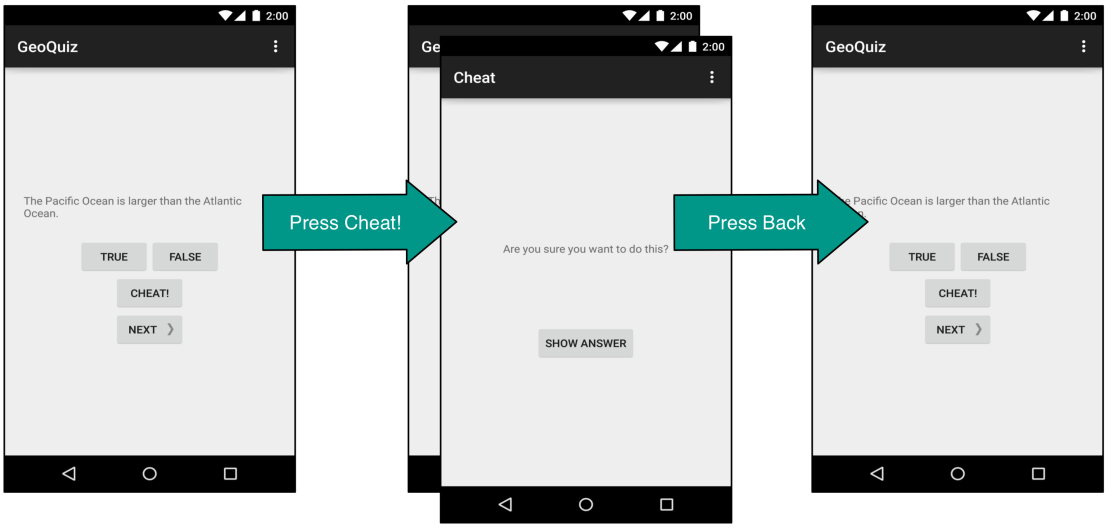
Listing 5.18 **`QuizActivity`** declared as launcher activity (`AndroidManifest.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  ... >

  ...

  <application
    ... >
    <activity
      android:name="com.bignerdranch.android.geoquiz.QuizActivity"
      android:label="@string/app_name" >
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity android:name=".CheatActivity" />
  </application>

</manifest>
```

After the instance of **`QuizActivity`** is on screen, the user can press the Cheat! button. When this happens, an instance of **`CheatActivity`** is started – on top of the **`QuizActivity`**. These activities exist in a stack (Figure 5.12).

Pressing the Back button in **`CheatActivity`** pops this instance off the stack, and the **`QuizActivity`** resumes its position at the top, as shown in Figure 5.12.

Figure 5.12 GeoQuiz's back stack



A call to **`Activity.finish()`** in **`CheatActivity`** would also pop the **`CheatActivity`** off the stack.

version that is later than the minimum SDK of Jelly Bean (API level 16)? When your app is installed and run on a Jelly Bean device, it will crash.

This used to be a testing nightmare. However, thanks to improvements in Android Lint, potential problems caused by calling newer code on older devices can be caught at compile time. If you use code from a higher version than your minimum SDK, Android Lint will report build errors.

Right now, all of GeoQuiz's simple code was introduced in API level 16 or earlier. Let's add some code from API level 21 (Lollipop) and see what happens.

Open `CheatActivity.java`. In the **`OnClickListener`** for the Show Answer button, add the following code to present a fancy circular animation while hiding the button:

Listing 6.2 Adding activity animation code (`CheatActivity.java`)

```
mShowAnswer.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (mAnswerIsTrue) {
            mAnswerTextView.setText(R.string.true_button);
        } else {
            mAnswerTextView.setText(R.string.false_button);
        }
        setAnswerShownResult(true);

        int cx = mShowAnswer.getWidth() / 2;
        int cy = mShowAnswer.getHeight() / 2;
        float radius = mShowAnswer.getWidth();
        Animator anim = ViewAnimationUtils
                .createCircularReveal(mShowAnswer, cx, cy, radius, 0);
        anim.addListener(new AnimatorListenerAdapter() {
            @Override
            public void onAnimationEnd(Animator animation) {
                super.onAnimationEnd(animation);
                mShowAnswer.setVisibility(View.INVISIBLE);
            }
        });
        anim.start();
    }
});
```

The **`createCircularReveal`** method creates an **`Animator`** from a few parameters. First, you specify the **`View`** that will be hidden or shown based on the animation. Next, a center position for the animation as well as the start radius and end radius of the animation. You are hiding the Show Answer button, so the radius moves from the width of the button to 0.

Before the newly created animation is started, you set a listener which allows you to know when the animation is complete. Once complete, you will show the answer and hide the button.

Finally, the animation is started and the circular reveal animation will begin. You will learn much more about animation in Chapter 30.

The **`ViewAnimationUtils`** and its **`createCircularReveal`** method were both added to the Android SDK in API level 21, so this code would crash on a device running a lower version than that.

After you enter the code in Listing 6.2, Android Lint should immediately present you with a warning that the code is not safe on your minimum SDK version. If you do not see a warning, you can manually trigger Lint by selecting Analyze → Inspect Code.... Because your build SDK version is API level 21, the compiler itself has no problem with this code. Android Lint, on the other hand, knows about your minimum SDK version and will complain loudly.

The error messages read something like Call requires API level 21 (Current min is 16). You can still run the code with this warning, but Lint knows it is not safe.

How do you get rid of these errors? One option is to raise the minimum SDK version to 21. However, raising the minimum SDK version is not really dealing with this compatibility problem as much as ducking it. If your app cannot be installed on API level 16 and older devices, then you no longer have a compatibility problem.

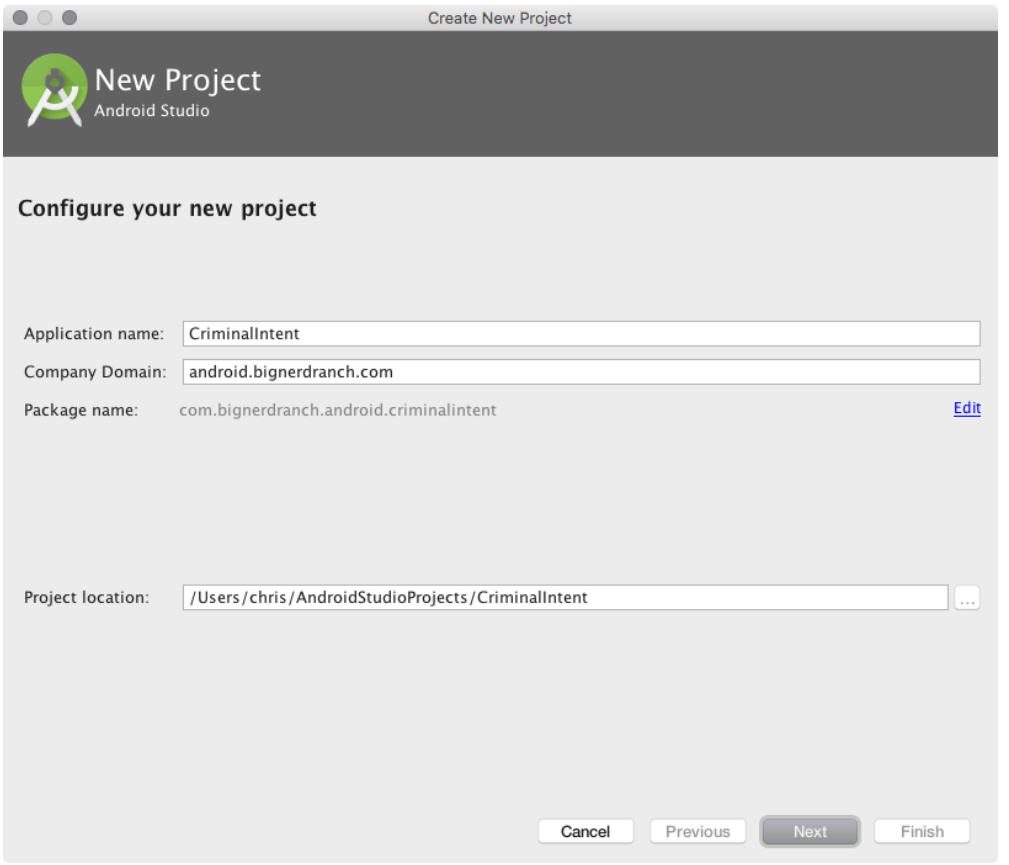
A better option is to wrap the higher API code in a conditional statement that checks the device's version of Android.

Listing 6.3 Checking the device's build version first

```
mShowAnswer.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (mAnswerIsTrue) {
            mAnswerTextView.setText(R.string.true_button);
        } else {
            mAnswerTextView.setText(R.string.false_button);
        }
        setAnswerShownResult(true);

        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
            int cx = mShowAnswer.getWidth() / 2;
            int cy = mShowAnswer.getHeight() / 2;
            float radius = mShowAnswer.getWidth();
            Animator anim = ViewAnimationUtils
                    .createCircularReveal(mShowAnswer, cx, cy, radius, 0);
            anim.addListener(new AnimatorListenerAdapter() {
                @Override
                public void onAnimationEnd(Animator animation) {
                    super.onAnimationEnd(animation);
                    mShowAnswer.setVisibility(View.INVISIBLE);
                }
            });
            anim.start();
        } else {
            mShowAnswer.setVisibility(View.INVISIBLE);
        }
    }
});
```

The `Build.VERSION.SDK_INT` constant is the device's version of Android. You then compare that version with the constant that stands for the Lollipop release. (Version codes are listed at `http://developer.android.com/reference/android/os/Build.VERSION_CODES.html`.)

Figure 7.7 Creating the CriminalIntent application



Click Next and specify a minimum SDK of API 16: Android 4.1. Also ensure that only the Phone and Tablet application type is checked.

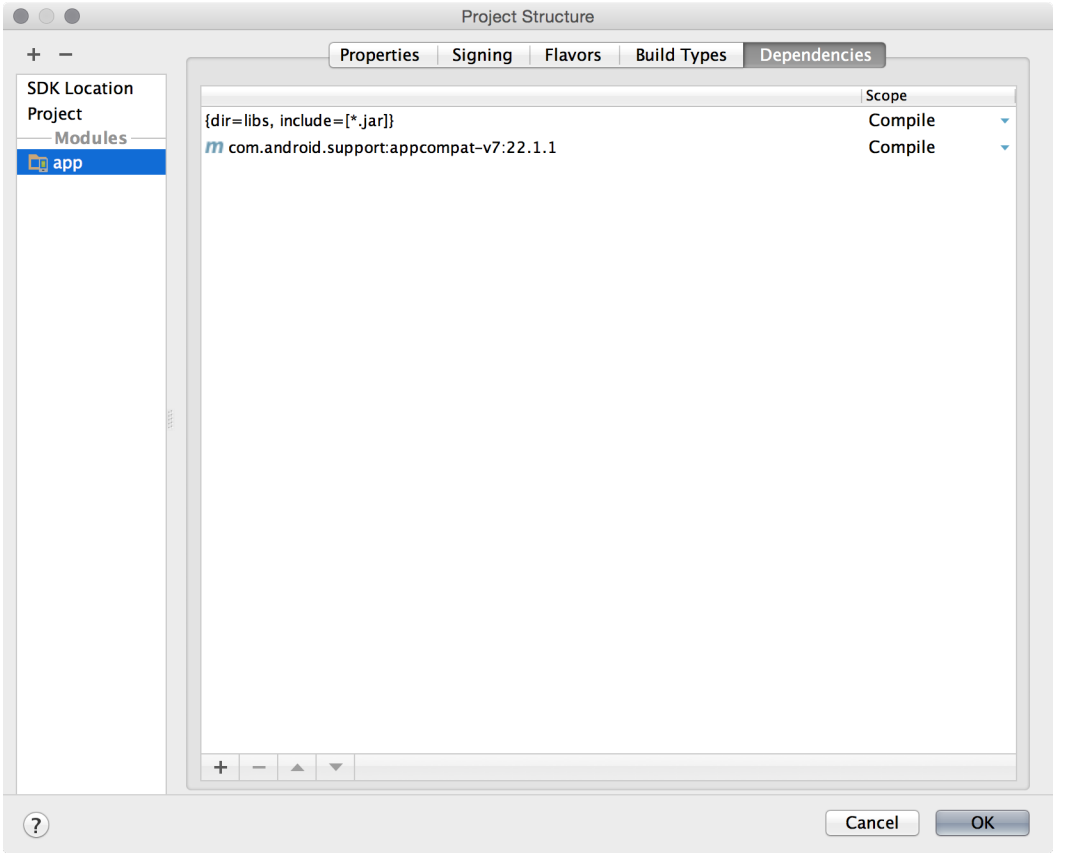
Click Next again to select the type of Activity to add. Choose Empty Activity and continue along in the wizard.

In the final step of the New Project wizard, name the activity **`CrimeActivity`** and click Finish (Figure 7.8).

Nobody can remember these incantations, though, so Android Studio maintains a list of common libraries for you. Navigate to the project structure for your project (File → Project Structure...).

Select the app module on the left and the Dependencies tab in the app module. The dependencies for the app module are listed here.

Figure 7.10 App dependencies



You may have additional dependencies specified here, such as the AppCompat dependency. If you have other dependencies, do not remove them. You will learn about the AppCompat library in Chapter 13.

Use the + button and choose Library dependency to add a new dependency (Figure 7.11). Choose the support-v4 library from the list and click OK.

Now that the support library is a dependency in the project, it is time to use it. In the project tool window, find and open `CrimeActivity.java`. Change **`CrimeActivity`**'s superclass to **`FragmentActivity`**.

Listing 7.3 Tweaking template code (`CrimeActivity.java`)

```
public class CrimeActivity extends AppCompatActivity FragmentActivity {

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

}
```

Before proceeding further with **`CrimeActivity`**, let's create the model layer for CriminalIntent by writing the **`Crime`** class.

Creating the Crime class

In the project tool window, right-click the `com.bignerdranch.android.criminalintent` package and select New → Java Class. Name the class **`Crime`** and click OK.

In `Crime.java`, add the following code:

Listing 7.4 Adding to **`Crime`** class (`Crime.java`)

```
public class Crime {

    private UUID mId;
    private String mTitle;

    public Crime() {
        // Generate unique identifier
        mId = UUID.randomUUID();
    }
}
```

Listing 8.3 Adding new widgets (`fragment_crime.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  >
  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/crime_title_label"
    style="?android:listSeparatorTextViewStyle"
    />
  <EditText android:id="@+id/crime_title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    android:hint="@string/crime_title_hint"
    />
  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/crime_details_label"
    style="?android:listSeparatorTextViewStyle"
    />
  <Button android:id="@+id/crime_date"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    />
  <CheckBox android:id="@+id/crime_solved"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="16dp"
    android:layout_marginRight="16dp"
    android:text="@string/crime_solved_label"
    />
</LinearLayout>
```

Notice that you did not give the **`Button`** an `android:text` attribute. This button will display the date of the **`Crime`** being displayed, and its text will be set in code.

Why display the date on a **`Button`**? You are preparing for the future. For now, a crime's date defaults to the current date and cannot be changed. In Chapter 12, you will wire up the button so that a press presents a **`DatePicker`** widget from which the user can set the date.

There are some new things in this layout to discuss, such as the `style` attribute and the margin attributes. But first let's get CriminalIntent up and running with the new widgets.

Open `res/values/strings.xml` and add the necessary string resources.

Listing 8.4 Adding string resources (`strings.xml`)

```
<resources>
    <string name="app_name">CriminalIntent</string>
    <string name="crime_title_hint">Enter a title for the crime.</string>
    <string name="crime_title_label">Title</string>
    <string name="crime_details_label">Details</string>
    <string name="crime_solved_label">Solved</string>
</resources>
```

Check for typos and save your files.

Wiring Widgets

Next, you are going to make the **`CheckBox`** display whether a **`Crime`** has been solved. You also need to update the **`Crime`**'s `mSolved` field when a user toggles the **`CheckBox`**.

For now, all the new **`Button`** needs to do is display the date in the **`Crime`**'s `mDate` field.

In `CrimeFragment.java`, add two new instance variables.

Listing 8.5 Adding widget instance variables (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    private Crime mCrime;
    private EditText mTitleField;
    private Button mDateButton;
    private CheckBox mSolvedCheckBox;

    @Override
    public void onCreate(Bundle savedInstanceState) {
    ...
```

Next, in **`onCreateView(…)`**, get a reference to the new button, set its text as the date of the crime, and disable it for now.

Listing 8.6 Setting **`Button`** text (`CrimeFragment.java`)

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
        Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_crime, parent, false);

    ...

    mTitleField.addTextChangedListener(new TextWatcher() {
        ...
    });

    mDateButton = (Button)v.findViewById(R.id.crime_date);
    mDateButton.setText(mCrime.getDate().toString());
    mDateButton.setEnabled(false);

    return v;
}
```

Passing data to DatePickerFragment

To get data into your **DatePickerFragment**, you are going to stash the date in **DatePickerFragment**'s arguments bundle, where the **DatePickerFragment** can access it.

Creating and setting fragment arguments is typically done in a **newInstance()** method that replaces the fragment constructor. In DatePickerFragment.java, add a **newInstance(Date)** method.

Listing 12.5 Adding a **newInstance(Date)** method (DatePickerFragment.java)

```
public class DatePickerFragment extends DialogFragment {

    private static final String ARG_DATE = "date";

    private DatePicker mDatePicker;

    public static DatePickerFragment newInstance(Date date) {
        Bundle args = new Bundle();
        args.putSerializable(ARG_DATE, date);

        DatePickerFragment fragment = new DatePickerFragment();
        fragment.setArguments(args);
        return fragment;
    }

    ...
}
```

In **CrimeFragment**, remove the call to the **DatePickerFragment** constructor and replace it with a call to **DatePickerFragment.newInstance(Date)**.

Listing 12.6 Adding call to **newInstance()** (CrimeFragment.java)

```
@Override
public View onCreateView(LayoutInflater inflater,
        ViewGroup parent, Bundle savedInstanceState) {
    ...

    mDateButton = (Button)v.findViewById(R.id.crime_date);
    mDateButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            FragmentManager manager = getFragmentManager();
            DatePickerFragment dialog = new DatePickerFragment();
            DatePickerFragment dialog = DatePickerFragment
                .newInstance(mCrime.getDate());
            dialog.show(manager, DIALOG_DATE);
        }
    });

    return v;
}
```

DatePickerFragment are destroyed and re-created by the OS. To create this relationship, you call the following **Fragment** method:

```
public void setTargetFragment(Fragment fragment, int requestCode)
```

This method accepts the fragment that will be the target and a request code just like the one you send in **startActivityForResult(…)**. The target fragment can use the request code later to identify which fragment is reporting back.

The **FragmentManager** keeps track of the target fragment and request code. You can retrieve them by calling **getTargetFragment()** and **getTargetRequestCode()** on the fragment that has set the target.

In `CrimeFragment.java`, create a constant for the request code and then make **CrimeFragment** the target fragment of the **DatePickerFragment** instance.

Listing 12.8 Setting target fragment (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {

    private static final String ARG_CRIME_ID = "crime_id";
    private static final String DIALOG_DATE = "DialogDate";

    private static final int REQUEST_DATE = 0;

    ...

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup parent,
            Bundle savedInstanceState) {
        ...

        mDateButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                FragmentManager manager = getFragmentManager();
                DatePickerFragment dialog = DatePickerFragment
                        .newInstance(mCrime.getDate());
                dialog.setTargetFragment(CrimeFragment.this, REQUEST_DATE);
                dialog.show(manager, DIALOG_DATE);
            }
        });

        return v;
    }

    ...
}
```

Sending data to the target fragment

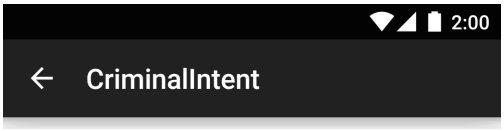
Now that you have a connection between **CrimeFragment** and **DatePickerFragment**, you need to send the date back to **CrimeFragment**. You are going to put the date on an **Intent** as an extra.

What method will you use to send this intent to the target fragment? Oddly enough, you will have **DatePickerFragment** pass it into **CrimeFragment.onActivityResult(int, int, Intent)**.

Activity.onActivityResult(…) is the method that the **ActivityManager** calls on the parent activity after the child activity dies. When dealing with activities, you do not call

Run the app and create a new crime. Notice the Up button, as shown in Figure 13.12. Pressing the Up button will take you up one level in CriminalIntent's hierarchy to `CrimeListActivity`.

Figure 13.12 `CrimePagerActivity`'s Up button



How hierarchical navigation works

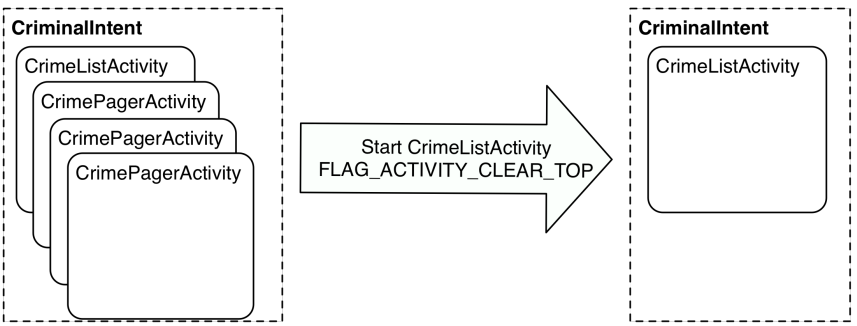
In CriminalIntent, navigating with the Back button and navigating with the Up button perform the same task. Pressing either of those from within the `CrimePagerActivity` will take the user back to the `CrimeListActivity`. Even though they accomplish the same result, behind the scenes they are doing very different things. This is important because, depending on the application, navigating up may pop the user back multiple activities in the back stack.

When the user navigates up from **`CrimePagerActivity`**, an intent like the following is created:

```
Intent intent = new Intent(this, CrimeListActivity.class);
intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
startActivity(intent);
finish();
```

`FLAG_ACTIVITY_CLEAR_TOP` tells Android to look for an existing instance of the activity in the stack, and if there is one, pop every other activity off the stack so that the activity being started will be top-most (Figure 13.13).

Figure 13.13 FLAG_ACTIVITY_CLEAR_TOP at work



An Alternative Action Item

In this section, you will use what you have learned about menu resources to add an action item that lets users show and hide the subtitle of **`CrimeListActivity`**'s toolbar.

In `res/menu/fragment_crime_list.xml`, add an action item that will read Show Subtitle and will appear in the toolbar if there is room.

Listing 16.5 Requesting external storage permission (`AndroidManifest.xml`)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bignerdranch.android.criminalintent" >

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18"
        />
    ...
```

The `maxSdkVersion` attribute makes it so that your app only asks for this permission on versions of Android that are older than API 19, Android KitKat.

Note that you are only asking to read external storage. There is also a `WRITE_EXTERNAL_STORAGE` permission, but you do not need it. You will not be writing anything to external storage: The camera app will do that for you.

Firing the intent

Now you are ready to fire the camera intent. The action you want is called `ACTION_IMAGE_CAPTURE`, and it is defined in the **`MediaStore`** class. **`MediaStore`** defines the public interfaces used in Android for interacting with common media – images, videos, and music. This includes the image capture intent, which fires up the camera.

By default, `ACTION_IMAGE_CAPTURE` will dutifully fire up the camera application and take a picture, but it will not be a full-resolution picture. Instead, it will take a small resolution thumbnail picture, and stick the whole thing inside the **`Intent`** object returned in **`onActivityResult(…)`**.

For a full-resolution output, you need to tell it where to save the image on the filesystem. This can be done by passing a **`Uri`** pointing to where you want to save the file in `MediaStore.EXTRA_OUTPUT`.

Write an implicit intent to ask for a new picture to be taken into the location saved in `mPhotoFile`. Add code to ensure that the button is disabled if there is no camera app, or if there is no location at which to save the photo. (To determine whether there is a camera app available, you will query **`PackageManager`** for activities that respond to your camera implicit intent. Querying the **`PackageManager`** is discussed in more detail in the section called “Checking for responding activities” in Chapter 15.)

Why Assets, Not Resources

Resources can store sounds. Just stash a file like `79_long_scream.wav` in `res/raw`, and you can get at it with the ID `R.raw.79_long_scream`. With sounds stored as resources, you can do all the usual resource things, like having different sounds for different orientations, languages, versions of Android, and so on.

However, BeatBox will have a lot of sounds: more than 20 different files. Dealing with them all one by one in the resources system would be cumbersome. It would be nice to just ship them all out in one big folder, but resources do not let you do this, nor do they allow you to give your resources anything other than a totally flat structure.

This is exactly what the assets system is great for. Assets are like a little file system that ships with your packaged application. With assets, you can use whatever folder structure you want. Since they give you this kind of organizational ability, assets are commonly used for loading graphics and sound in applications that have a lot of those things, like games.

Creating BeatBox

Time to get started. The first step is to create your BeatBox app. In Android Studio, select File → New Project... to create a new project. Call it BeatBox, and give it a company domain of `android.bignerdranch.com`. Use API 16 for your minimum SDK, and start with one Empty Activity called **`BeatBoxActivity`**. Leave the defaults as they are.

You will be using **`RecyclerView`** again, so open your project preferences and add the `com.android.support:recyclerview-v7` dependency.

Now, let's build out the basics of the app. The main screen will show a grid of buttons, each of which plays a sound. So, you will need two layout files: one for the grid and one for the buttons.

Create your layout file for the **`RecyclerView`** first. You will not need `res/layout/activity_beat_box.xml`, so go ahead and rename it `fragment_beat_box.xml`. Then fill it up like so:

Listing 18.1 Create main layout file (`res/layout/fragment_beat_box.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/fragment_beat_box_recycler_view"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  />
```

Next, create the layout for the buttons, `res/layout/list_item_sound.xml`.

Listing 18.2 Create sound layout (`res/layout/list_item_sound.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<Button
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/list_item_sound_button"
  android:layout_width="match_parent"
  android:layout_height="120dp"
  tools:text="Sound name"/>
```

19

Audio Playback with SoundPool

Now that you are ready to go with your assets, it is time to actually play all these `.wav` files. Android's audio APIs are fairly low level for the most part, but there is a tool practically tailor-made for the app you are writing: **`SoundPool`**.

`SoundPool` can load a large set of sounds into memory and control the maximum number of sounds that are playing back at any one time. So if your app's user gets a bit too excited and mashes all the buttons at the same time, it will not break your app or overtax your phone.

Ready? Time to get started.

Creating a SoundPool

The first step is to create a **`SoundPool`** object.

Listing 19.1 Creating a **`SoundPool`** (`BeatBox.java`)

```
public class BeatBox {
    private static final String TAG = "BeatBox";

    private static final String SOUNDS_FOLDER = "sample_sounds";
    private static final int MAX_SOUNDS = 5;

    private AssetManager mAssets;
    private List<Sound> mSounds = new ArrayList<>();
    private SoundPool mSoundPool;

    public BeatBox(Context context) {
        mAssets = context.getAssets();
        // This old constructor is deprecated, but we need it for
        // compatibility.
        mSoundPool = new SoundPool(MAX_SOUNDS, AudioManager.STREAM_MUSIC, 0);
        loadSounds();
    }

    ...
}
```

Lollipop introduced a new way of creating a **`SoundPool`** using a **`SoundPool.Builder`**. However, since **`SoundPool.Builder`** is not available on your minimum-supported API 16, you are using the older **`SoundPool(int, int, int)`** constructor instead.

Listing 19.3 Loading sounds into **SoundPool** (BeatBox.java)

```
    private void loadSounds() {
        ...
    }

    private void load(Sound sound) throws IOException {
        AssetFileDescriptor afd = mAssets.openFd(sound.getAssetPath());
        int soundId = mSoundPool.load(afd, 1);
        sound.setSoundId(soundId);
    }
}
```

Calling **mSoundPool.load(AssetFileDescriptor, int)** loads a file into your **SoundPool** for later playback. To keep track of the sound and play it back again (or unload it), **mSoundPool.load(…)** returns an `int` ID, which you stash in the `mSoundId` field you just defined. And since calling **openFd(String)** throws **IOException**, **load(Sound)** throws **IOException**, too.

Now load up all your sounds by calling **load(Sound)** inside **BeatBox.loadSounds()**.

Listing 19.4 Loading up all your sounds (BeatBox.java)

```
private void loadSounds() {
    ...

    for (String filename : soundNames) {
        try {
            String assetPath = SOUNDS_FOLDER + "/" + filename;
            Sound sound = new Sound(assetPath);
            load(sound);
            mSounds.add(sound);
        } catch (IOException ioe) {
            Log.e(TAG, "Could not load sound " + filename, ioe);
        }
    }
}
```

Run BeatBox to make sure that all the sounds loaded correctly. If they did not, you will see red exception logs in LogCat.

Playing Sounds

One last step: playing the sounds back. Add the **play(Sound)** method to **BeatBox**.

Listing 19.5 Playing sounds back (BeatBox.java)

```
    mSoundPool = new SoundPool(MAX_SOUNDS, AudioManager.STREAM_MUSIC, 0);
    loadSounds();
}

public void play(Sound sound) {
    Integer soundId = sound.getSoundId();
    if (soundId == null) {
        return;
    }
    mSoundPool.play(soundId, 1.0f, 1.0f, 1, 0, 1.0f);
}

public List<Sound> getSounds() {
    return mSounds;
}
```

Before playing your `soundId`, you check to make sure it is not `null`. This might happen if the **`Sound`** failed to load.

Once you are sure you have a non-`null` value, play the sound by calling **`SoundPool.play(int, float, float, int, int, float)`**. Those parameters are, respectively: the sound ID, volume on the left, volume on the right, priority (which is ignored), whether the audio should loop, and playback rate. For full volume and normal playback rate, you pass in `1.0`. Passing in `0` for the looping value says "do not loop." (You can pass in `-1` if you want it to loop forever. We speculate that this would be incredibly annoying.)

With that method written, you can play the sound each time one of the buttons is pressed.

Listing 19.6 Playing sound on button press (BeatBoxFragment.java)

```
private class SoundHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener {
    private Button mButton;
    private Sound mSound;

    public SoundHolder(LayoutInflater inflater, ViewGroup container) {
        super(inflater.inflate(R.layout.list_item_sound, parent, false));

        mButton = (Button)itemView.findViewById(R.id.list_item_sound_button);
        mButton.setOnClickListener(this);
    }

    public void bindSound(Sound sound) {
        mSound = sound;
        mButton.setText(mSound.getName());
    }

    @Override
    public void onClick(View v) {
        mBeatBox.play(mSound);
    }
}
```

Press a button, as shown in Figure 19.1, and you should hear a sound played.

Listing 20.1 Defining a few colors (`res/values/colors.xml`)

```
<resources>
  <color name="red">#F44336</color>
  <color name="dark_red">#C3352B</color>
  <color name="gray">#607D8B</color>
  <color name="soothing_blue">#0083BF</color>
  <color name="dark_blue">#005A8A</color>
</resources>
```

Color resources are a convenient way to specify color values in one place that you reference throughout your application.

Styles

Now, update the buttons in BeatBox with a *style*. A style is a set of attributes that you can apply to a widget.

Navigate to `res/values/styles.xml` and add a style named **`BeatBoxButton`**. (When you created BeatBox, your new project should have come with a built-in `styles.xml` file. If your project did not, create the file.)

Listing 20.2 Adding a style (`res/values/styles.xml`)

```
<resources>

  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
  </style>

  <style name="BeatBoxButton">
      <item name="android:background">@color/dark_blue</item>
  </style>

</resources>
```

Here, you create a style called **`BeatBoxButton`**. This style defines a single attribute, **`android:background`**, and sets it to a dark blue color. You can apply this style to as many widgets as you like and then update the attributes of all of those widgets in this one place.

Now that the style is defined, apply **`BeatBoxButton`** to your buttons in BeatBox.

Listing 20.3 Using a style (`res/layout/list_item_sound.xml`)

```
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  style="@style/BeatBoxButton"
  android:id="@+id/list_item_sound_button"
  android:layout_width="match_parent"
  android:layout_height="120dp"
  tools:text="Sound name"/>
```

Run BeatBox and you will see that all of your buttons now have a dark blue background color (Figure 20.2).

There is also an alternative inheritance naming style. You can specify a **`parent`** when declaring the style:

```
<style name="BeatBoxButton">
    <item name="android:background">@color/dark_blue</item>
</style>

<style name="StrongBeatBoxButton" parent="@style/BeatBoxButton">
    <item name="android:textStyle">bold</item>
</style>
```

Stick with the **`BeatBoxButton.Strong`** style in BeatBox.

Update `list_item_sound.xml` to use your newer, stronger style.

Listing 20.5 Using a bolder style (`res/layout/list_item_sound.xml`)

```
    <Button xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      style="@style/BeatBoxButton.Strong"
      android:id="@+id/list_item_sound_button"
      android:layout_width="match_parent"
      android:layout_height="120dp"
      tools:text="Sound name"/>
```

Run BeatBox and verify that your button text is indeed bold, as in Figure 20.3.

Figure 20.3 A bolder BeatBox

Figure 20.4 A dark BeatBox

Adding Theme Colors

With the base theme squared away, it is time to customize the attributes of BeatBox's **`AppTheme`**.

In the `styles.xml` file, define three attributes on your theme.

Listing 20.9 Setting theme attributes (`res/values/styles.xml`)

```
<resources>

  <style name="AppTheme" parent="Theme.AppCompat">
      <item name="colorPrimary">@color/red</item>
      <item name="colorPrimaryDark">@color/dark_red</item>
      <item name="colorAccent">@color/gray</item>
  </style>

  ...

</resources>
```

Here, you define three theme attributes. These theme attributes look similar to the style attributes that you set up earlier, but they specify different properties. Style attributes specify properties for an individual widget, such as the **`textStyle`** that you used to bold the button text. Theme attributes have a larger scope: they are properties that are set on the theme that any widget can access. For example, the toolbar will look at the **`colorPrimary`** attribute on the theme to set its background color.

Modifying Button Attributes

Earlier you customized the buttons in BeatBox by manually setting a **`style`** attribute in the `res/layout/list_item_sound.xml` file. If you have a more complex app, with buttons throughout many fragments, setting a **`style`** attribute on each and every button does not scale well. You can take your theme a step further by defining a style in your theme for every button in your app.

Before adding a button style to your theme, remove the **`style`** attribute from your `res/layout/list_item_sound.xml` file.

Listing 20.11 Be gone! We have a better way (`res/layout/list_item_sound.xml`)

```
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  style="@style/BeatBoxButton.Strong"
  android:id="@+id/list_item_sound_button"
  android:layout_width="match_parent"
  android:layout_height="120dp"
  tools:text="Sound name"/>
```

Run BeatBox and verify that your buttons are back to the old, bland look.

Navigate back to the **`Theme.Holo`** definition and look for a group of button attributes.

```
<style name="Theme.Holo">
  ...

  <!-- Button styles -->
  <item name="buttonStyle">@style/Widget.Holo.Button</item>

  <item name="buttonStyleSmall">@style/Widget.Holo.Button.Small</item>
  <item name="buttonStyleInset">@style/Widget.Holo.Button.Inset</item>

  ...
</style>
```

Notice the attribute named **`buttonStyle`**. This is the style of any normal button within your app.

The **`buttonStyle`** attribute points to a style resource rather than a value. When you updated the **`colorBackground`** attribute, you passed in a value: the color. In this case, **`buttonStyle`** should point to a style. Navigate to **`Widget.Holo.Button`** to see the button style.

```
<style name="Widget.Holo.Button" parent="Widget.Button">
  <item name="background">@drawable/btn_default_holo_dark</item>
  <item name="textAppearance">?attr/textAppearanceMedium</item>
  <item name="textColor">@color/primary_text_holo_dark</item>
  <item name="minHeight">48dip</item>
  <item name="minWidth">64dip</item>
</style>
```

Every **`Button`** that you use in BeatBox is given these attributes.

Duplicate what happens in Android's own theme in BeatBox. Change the parent of **`BeatBoxButton`** to inherit from the existing button style. Also, remove your **`BeatBoxButton.Strong`** style from earlier.

Listing 20.12 Creating a button style (`res/values/styles.xml`)

```
<resources>

  <style name="AppTheme" parent="Theme.AppCompat">
    <item name="colorPrimary">@color/red</item>
    <item name="colorPrimaryDark">@color/dark_red</item>
    <item name="colorAccent">@color/gray</item>

    <item name="android:colorBackground">@color/soothing_blue</item>
  </style>

  <style name="BeatBoxButton" parent="android:style/Widget.Holo.Button">
    <item name="android:background">@color/dark_blue</item>
  </style>

  <style name="BeatBoxButton.Strong">
    <item name="android:textStyle">bold</item>
  </style>

</resources>
```

You specified a parent of **`android:style/Widget.Holo.Button`**. You want your button to inherit all of the properties that a normal button would receive and then selectively modify attributes.

If you do not specify a parent theme for **`BeatBoxButton`**, you will notice that your buttons devolve into something that does not look like a button at all. Properties you expect to see, such as the text centered in the button, will be lost.

Now that you have fully defined **`BeatBoxButton`**, it is time to use it. Look back at the **`buttonStyle`** attribute that you found earlier when digging through Android's themes. Duplicate this attribute in your own theme.

Listing 20.13 Using the **`BeatBoxButton`** style (`res/values/styles.xml`)

```
<resources>

  <style name="AppTheme" parent="Theme.AppCompat">
    <item name="colorPrimary">@color/red</item>
    <item name="colorPrimaryDark">@color/dark_red</item>
    <item name="colorAccent">@color/gray</item>

    <item name="android:colorBackground">@color/soothing_blue</item>
    <item name="android:buttonStyle">@style/BeatBoxButton</item>
  </style>

  <style name="BeatBoxButton" parent="android:style/Widget.Holo.Button">
    <item name="android:background">@color/dark_blue</item>
  </style>

</resources>
```

You are now overriding the **`buttonStyle`** attribute from Android's themes and substituting your own style: **`BeatBoxButton`**.

In your own applications, it is a good idea to follow the same convention. Specify your theme parent in the name of your theme if you are inheriting from one of your own themes. If you inherit from a style or theme in the Android OS, explicitly specify the **`parent`** attribute.

For the More Curious: Accessing Theme Attributes

Once attributes are declared in your theme, you can access them in XML or in code.

To access a theme attribute in XML, you use the notation that you saw on the **`divider`** attribute in Chapter 17. When referencing a concrete value in XML, such as a color, you use the `@` notation. `@color/gray` points to a specific resource.

When referencing a resource in the theme, you use the `?` notation.

```
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/list_item_sound_button"
  android:layout_width="match_parent"
  android:layout_height="120dp"
  android:background="?attr/colorAccent"
  tools:text="Sound name"/>
```

The `?` notation says to use the resource that the **`colorAccent`** attribute on your theme points to. In your case, this is the gray color that you defined in your `colors.xml` file.

You can also use theme attributes in code, although it is much more verbose.

```
Resources.Theme theme = getActivity().getTheme();
int[] attrsToFetch = { R.attr.colorAccent };
TypedArray a = theme.obtainStyledAttributes(R.style.AppTheme, attrsToFetch);
int accentColor = a.getInt(0, 0);
a.recycle();
```

On the **`Theme`** object, you ask to resolve the attribute `R.attr.colorAccent` that is defined in your **`AppTheme`**: `R.style.AppTheme`. This call returns a **`TypedArray`**, which holds your data. On the **`TypedArray`**, you ask for an `int` value to pull out the accent color. From here, you can use that color to change the background of a button, for example.

The toolbar and buttons in BeatBox are doing exactly this to style themselves based on your theme attributes.

Challenge: An Appropriate Base Theme

When you created **`BeatBoxButton`**, you inherited attributes from **`android:style/Widget.Holo.Button`**. While inheriting from the Holo theme works, you are not taking advantage of the latest theme available.

In Android 5.0 (Lollipop), the material theme was released. This theme makes changes to various properties of your button, including the font size. It is a good idea to take advantage of this new look on a device that supports the material theme.

Your challenge is to create a resource-qualified version of your `styles.xml` file: `values-v21/styles.xml`. Next, create two versions of your **`BeatBoxButton`** style. One should inherit attributes from **`Widget.Holo.Button`**, and the other from **`Widget.Material.Button`**.

Setting Up NerdLauncher

Create a new Android application project named NerdLauncher. Select Phone and Tablet as the form factor and API 16: Android 4.1 (Jelly Bean) as the minimum SDK. Create an empty activity named **NerdLauncherActivity**.

NerdLauncherActivity will host a single fragment and in turn should be a subclass of **SingleFragmentActivity**. Copy SingleFragmentActivity.java and activity_fragment.xml into your NerdLauncher from the CriminalIntent project.

Open NerdLauncherActivity.java and change **NerdLauncherActivity**'s superclass to **SingleFragmentActivity**. Remove the template's code and override **createFragment()** to return a **NerdLauncherFragment**. (Bear with the error caused by the return line in **createFragment()**. This will be fixed in a moment when you create the **NerdLauncherFragment** class.)

Listing 22.1 Another **SingleFragmentActivity** (NerdLauncherActivity.java)

```
public class NerdLauncherActivity extends SingleFragmentActivityAppCompatActivity {

    @Override
    protected Fragment createFragment() {
        return NerdLauncherFragment.newInstance();
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        /* Auto-generated template code... */
    }
}
```

NerdLauncherFragment will display a list of application names in a **RecyclerView**. Add the RecyclerView library as a dependency, as you did in Chapter 9.

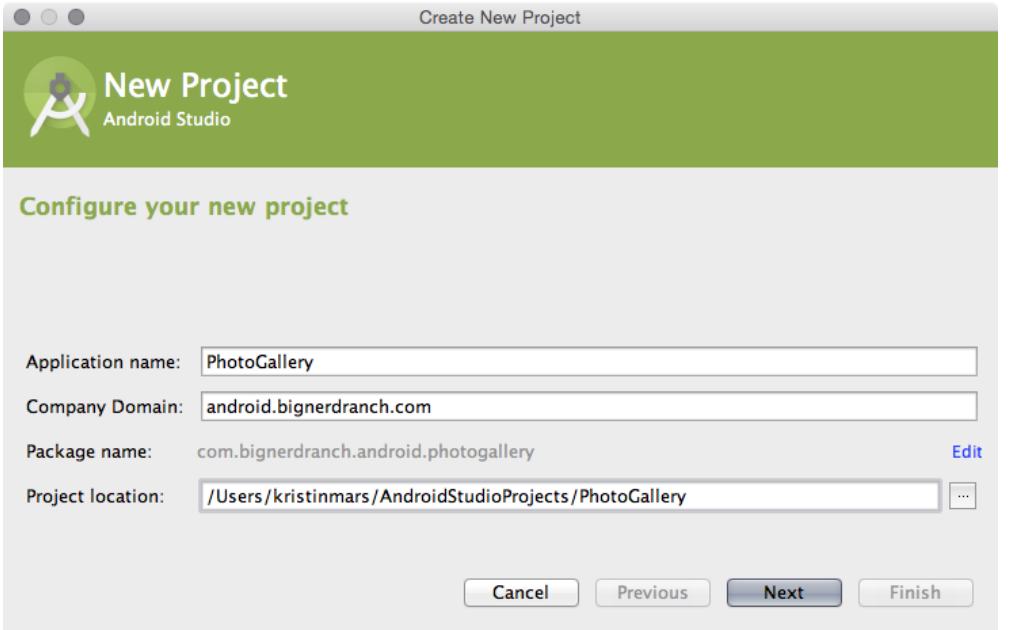
Rename layout/activity_nerd_launcher.xml to layout/fragment_nerd_launcher.xml to create a layout for the fragment. Replace its contents with the **RecyclerView** shown in Figure 22.2.

Figure 22.2 Create **NerdLauncherFragment** layout (layout/fragment_nerd_launcher.xml)

android.support.v7.widget.RecyclerView

```
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/fragment_nerd_launcher_recycler_view"
android:layout_width="match_parent"
android:layout_height="match_parent"
```

Figure 23.3 Creating PhotoGallery



Click Next. When prompted, check Phone and Tablet as the target form factor and choose API 16: Android 4.1 (Jelly Bean) from the Minimum SDK dropdown.

Then have the wizard create an empty activity named **`PhotoGalleryActivity`**.

PhotoGallery will follow the same architecture you have been using so far. **`PhotoGalleryActivity`** will be a **`SingleFragmentActivity`** subclass and its view will be the container view defined in `activity_fragment.xml`. This activity will host a fragment – in particular, an instance of **`PhotoGalleryFragment`**, which you will create shortly.

Copy `SingleFragmentActivity.java` and `activity_fragment.xml` into your project from a previous project.

In `PhotoGalleryActivity.java`, set up **`PhotoGalleryActivity`** as a **`SingleFragmentActivity`** by deleting the code that the template generated and replacing it with an implementation of **`createFragment()`**. Have **`createFragment()`** return an instance of **`PhotoGalleryFragment`**. (Bear with the error that this code will cause for the moment. It will go away after you create the **`PhotoGalleryFragment`** class.)

Listing 23.1 Activity setup (PhotoGalleryActivity.java)

```
public class PhotoGalleryActivity extends Activity SingleFragmentActivity {

    @Override
    public Fragment createFragment() {
        return PhotoGalleryFragment.newInstance();
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        /* Auto-generated template code... */
    }

}
```

PhotoGallery will display its results in a **RecyclerView**, using the built-in **GridLayoutManager** to arrange the items in a grid.

First, add the RecyclerView library as a dependency, as you did in Chapter 9. Open the Project Structure window and select the app module on the left. Select the Dependencies tab and click the + button. Select Library dependency from the drop-down menu that appears. Find and select the recyclerview-v7 library and click OK.

Rename layout/activity_photo_gallery.xml to layout/fragment_photo_gallery.xml to create a layout for the fragment. Then replace its contents with the **RecyclerView** shown in Figure 23.4.

Figure 23.4 A RecyclerView (layout/fragment_photo_gallery.xml)

android.support.v7.widget.RecyclerView

```
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/fragment_photo_gallery_recycler_view"
android:layout_width="match_parent"
android:layout_height="match_parent"
```

Finally, create the **PhotoGalleryFragment** class. Retain the fragment, inflate the layout you just created, and initialize a member variable referencing the **RecyclerView** (Listing 23.2).

Next, you will need a placeholder image for each **`ImageView`** to display until you download an image to replace it. Find `bill_up_close.png` in the solutions file and put it in `res/drawable`. (See the section called “Adding an Icon” in Chapter 2 for more on the solutions.)

Update **`PhotoAdapter`**’s **`onBindViewHolder()`** to set the placeholder image as the **`ImageView`**’s **`Drawable`**.

Listing 24.3 Binding default image (`PhotoGalleryFragment.java`)

```
public class PhotoGalleryFragment extends Fragment {
  ...

  private class PhotoAdapter extends RecyclerView.Adapter<PhotoHolder> {
        ...

        @Override
        public void onBindViewHolder(PhotoHolder photoHolder, int position) {
          GalleryItem galleryItem = mGalleryItems.get(position);
          photoHolder.bindGalleryItem(galleryItem);
          Drawable placeholder = getResources().getDrawable(R.drawable.bill_up_close);
          photoHolder.bindDrawable(placeholder);
        }

        ...
    }

  ...
}
```

Run PhotoGallery, and you should see an array of close-up Bills, as in Figure 24.2.

Figure 24.2 A Billsplosion

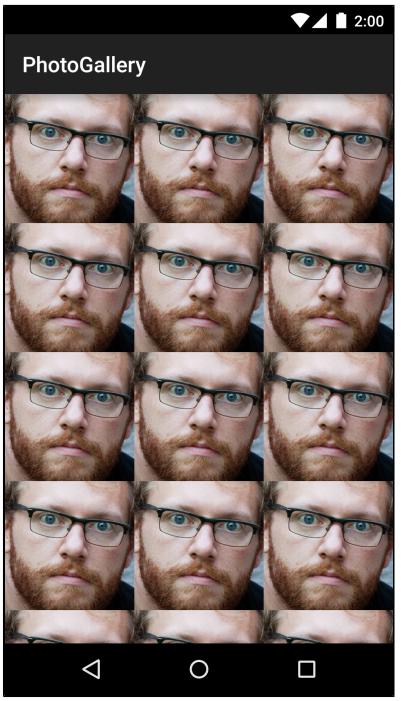
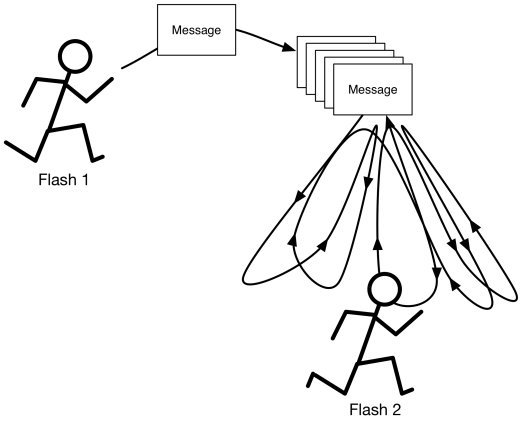


Figure 24.3 Flash dance



A message loop consists of a thread and a *looper*. The **`Looper`** is the object that manages a thread's message queue. The main thread is a message loop and has a looper. Everything your main thread does is performed by its looper, which grabs messages off of its message queue and performs the task they specify.

You are going to create a background thread that is also a message loop. You will use a class called **`HandlerThread`** that prepares a **`Looper`** for you.

Assembling a Background Thread

Create a new class called **`ThumbnailDownloader`** that extends **`HandlerThread`**. Then give it a constructor, a stub implementation of a method called **`queueThumbnail()`**, and an override of the **`quit()`** that signals when your thread has quit (you will need this later) (Listing 24.4).

Listing 24.4 Initial thread code (`ThumbnailDownloader.java`)

```
public class ThumbnailDownloader<T> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";

    private boolean mHasQuit = false;

    public ThumbnailDownloader() {
        super(TAG);
    }

    @Override
    public boolean quit() {
        mHasQuit = true;
        return super.quit();
    }

    public void queueThumbnail(T target, String url) {
        Log.i(TAG, "Got a URL: " + url);
    }
}
```

Notice you gave the class a single generic argument, <T>. Your **`ThumbnailDownloader`**'s user, **`PhotoGalleryFragment`** in this case, will need to use some object to identify each download and to

determine which UI element to update with the image once it is downloaded. Rather than locking the user into a specific type of object as the identifier, using a generic makes the implementation more flexible.

The **queueThumbnail()** method expects an object of type **T** to use as the identifier for the download and a **String** containing the URL to download. This is the method you will have **PhotoAdapter** call in its **onBindViewHolder(…)** implementation.

Open PhotoGalleryFragment.java. Give **PhotoGalleryFragment** a **ThumbnailDownloader** member variable. In **onCreate(…)**, create the thread and start it. Override **onDestroy()** to quit the thread.

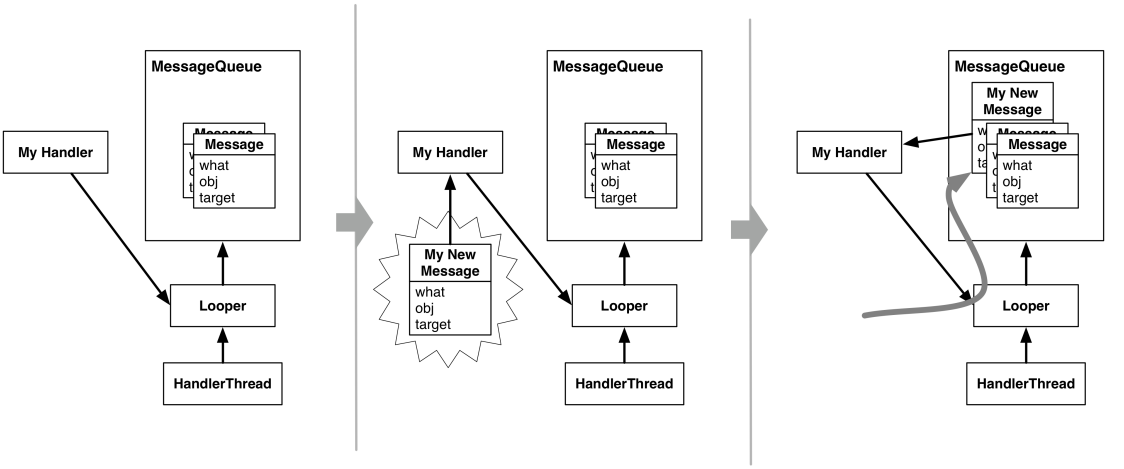
Listing 24.5 Creating **ThumbnailDownloader** (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {

    private static final String TAG = "PhotoGalleryFragment";

    private RecyclerView mPhotoRecyclerView;
    private List<GalleryItem> mItems = new ArrayList<>();
    private ThumbnailDownloader<PhotoHolder> mThumbnailDownloader;

    ...

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        new FetchItemsTask().execute();

        mThumbnailDownloader = new ThumbnailDownloader<>();
        mThumbnailDownloader.start();
        mThumbnailDownloader.getLooper();
        Log.i(TAG, "Background thread started");
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        ...
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        mThumbnailDownloader.quit();
        Log.i(TAG, "Background thread destroyed");
    }
    ...
}
```

You can specify any type for **ThumbnailDownloader**'s generic argument. However, recall that this argument specifies the type of the object that will be used as the identifier for your download. In this case, the **PhotoHolder** makes for a convenient identifier as it is also the target where the downloaded images will eventually go.

A couple of safety notes. One: notice that you call **getLooper()** after calling **start()** on your **ThumbnailDownloader** (you will learn more about the **Looper** in a moment). This is a way to ensure

Figure 24.6 Creating a **`Message`** and sending it



In this case, your implementation of **`handleMessage(…)`** will use **`FlickrFetchr`** to download bytes from the URL and then turn these bytes into a bitmap.

First, add the constant and member variables as shown in Listing 24.7.

Listing 24.7 Adding constant and member variables (`ThumbnailDownloader.java`)

```
public class ThumbnailDownloader<T> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;

    private boolean mHasQuit = false;
    private Handler mRequestHandler;
    private ConcurrentMap<T,String> mRequestMap = new ConcurrentHashMap<>();

    ...
}
```

`MESSAGE_DOWNLOAD` will be used to identify messages as download requests. (**`ThumbnailDownloader`** will set this as the `what` on any new download messages it creates.)

The newly added `mRequestHandler` will store a reference to the **`Handler`** responsible for queueing download requests as messages onto the **`ThumbnailDownloader`** background thread. This handler will also be in charge of processing download request messages when they are pulled off the queue.

The `mRequestMap` variable is a **`ConcurrentHashMap`**. A **`ConcurrentHashMap`** is a thread-safe version of **`HashMap`**. Here, using a download request's identifying object of type **`T`** as a key, you can store and retrieve the URL associated with a particular request. (In this case, the identifying object is a **`PhotoHolder`**, so the request response can be easily routed back to the UI element where the downloaded image should be placed.)

Next, add code to **`queueThumbnail(…)`** to update `mRequestMap` and to post a new message to the background thread's message queue.

Listing 24.8 Obtaining and sending a message (`ThumbnailDownloader.java`)

```
public class ThumbnailDownloader<T> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;

    private boolean mHasQuit = false;
    private Handler mRequestHandler;
    private ConcurrentMap<T,String> mRequestMap = new ConcurrentHashMap<>();

    public ThumbnailDownloader() {
        super(TAG);
    }

    @Override
    public boolean quit() {
        mHasQuit = true;
        return super.quit();
    }

    public void queueThumbnail(T target, String url) {
        Log.i(TAG, "Got a URL: " + url);

        if (url == null) {
            mRequestMap.remove(target);
        } else {
            mRequestMap.put(target, url);
            mRequestHandler.obtainMessage(MESSAGE_DOWNLOAD, target)
                    .sendToTarget();
        }
    }
}
```

You obtain a message directly from `mRequestHandler`, which automatically sets the new **`Message`** object's `target` field to `mRequestHandler`. This means `mRequestHandler` will be in charge of processing the message when it is pulled off the message queue. The message's `what` field is set to `MESSAGE_DOWNLOAD`. Its `obj` field is set to the `T target` value (a **`PhotoHolder`** in this case) that is passed to **`queueThumbnail(…)`**.

The new message represents a download request for the specified `T target` (a **`PhotoHolder`** from the **`RecyclerView`**). Recall that **`PhotoGalleryFragment`**'s **`RecyclerView`**'s adapter implementation calls **`queueThumbnail(…)`** from **`onBindViewHolder(…)`**, passing along the **`PhotoHolder`** the image is being downloaded for and the URL location of the image to download.

Notice that the message itself does not include the URL. Instead you update **`mRequestMap`** with a mapping between the request identifier (**`PhotoHolder`**) and the URL for the request. Later you will pull the URL from `mRequestMap` to ensure that you are always downloading the most recently requested URL for a given **`PhotoHolder`** instance. (This is important because **`ViewHolder`** objects in **`RecyclerView`**s are recycled and reused.)

Finally, initialize `mRequestHandler` and define what that **`Handler`** will do when downloaded messages are pulled off the queue and passed to it.

Listing 24.9 Handling a message (ThumbnailDownloader.java)

```
public class ThumbnailDownloader<T> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;

    private boolean mHasQuit = false;
    private Handler mRequestHandler;
    private ConcurrentMap<T,String> mRequestMap = new ConcurrentHashMap<>();

    public ThumbnailDownloader() {
        super(TAG);
    }

    @Override
    protected void onLooperPrepared() {
        mRequestHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MESSAGE_DOWNLOAD) {
                    T target = (T) msg.obj;
                    Log.i(TAG, "Got a request for URL: " + mRequestMap.get(target));
                    handleRequest(target);
                }
            }
        };
    }

    @Override
    public boolean quit() {
        mHasQuit = true;
        return super.quit();
    }

    public void queueThumbnail(T target, String url) {
        ...
    }

    private void handleRequest(final T target) {
        try {
            final String url = mRequestMap.get(target);

            if (url == null) {
                return;
            }

            byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);
            final Bitmap bitmap = BitmapFactory
                    .decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
            Log.i(TAG, "Bitmap created");

        } catch (IOException ioe) {
            Log.e(TAG, "Error downloading image", ioe);
        }
    }
}
```

You implemented **`Handler.handleMessage(…)`** in your **`Handler`** subclass within **`onLooperPrepared()`**. **`HandlerThread.onLooperPrepared()`** is called before the **`Looper`** checks the queue for the first time. This makes it a good place to create your **`Handler`** implementation.

Within **`Handler.handleMessage(…)`**, you check the message type, retrieve the `obj` value (which will be of type **`T`** and serves as the identifier for the request), and then pass it to **`handleRequest(…)`**. (Recall that **`Handler.handleMessage(…)`** will get called when a download message is pulled off the queue and ready to be processed.)

The **`handleRequest()`** method is a helper method where the downloading happens. Here you check for the existence of a URL. Then you pass the URL to a new instance of your old friend **`FlickrFetchr`**. In particular, you use the **`FlickrFetchr.getUrlBytes(…)`** method that you created with such foresight in the last chapter.

Finally, you use **`BitmapFactory`** to construct a bitmap with the array of bytes returned from **`getUrlBytes(…)`**.

Run PhotoGallery and check LogCat for your confirming log statements.

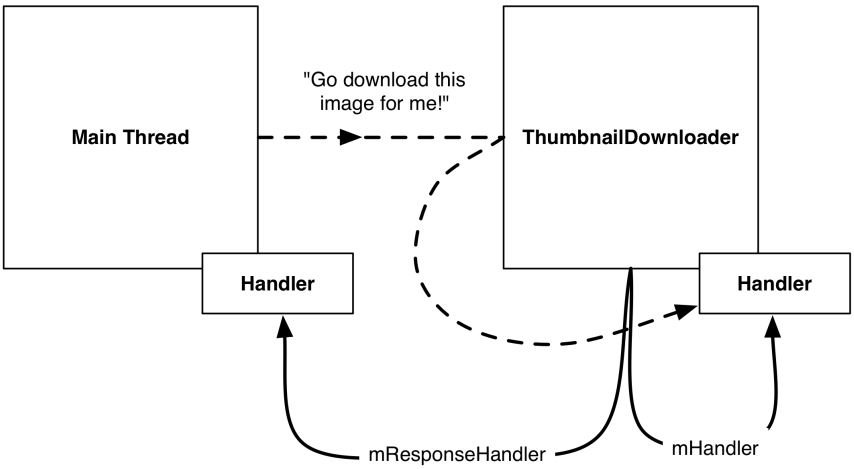
Of course, the request will not be completely handled until you set the bitmap on the **`PhotoHolder`** that originally came from **`PhotoAdapter`**. However, this is UI work, so it must be done on the main thread.

Everything you have seen so far uses handlers and messages on a single thread – **`ThumbnailDownloader`** putting messages in **`ThumbnailDownloader`**'s own inbox. In the next section, you will see how **`ThumbnailDownloader`** can use a **`Handler`** to post requests to a separate thread (namely, the main thread).

Passing handlers

So far you are able to schedule work on the background thread from the main thread using **`ThumbnailDownloader`**'s `mRequestHandler`. This flow is shown in Figure 24.7.

Figure 24.7 Scheduling work on **`ThumbnailDownloader`** from the main thread



You can also schedule work on the main thread from the background thread using a **`Handler`** attached to the main thread. This flow looks like Figure 24.8.

Listing 24.10 Handling a message (ThumbnailDownloader.java)

```
public class ThumbnailDownloader<T> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;

    private boolean mHasQuit = false;
    private Handler mRequestHandler;
    private ConcurrentMap<T,String> mRequestMap = new ConcurrentHashMap<>();
    private Handler mResponseHandler;
    private ThumbnailDownloadListener<T> mThumbnailDownloadListener;

    public interface ThumbnailDownloadListener<T> {
        void onThumbnailDownloaded(T target, Bitmap thumbnail);
    }

    public void setThumbnailDownloadListener(ThumbnailDownloadListener<T> listener) {
        mThumbnailDownloadListener = listener;
    }

    public ThumbnailDownloader(Handler responseHandler) {
        super(TAG);
        mResponseHandler = responseHandler;
    }

    ...
}
```

The **onThumbnailDownloaded(…)** method defined in your new **ThumbnailDownloadListener** interface will eventually be called when an image has been fully downloaded and is ready to be added to the UI. Using this listener delegates the responsibility of what to do with the downloaded image to a class other than **ThumbnailDownloader** (in this case, to **PhotoGalleryFragment**). Doing so separates the downloading task from the UI updating task (putting the images into **ImageView**s), so that **ThumbnailDownloader** could be used for downloading into other kinds of **View** objects as needed.

Next, modify **PhotoGalleryFragment** to pass a **Handler** attached to the main thread to **ThumbnailDownloader**. Also, set a **ThumbnailDownloadListener** to handle the downloaded image once it is complete.

Listing 24.11 Hooking up to response **Handler** (PhotoGalleryFragment.java)

```
...

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
    new FetchItemsTask().execute();

    Handler responseHandler = new Handler();
    mThumbnailDownloader = new ThumbnailDownloader<>(responseHandler);
    mThumbnailDownloader.setThumbnailDownloadListener(
        new ThumbnailDownloader.ThumbnailDownloadListener<PhotoHolder>() {
            @Override
            public void onThumbnailDownloaded(PhotoHolder photoHolder, Bitmap bitmap) {
                Drawable drawable = new BitmapDrawable(getResources(), bitmap);
                photoHolder.bindDrawable(drawable);
            }
        }
    );
    mThumbnailDownloader.start();
    mThumbnailDownloader.getLooper();
    Log.i(TAG, "Background thread started");
}

...
```

Remember that by default, the **Handler** will attach itself to the **Looper** for the current thread. Because this **Handler** is created in **onCreate(…)**, it will be attached to the main thread's **Looper**.

Now **ThumbnailDownloader** has access via `mResponseHandler` to a **Handler** that is tied to the main thread's **Looper**. It also has your **ThumbnailDownloadListener** to do the UI work with the returning **Bitmap**s. Specifically, the **onThumbnailDownloaded** implementation sets the **Drawable** of the originally requested **PhotoHolder** to the newly downloaded **Bitmap**.

You could send a custom **Message** back to the main thread requesting to add the image to the UI, similar to how you queued a request on the background thread to download the image. However, this would require another subclass of **Handler**, with an override of **handleMessage(…)**.

Instead, let's use another handy **Handler** method – **post(Runnable)**.

Handler.post(Runnable) is a convenience method for posting **Message**s that look like this:

```
Runnable myRunnable = new Runnable() {
    @Override
    public void run() {
        /* Your code here */
    }
};
Message m = mHandler.obtainMessage();
m.callback = myRunnable;
```

When a **Message** has its `callback` field set, it is not routed to its `target` **Handler** when pulled off the message queue. Instead, the **run()** method of the **Runnable** stored in `callback` is executed directly.

In **ThumbnailDownloader.handleRequest()**, add the following code.

Listing 24.12 Downloading and displaying images (ThumbnailDownloader.java)

```
public class ThumbnailDownloader<T> extends HandlerThread {

    ...
    private Handler mResponseHandler;
    private ThumbnailDownloadListener<T> mThumbnailDownloadListener;

    ...

    private void handleRequest(final T target) {
        try {
            final String url = mRequestMap.get(target);

            if (url == null) {
                return;
            }

            byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);
            final Bitmap bitmap = BitmapFactory
                    .decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
            Log.i(TAG, "Bitmap created");

            mResponseHandler.post(new Runnable() {
                public void run() {
                    if (mRequestMap.get(target) != url ||
                            mHasQuit) {
                        return;
                    }

                    mRequestMap.remove(target);
                    mThumbnailDownloadListener.onThumbnailDownloaded(target, bitmap);
                }
            });

        } catch (IOException ioe) {
            Log.e(TAG, "Error downloading image", ioe);
        }
    }
}
```

Because `mResponseHandler` is associated with the main thread's **`Looper`**, all of the code inside of **`run()`** will be executed on the main thread.

So what does this code do? First, you double-check the `requestMap`. This is necessary because the **`RecyclerView`** recycles its views. By the time **`ThumbnailDownloader`** finishes downloading the **`Bitmap`**, **`RecyclerView`** may have recycled the **`PhotoHolder`** and requested a different URL for it. This check ensures that each **`PhotoHolder`** gets the correct image, even if another request has been made in the meantime.

Next, you check `mHasQuit`. If **`ThumbnailDownloader`** has already quit, it may be unsafe to run any callbacks.

Finally, you remove the **`PhotoHolder`**-URL mapping from the `requestMap` and set the bitmap on the target **`PhotoHolder`**.

Before running PhotoGallery and seeing your hard-won images, there is one last danger you need to account for. If the user rotates the screen, **ThumbnailDownloader** may be hanging on to invalid **PhotoHolder**s. Bad things will happen if the corresponding **ImageView**s get pressed.

Write the following method to clean all the requests out of your queue.

Listing 24.13 Adding cleanup method (ThumbnailDownloader.java)

```
public class ThumbnailDownloader<T> extends HandlerThread {

    ...

    public void queueThumbnail(T target, String url) {
        ...
    }

    public void clearQueue() {
        mRequestHandler.removeMessages(MESSAGE_DOWNLOAD);
    }

    private void handleRequest(final T target) {
        ...
    }
}
```

Then clean out your downloader in **PhotoGalleryFragment** when your view is destroyed.

Listing 24.14 Calling cleanup method (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {

  ...

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
      ...
  }

  @Override
  public void onDestroyView() {
      super.onDestroyView();
      mThumbnailDownloader.clearQueue();
  }

  @Override
  public void onDestroy() {
      ...
  }

  ...
}
```

With that, your work for this chapter is complete. Run PhotoGallery. Scroll around to see images dynamically loading.

PhotoGallery has achieved its basic goal of displaying images from Flickr. In the next few chapters, you will add more functionality, like searching for photos and opening each photo's Flickr page in a web view.

```
Friend someNewFriend = ...;
NewFriendAddedEvent event = new NewFriendAddedEvent(someNewFriend);
eventBus.onNext(event);
```

And subscribe to events on it:

```
eventBus.subscribe(new Action1<Object>() {
    @Override
    public void call(Object event) {
        if (event instanceof NewFriendAddedEvent) {
            Friend newFriend = ((NewFriendAddedEvent)event).getFriend();
            // Update the UI
        }
    }
})
```

The advantage of RxJava's solution is that your `eventBus` is now also an **`Observable`**, RxJava's representation of a stream of events. That means that you get to use all of RxJava's various event manipulation tools. If that piques your interest, check out the wiki on RxJava's project page: `https://github.com/ReactiveX/RxJava/wiki`.

For the More Curious: Detecting the Visibility of Your Fragment

When you reflect on your PhotoGallery implementation, you may notice that you used the global broadcast mechanism to broadcast the `SHOW_NOTIFICATION` intent. However, you locked the receiving of that broadcast to items local to your app progress by using custom permissions. You may find yourself asking, "Why am I using a global mechanism if I am just communicating things in my own app? Why not a local mechanism instead?"

This is because you were specifically trying to solve the problem of knowing whether or not **`PhotoGalleryFragment`** was visible. The combination of ordered broadcasts, standalone receivers, and dynamically registered receivers you implemented gets the job done. There is not a more straightforward way to do this in Android.

More specifically, **`LocalBroadcastManager`** would not work for PhotoGallery's notification broadcast and visible fragment detection, for two main reasons.

First, **`LocalBroadcastManager`** does not support ordered broadcasts (though it does provide a blocking way to broadcast, namely **`sendBroadcastSync(Intent intent)`**). This will not work for PhotoGallery because you need to force **`NotificationReceiver`** to run last in the chain.

Second, **`sendBroadcastSync(Intent intent)`** does not support sending and receiving a broadcast on separate threads. In PhotoGallery you need to send the broadcast from a background thread (in **`PollService.onHandleIntent(…)`**) and receive the intent on the main thread (by the dynamic receiver that is registered by **`PhotoGalleryFragment`** on the main thread in **`onStart(…)`**).

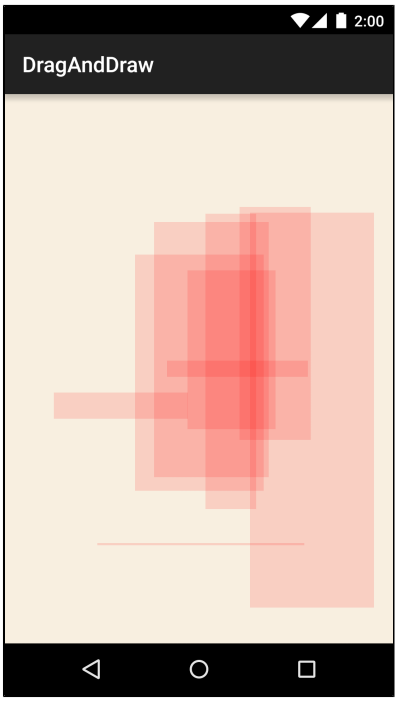
As of this writing, the semantics of **`LocalBroadcastManager`**'s thread delivery are not well documented and, in our experience, are not intuitive. For example, if you call **`sendBroadcastSync(…)`** from a background thread, all pending broadcasts will get flushed out on that background thread regardless of whether they were posted from the main thread.

This is not to say **`LocalBroadcastManager`** is not useful. It is simply not the right tool for the problems you solved in this chapter.

29
Custom Views and Touch Events

In this chapter, you will learn how to handle touch events by writing a custom subclass of **`View`** named **`BoxDrawingView`**. The **`BoxDrawingView`** class will be the star of a new project named DragAndDraw and will draw boxes in response to the user touching the screen and dragging. The finished product will look like Figure 29.1.

Figure 29.1 Boxes drawn in many shapes and sizes



Setting Up the DragAndDraw Project

Create a new project named “DragAndDraw”. Select API 16 as the minimum SDK and create an empty activity. Name the activity **`DragAndDrawActivity`**.

Setting up DragAndDrawActivity

DragAndDrawActivity will be a subclass of **SingleFragmentActivity** that inflates the usual single-fragment-containing layout. Copy `SingleFragmentActivity.java` and its `activity_fragment.xml` layout file into the DragAndDraw project.

In `DragAndDrawActivity.java`, make **DragAndDrawActivity** a **SingleFragmentActivity** that creates a **DragAndDrawFragment** (a class that you will create next).

Listing 29.1 Modifying the activity (`DragAndDrawActivity.java`)

```
public class DragAndDrawActivity extends AppCompatActivity SingleFragmentActivity {

    @Override
    public Fragment createFragment() {
        return DragAndDrawFragment.newInstance();
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
    }
}
```

Setting up DragAndDrawFragment

To prepare a layout for **DragAndDrawFragment**, rename the `activity_drag_and_draw.xml` layout file to `fragment_drag_and_draw.xml`.

DragAndDrawFragment's layout will eventually consist of a **BoxDrawingView**, the custom view that you are going to write. All of the drawing and touch-event handling will be implemented in **BoxDrawingView**.

30
Property Animation

For an app to be functional, all you need to do is write your code correctly so that it does not crash. For an app to be a joy to use, though, you need to give it more love than that. You need to make it feel like a real, physical phenomenon playing out on your phone or tablet's screen.

Real things move. To make your user interface move, you *animate* its elements into new positions.

In this chapter, you will write an app that shows a scene of the sun in the sky. When you press on the scene, it will animate the sun down below the horizon, and the sky will change colors like a sunset.

Building the Scene

The first step is to build the scene that will be animated. Create a new project called Sunset. Make sure that your `minSdkVersion` is set to `16` and use the empty activity template. Name your main activity **`SunsetActivity`**. Add `SingleFragmentActivity.java` and `activity_fragment.xml` to your project.

Now, build out your scene. A sunset by the sea should be colorful, so it will help to name a few colors. Add a `colors.xml` file to your `res/values` folder, and add the following values to it:

Listing 30.1 Adding sunset colors (`res/values/colors.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="bright_sun">#fcfcb7</color>
    <color name="blue_sky">#1e7ac7</color>
    <color name="sunset_sky">#ec8100</color>
    <color name="night_sky">#05192e</color>
    <color name="sea">#224869</color>
</resources>
```

Rectangular views will make for a fine impression of the sky and the sea. But people will not buy a rectangular sun, no matter how much you argue in favor of its technical simplicity. So, in the `res/drawable/` folder, add an oval shape drawable for a circular sun called `sun.xml`.

Listing 30.2 Adding sun XML drawable (`res/drawable/sun.xml`)

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
    <solid android:color="@color/bright_sun" />
</shape>
```

When you display this oval in a square view, you will get a circle. People will nod their heads in approval, and then think about the real sun up in the sky.

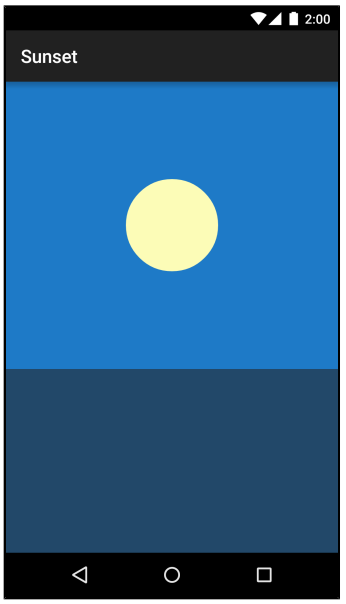
Now turn **`SunsetActivity`** into a **`SingleFragmentActivity`** that displays your fragment.

Listing 30.5 Displaying **`SunsetFragment`** (`SunsetActivity.java`)

```
public class SunsetActivity extends SingleFragmentActivity {

    @Override
    protected Fragment createFragment() {
        return SunsetFragment.newInstance();
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
    }

}
```

Take a moment to run Sunset to make sure everything is hooked up correctly before moving on. It should look like Figure 30.1. Ahhh.

Figure 30.1 Before sunset



size of that view in relation to its parent, as determined when the view was laid out. It is possible to change the location of the view on screen by modifying these values, but it is not recommended. They are reset every time a layout pass occurs, so they tend not to hold their value.

In any event, the animation will start with the top of the view at its current location. It needs to end with the top at the bottom of `mSunView`'s parent, `mSkyView`. To get it there, it should be as far down as `mSkyView` is tall, which you find by calling **`getHeight()`**. The **`getHeight()`** method returns the same thing as `getBottom() - getTop()`.

Now that you know where the animation should start and end, create and run an **`ObjectAnimator`** to perform it.

Listing 30.8 Creating a sun animator (`SunsetFragment.java`)

```
private void startAnimation() {
    float sunYStart = mSunView.getTop();
    float sunYEnd = mSkyView.getHeight();

    ObjectAnimator heightAnimator = ObjectAnimator
            .ofFloat(mSunView, "y", sunYStart, sunYEnd)
            .setDuration(3000);

    heightAnimator.start();
}
```

Then hook up **`startAnimation()`** so that it is called every time the user presses anywhere in the scene.

Listing 30.9 Starting animation on press (`SunsetFragment.java`)

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_sunset, container, false);

    mSceneView = view;
    mSunView = view.findViewById(R.id.sun);
    mSkyView = view.findViewById(R.id.sky);

    mSceneView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            startAnimation();
        }
    });

    return view;
}
```

Run Sunset and press anywhere on the scene to run the animation (Figure 30.2).

Out of the box, Android provides a basic Location API. This API lets you listen to location data from a variety of sources. For most phones, those sources are fine location points from a GPS radio and coarse points from cell towers or WiFi connections. These APIs have been around for as long as Android itself. You can find them in the `android.location` package.

So the `android.location` APIs exist. But they fall short of perfection. Real-world applications make requests like, "Use as much battery as you can to get as much accuracy as possible," or "I need a location, but I would rather not waste my battery life." Rarely if ever do they need to make a request as specific as, "Please fire up the GPS radio and tell me what it says."

This starts to be a problem when your devices move around. If you are outside, GPS is best. If you have no GPS signal, the cell tower fix may be best. And if you can find neither of those signals, it would be nicer to get by with the accelerometer and gyroscope than with no location fix at all.

In the past, high-quality apps had to manually subscribe to all of these different data sources and switch between them as appropriate. This was not straightforward or easy to do right.

Google Play Services

A better API was needed. However, if it were added to the standard library, it would take a couple of years for all developers to be able to use it. This was annoying, because the OS had everything that a better API would need: GPS, coarse location, and so forth.

Fortunately, the standard library is not the only way Google can get code into your hands. In addition to the standard library, Google provides Play Services. This is a set of common services that are installed alongside the Google Play store application. To fix this locations mess, Google shipped a new locations service in Play Services called the Fused Location Provider.

Since these libraries live in another application, you must actually have that application installed. This means that only devices with the Play Store app installed and up to date will be able to use your application. This almost certainly means that your app will be distributed through the Play Store, too. If your app is *not* available through the Play store, you are unfortunately out of luck, and will need to use another location API.

For the purposes of this exercise, if you will be testing on a hardware device make sure that you have an up-to-date Play Store app. And what if you are running on an emulator? Never fear – we will cover that later in this chapter.

Creating Locatr

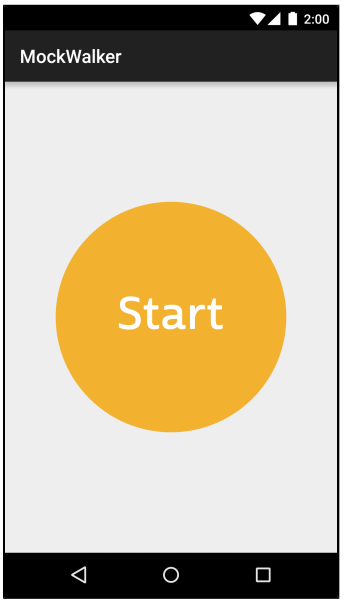
Now to get started. In Android Studio, create a new project called Locatr. Create an empty activity and name it **`LocatrActivity`**. As you have for your other apps, set your minSdkVersion to 16 and copy in **`SingleFragmentActivity`** and `activity_fragment.xml`.

You will also want some additional code from PhotoGallery. You will be querying Flickr again, so having your old query code will be handy. Open up your PhotoGallery solution (anything after Chapter 24 will do), select `FlickrFetchr.java` and `GalleryItem.java`, and right-click to copy them. Then paste them into your Java code area in Locatr.

In a minute, you will get started on building out your user interface. If you are using an emulator, though, read this next section so that you can test all the code you are about to write. If you are not, feel free to skip on ahead to the section called "Building out Locatr".

While the service is running, any time Locatr asks Fused Location Provider for a location fix, it will receive a location posted by MockWalker (Figure 31.6).

Figure 31.6 Running MockWalker



Run MockWalker and press Start. Its service will keep running after you exit the app. (Do not exit the emulator, however. Leave the emulator running while you work on Locatr.) When you no longer need those mock locations, open MockWalker again and press the Stop button.

If you would like to know how MockWalker works, you can find its source code in the solutions folder for this chapter (see the section called “Adding an Icon” in Chapter 2 for more on the solutions). It uses a few interesting things: RxJava and a sticky foreground service to manage the ongoing location updates. If those sound interesting to you, check it out.

Building out Locatr

Next, create your interface. First, add a string for your search button in `res/values/strings.xml`.

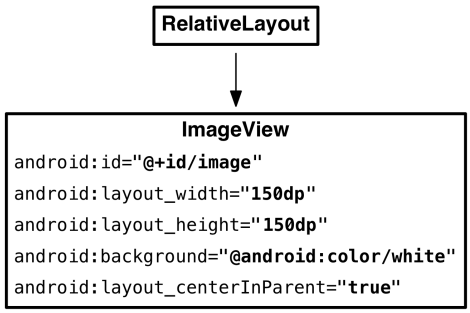
Listing 31.1 Adding search button text (`res/values/strings.xml`)

```
<resources>
    <string name="app_name">Locatr</string>

    <string name="search">Find an image near you</string>
</resources>
```

You will be using a fragment, as usual, so rename `activity_locatr.xml` to `fragment_locatr.xml`. Change out the insides of its **`RelativeLayout`** to have an **`ImageView`** to display the image you find

(Figure 31.7). (The padding attribute values come from the template code as of this writing. They are not important, so feel free to leave them out.)

Figure 31.7 Locatr's layout (`res/layout/fragment_locatr.xml`)



You also need a button to trigger the search. You can use your toolbar for that. Create `res/menu/fragment_locatr.xml` and add a menu item to display a location icon. (Yes, this is the same filename as `res/layout/fragment_locatr.xml`. This is no problem at all: menu resources live in a different namespace.)

Listing 31.2 Setting up Locatr's menu (`res/menu/fragment_locatr.xml`)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item android:id="@+id/action_locate"
        android:icon="@android:drawable/ic_menu_compass"
        android:title="@string/search"
        android:enabled="false"
        app:showAsAction="ifRoom"/>
</menu>
```

The button is disabled in XML by default. Later on, you will enable it once you are connected to Play Services.

Now create a **`Fragment`** subclass called **`LocatrFragment`** that hooks up your layout and pulls out that **`ImageView`**.

Listing 31.11 Listening for connection events (`LocatrFragment.java`)

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setHasOptionsMenu(true);

    mClient = new GoogleApiClient.Builder(getActivity())
            .addApi(LocationServices.API)
            .addConnectionCallbacks(new GoogleApiClient.ConnectionCallbacks() {
                @Override
                public void onConnected(Bundle bundle) {
                    getActivity().invalidateOptionsMenu();
                }

                @Override
                public void onConnectionSuspended(int i) {

                }
            })
            .build();
}
```

If you are curious, you can hook up an **`OnConnectionFailedListener`** and see what it reports. But it is not necessary.

With that, your Google Play Services hookup is ready.

Flickr Geosearch

The next step is to add the ability to search for geographic locations on Flickr. To do this, you perform a regular search, but you also provide a latitude and longitude.

In Android, the location APIs pass around these location fixes in **`Location`** objects. So write a new **`buildUrl(…)`** override that takes in one of these **`Location`** objects and builds an appropriate search query.

Listing 31.12 New **`buildUrl(Location)`** (`FlickrFetchr.java`)

```
private String buildUrl(String method, String query) {
    ...
}

private String buildUrl(Location location) {
    return ENDPOINT.buildUpon()
            .appendQueryParameter("method", SEARCH_METHOD)
            .appendQueryParameter("lat", "" + location.getLatitude())
            .appendQueryParameter("lon", "" + location.getLongitude())
            .build().toString();
}
```

And then write a matching **`searchPhotos(Location)`** method.

Index

Symbols

A

D