Creating Android Applications

DEVELOP AND DESIGN

Chris Haseman

Creating Android Applications

DEVELOP AND DESIGN

Chris Haseman

Peachpit Press

Creating Android Applications: Develop and Design

Chris Haseman

Peachpit Press

1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)

Find us on the Web at: www.peachpit.com
To report errors, please send a note to errata@peachpit.com
Peachpit Press is a division of Pearson Education.
Copyright © 2012 by Chris Haseman

Editor: Clifford Colby
Development editor: Robyn Thomas
Production editor: Myrna Vladic
Copyeditor: Scout Festa
Technical editor: Jason LeBrun
Cover design: Aren Howell Straiger
Interior design: Mimi Heft
Compositor: Danielle Foster
Indexer: Valerie Haynes Perry

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability

The information in this book is distributed on an "As Is" basis without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Android is a trademark of Google Inc., registered in the United States and other countries. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN-13: 978-0-321-78409-4
ISBN-10: 0-321-78409-x

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

To my wife, Meghan,
who's made me the teacher, writer, and man I am today.

BIO

Chris Haseman has been writing mobile software in various forms since 2003. He was involved in several large-scale BREW projects, from MMS messaging to Major League Baseball. More recently, he was an early Android engineer behind the doubleTwist media player, and he is now the lead Android developer for the website Tumblr. He's a faculty member of General Assembly in NYC, where he teaches Android development. He lives in Brooklyn, where he constantly debates shaving his beard.

ACKNOWLEDGMENTS

As always, I could spend more pages thanking people than are in the work itself. Here are a few who stand out:

David and Susanne H for their support. Ellen Y. for believing so early that I could do this. JBL for fixing my code. Robyn T. for her patience. Cliff C. for finding me. Scout F. for her tolerance of my grammar. Sharon H. for her harassment IMs. Dan C. for his backing. Edwin and Susan K. for their care. Thomas K. for his subtle and quiet voice. Sparks for his humor. Cotton for “being there.” Lee for the place to write. The teams at both Tumblr and doubleTwist for all their encouragement. The Android team at Google for all their hard work. Most of all, Peachpit for giving me the opportunity to write for you.

CONTENTS

Introduction . . . xi
Welcome to Android . . . xiii

CHAPTER 1 **GETTING STARTED WITH ANDROID** . . . **2**

Downloading Developer Software . . . 4
The Android Software Development Kit . . . *4*
Eclipse . . . *4*
Java . . . *4*
Getting Everything Installed . . . 5
Installing Eclipse . . . *5*
Installing the Android SDK . . . *5*
Downloading a Package . . . *6*
Configuring Eclipse . . . 8
Adding the Android Plug-in to Eclipse . . . *8*
Locating the SDK . . . *9*
Creating an Emulator . . . *10*
Working with Your Android Phone . . . *12*
Creating a New Android Project . . . 14
Running a New Project . . . 17
Troubleshooting the Emulator . . . 18
Wrapping Up . . . 19

CHAPTER 2 **EXPLORING THE APPLICATION BASICS** . . . **20**

The Files . . . 22
The Manifest . . . *22*
The Activity Class . . . 23
Watching the Activity in Action . . . *23*
Implementing Your Own Activity . . . *24*
The Life and Times of an Activity . . . *31*
Bonus Round—Data Retention Methods . . . *35*
The Intent Class . . . 37
Manifest Registration . . . *37*
Adding an Intent . . . *38*
Listening for Intents at Runtime . . . *41*
Moving Your Own Data . . . *45*
The Application Class . . . 48
The Default Application Declaration . . . *48*

Customizing Your Own Application ... 48
Accessing the Application ... 50
Wrapping Up ... 51

CHAPTER 3 **CREATING USER INTERFACES** ... **52**
The View Class ... 54
Creating a View ... 54
Altering the UI at Runtime ... 58
Handling a Few Common Tasks ... 61
Creating Custom Views ... 65
Resource Management ... 71
Resource Folder Overview ... 71
Values Folder ... 73
Layout Folders ... 74
Drawable Folders ... 76
Layout Management ... 77
The ViewGroup ... 77
The AbsoluteLayout ... 78
The LinearLayout ... 82
The RelativeLayout ... 90
Wrapping Up ... 97

CHAPTER 4 **ACQUIRING DATA** ... **98**
The Main Thread ... 100
You There, Fetch Me that Data! ... 100
Watchdogs ... 101
What Not to Do ... 102
When Am I on the Main Thread? ... 102
Getting Off the Main Thread ... 103
Getting Back to Main Land ... 104
There Must Be a Better Way! ... 105
The AsyncTask ... 106
How to Make It Work for You ... 108
A Few Important Caveats ... 111
The IntentService ... 113
Declaring a Service ... 113
Fetching Images ... 114

Checking Your Work 120
Wrapping Up 122

CHAPTER 5 **ADAPTERS, LISTVIEWS, AND LISTS** 124
Two Pieces to Each List 126
ListView 126
Adapter 126
A Main Menu 127
Creating the Menu Data 127
Creating a ListActivity 128
Defining a Layout for Your ListActivity 128
Making a Menu List Item 130
Creating and Populating the ArrayAdapter 131
Reacting to Click Events 133
Complex List Views 134
The 1000-foot View 134
Creating the Main Layout View 134
Creating the ListActivity 135
Getting Twitter Data 136
Making a Custom Adapter 138
Building the ListViews 141
How Do These Objects Interact? 144
Wrapping Up 145

CHAPTER 6 **THE WAY OF THE SERVICE** 146
What Is a Service? 148
The Service Lifecycle 148
Keeping Your Service Running 149
Shut It Down! 149
Communication 150
Intent-Based Communication 150
Binder Service Communication 160
Wrapping Up 166

CHAPTER 7 **MANY DEVICES, ONE APPLICATION** 168
Uncovering the Secrets of the res/ Folder 170
Layout Folders 170
What Can You Do Beyond Landscape? 177

The Full Screen Define 177
Limiting Access to Your App to Devices That Work 180
The <uses> Tag 180
SDK Version Number 181
Handling Code in Older Android Versions 182
SharedPreferences and Apply 182
Reflecting Your Troubles Away 183
Always Keep an Eye on API Levels 184
Wrapping Up 185

CHAPTER 8 **MOVIES AND MUSIC** 186
Movies 188
Adding a VideoView 188
Setting up for the VideoView 189
Getting Media to Play 190
Loading and Playing Media 192
Cleanup 193
The Rest, as They Say, Is Up to You 194
Music 195
MediaPlayer and State 195
Playing a Sound 196
Cleanup 197
It really is that simple 197
Longer-Running Music Playback 198
Binding to the Music Service 198
Finding the Most Recent Track 199
Playing the Audio in the Service 201
Cleanup 204
Interruptions 205
Wrapping Up 207

CHAPTER 9 **DETERMINING LOCATIONS AND USING MAPS** 208
Location Basics 210
Mother May I? 210
Be Careful What You Ask For 210
Finding a Good Supplier 211
Getting the Goods 211

The Sneaky Shortcut ... 213
That's It! ... 213
Show Me the Map! ... 214
Getting the Library ... 214
Adding to the Manifest ... 214
Creating the MapActivity ... 215
Creating a MapView ... 216
Run, Baby, Run ... 217
Wrapping Up ... 219

CHAPTER 10 **TABLETS, FRAGMENTS, AND ACTION BARS, OH MY** ... 220
Fragments ... 222
The Lifecycle of the Fragment ... 222
Creating a Fragment ... 224
Showing a Fragment ... 225
Providing Backward Compatibility ... 230
The Action Bar ... 232
Showing the Action Bar ... 232
Adding Elements to the Action Bar ... 233
Wrapping Up ... 237

CHAPTER 11 **PUBLISHING YOUR APPLICATION** ... **238**
Packaging and Versioning ... 240
Preventing Debugging ... 240
Naming the Package ... 240
Versioning ... 241
Setting a Minimum SDK value ... 242
Packaging and Signing ... 243
Exporting a Signed Build ... 243
Backing Up Your Keystore File ... 244
Submitting Your Build ... 246
Watch Your Crash Reports and Fix Them ... 246
Update Frequently ... 246
Wrapping Up ... 247

Index ... 248

INTRODUCTION

If you've got a burning idea for an application that you're dying to share, or if you recognize the power and possibilities of the Android platform, you've come to the right place. This is a short book on an immense topic.

I don't mean to alarm anyone right off the bat here, but let me be honest: Android development is hard. Its architecture is dissimilar to that of many existing platforms (especially other mobile SDKs), there are many traps for beginners to fall into, and the documentation is frequently sparse at best. In exchange for its difficulty, however, Google's Android offers unprecedented power, control, and—yes—responsibility to those who are brave enough to develop for it.

This is where my job comes in. I'm here to make the process of learning to write amazing Android software as simple as possible.

Who am I to ask such things of you? I've been writing mobile software in a professional capacity for more than eight years, and for three of those years, I've been developing software for Android. I've written code that runs on millions of handsets throughout the world. Also, I have a beard. We all know that people with ample facial hair appear to be more authoritative on all subjects.

In return for making this learning process as easy as possible, I ask for a few things:

- **You have a computer.** My third-grade teacher taught me never to take anything for granted; maybe you *don't* have a computer. If you don't already have a computer, you'll need one—preferably a fast one, because the Android emulator and Eclipse can use up a fair amount of resources quickly.

NOTE: Android is an equal opportunity development platform. While I personally develop on a Mac, you can use any of the three major platforms (Mac, PC, or Linux).

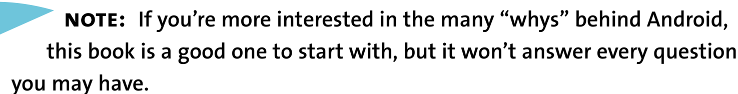
- **You're fluent in Java.** Notice that I say *fluent,* not *expert.* Because you'll be writing usable applications (rather than production libraries, at least to start), I expect you to know the differences between classes and interfaces. You should be able to handle threads and concurrency without batting an eyelash. Further, the more you know about what happens under the hood (in terms of object creation and garbage collection), the faster and better your mobile applications will be.

  Yes, you can get through the book and even put together rudimentary applications without knowing much about the Java programming language.

However, when you encounter problems—in both performance and possibilities—a weak foundation in the programming language may leave you without a solution.

- **You have boundless patience and endless curiosity.** Your interest in and passion for Android will help you through the difficult subjects covered in this book and let you glide through the easy ones.

Throughout this book, I focus on how to write features, debug problems, and make interesting software. I hope that when you've finished the book, you'll have a firm grasp of the fundamentals of Android software development.



NOTE: If you're more interested in the many "whys" behind Android, this book is a good one to start with, but it won't answer every question you may have.

All right, that's quite enough idle talking. Let's get started.

WHO THIS BOOK IS FOR

This book is for people who have some programming experience and are curious about the wild world of Android development.

WHO THIS BOOK IS NOT FOR

This book is not for people who have never seen a line of Java before. It is also not for expert Android engineers with several applications under their belt.

HOW YOU WILL LEARN

In this book, you'll learn by doing. Each chapter comes with companion sample code and clear, concise instructions for how to build that code for yourself. You'll find the code samples on the book's website (www.peachpit.com/androiddevelopanddesign).

WHAT YOU WILL LEARN

You'll learn the basics of Android development, from creating a project to building scalable UIs that move between tablets and phones.

i

WELCOME TO **ANDROID**

WELCOME TO ANDROID

Eclipse and the Android SDK are the two major tools you'll use to follow along with the examples in this book. There are, however, a few others you should be aware of that will be very useful now and in your future work with Android. While you may not use all of these tools until you're getting ready to ship an application, it will be helpful to know about them when the need arises.

THE TOOLS

Over the course of this book, you'll work with several tools that will make your life with Google's Android much easier. Here they are in no particular order:

ECLIPSE

Eclipse is the primary tool that I'll be using throughout the book. Google has blessed it as the primary IDE for Android development and has released plug-ins to help. Make sure you get them, because they take all the pain out of creating a project and stepping through your application on the device. You're welcome to use Eclipse as well, or, if you're some sort of command-line junkie, you can follow along with Vim or Emacs if you prefer.

ANDROID SDK

The Android SDK contains all the tools you'll need to develop Android applications from the command line as well as other tools to help you find and diagnose problems and streamline your applications. You can download the Android SDK at http://developer.android.com/sdk/index.html.

ANDROID SDK MANAGER

The Android SDK Manager (found within the SDK `tools/` directory) will help you pull down all versions of the SDK as well as a plethora of tools, third-party add-ons, and all things Android. This will be the primary way in which you get new software from Google's headquarters in Mountain View, California.

HIERARCHY VIEWER

This tool will help you track the complex connections between your layouts and views as you build and debug your applications. This viewer can be indispensable when tracking down those hard-to-understand layout issues. You can find this tool in the SDK `tools/` directory as `hierarchyviewer`.

DDMS

DDMS (Dalvik Debug Monitor Server) is your primary way to interface with and debug Android devices. You'll find it in the `tools/` directory inside the Android SDK. It does everything from gathering logs, sending mock text messages or locations, and mapping memory allocations to taking screenshots. Eclipse users have a perspective that duplicates, within Eclipse, all the functionality that this stand-alone application offers. This tool is very much the Swiss Army knife of your Android toolkit.

4

ACQUIRING **DATA**

While the prime directive of this chapter is to teach you how to acquire data from a remote source, this is really just a sneaky way for me to teach you about Android and the main thread. For the sake of simplicity, all the examples in this chapter will deal with downloading and rendering image data. In the next chapter, on adapters and lists, I'll introduce you to parsing complex data and displaying it to users. Image data, as a general rule, is larger and more cumbersome, so you'll run into more interesting and demonstrative timing issues in dealing with it.

THE MAIN THREAD

The Android operation system has exactly one blessed thread authorized to change anything that will be seen by the user. This alleviates what could be a concurrency nightmare, such as view locations and data changing in one thread while a different one is trying to lay them out onscreen. If only one thread is allowed to touch the user interface, Android can guarantee that nothing vital is changed while it's measuring views and rendering them to the screen. This has, unfortunately, serious repercussions for how you'll need to acquire and process data. Let me start with a simple example.

YOU THERE, FETCH ME THAT DATA!

Were I to ask you, right now, to download an image and display it to the screen, you'd probably write code that looks a lot like this:

```
public void onCreate(Bundle extra){
try{
        URL url = new URL("http://wanderingoak.net/bridge.png");
        HttpURLConnection httpCon =
            (HttpURLConnection)url.openConnection();
        if(httpCon.getResponseCode() != 200)
        throw new Exception("Failed to connect");
        InputStream is = httpCon.getInputStream();
        Bitmap bitmap = BitmapFactory.decodeStream(is);
        ImageView iv = (ImageView)findViewById(R.id.main_image);
        if(iv!=null)
            iv.setImageBitmap(bitmap);
    }catch(Exception e){
    Log.e("ImageFetching","Didn't work!",e);
    }
}
```

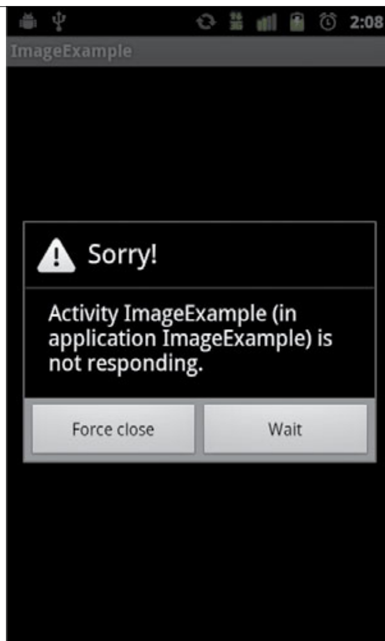


FIGURE 4.1 What the user sees when you hold the main thread hostage.

This is exactly what I did when initially faced with the same problem. While this code will fetch and display the required bitmap, there is a very sinister issue lurking in the code—namely, the code itself is running on the main thread. Why is this a problem? Consider that there can be only one main thread and that the main thread is the only one that can interact with the screen in any capacity. This means that while the example code is waiting for the network to come back with image data, nothing whatsoever can be rendered to the screen. This image-fetching code will block any action from taking place anywhere on the device. If you hold the main thread hostage, buttons will not be processed, phone calls cannot be answered, and nothing can be drawn to the screen until you release it.

WATCHDOGS

Given that a simple programmer error (like the one in the example code) could effectively cripple any Android device, Google has gone to great lengths to make sure no single application can control the main thread for any length of time. Hogging too much of the main thread's time will result in this disastrous dialog screen (**Figure 4.1**) showing up over your application.

TRACKING DOWN ANR CRASHES

Anytime you see an ANR crash, Android will write a file containing a full stack trace. You can access this file with the following ADB command line: `adb pull /data/anr/traces.txt`. This should help you find the offending line. The `traces.txt` file shows the stack trace of every thread in your program. The first thread in the list is usually the one to look at carefully. Sometimes, the long-running blocking operation will have completed before the system starts writing `traces.txt`, which can make for a bewildering stack trace. Your long-running operation probably finished just after Android started to get huffy about the main thread being delayed. In the example code that displays the image, however, it will probably show that `httpCon.getResponseCode()` was the culprit. You'll know this because it will be listed as the topmost stack trace under your application's thread list.

This dialog is unaffectionately referred to by developers as an ANR (App Not Responding) crash. Although operations will continue in the background, and the user can press the Wait button to return to whatever's going on within your application, this is catastrophic for most users, and you should avoid it at all costs.

WHAT NOT TO DO

What kind of things should you avoid on the main thread?

- Anything involving the network
- Any task requiring a read or write from or to the file system
- Heavy processing of any kind (such as image or movie modification)
- Any task blocking a thread while you wait for something to complete

Excluding this list, there isn't much left, so, as a general rule, if it doesn't involve setup or modification of the user interface, *don't* do it on the main thread.

WHEN AM I ON THE MAIN THREAD?

Anytime a method is called from the system (unless explicitly otherwise stated), you can be sure you're on the main thread. Again, as a general rule, if you're not in a thread created by you, it's safe to assume you're probably on the main one, so be careful.

GETTING OFF THE MAIN THREAD

You can see why holding the main thread hostage while grabbing a silly picture of the Golden Gate Bridge is a bad idea. But how, you might be wondering, do I get off the main thread? An inventive hacker might simply move all the offending code into a separate thread. This imaginary hacker might produce code looking something like this:

```
public void onCreate(Bundle extra){
new Thread(){
        public void run(){
            try{
                URL url = new URL("http://wanderingoak.net/bridge.
                → png");
                HttpURLConnection httpCon = (HttpURLConnection)
                → url.openConnection();
                if(httpCon.getResponseCode() != 200)
                    throw new
                    Exception("Failed to connect");
                InputStream is = httpCon.getInputStream();
                Bitmap bt = BitmapFactory.decodeStream(is);
                ImageView iv =
                (ImageView)findViewById(R.id.remote_image);
                iv.setImageBitmap(bt);
            }catch(Exception e){
                //handle failure here
            }
        }
    }.start();
}
```

"There," your enterprising hacker friend might say, "I've fixed your problem. The main thread can continue to run unimpeded by the silly PNG downloading code." There is, however, another problem with this new code. If you run the method on your own emulator, you'll see that it throws an exception and cannot display the image onscreen.

Why, you might now ask, is this new failure happening? Well, remember that the main thread is the only one allowed to make changes to the user interface? Calling `setImageBitmap` is very much in the realm of one of those changes and, thus, can be done only while on the main thread.

GETTING BACK TO MAIN LAND

Android provides, through the `Activity` class, a way to get back on the main thread as long as you have access to an activity. Let me fix the hacker's code to do this correctly. As I don't want to indent the code into the following page, I'll continue this from the line on which the bitmap is created (remember, we're still inside the `Activity` class, within the `onCreate` method, inside an inline thread declaration) (why do I hear the music from *Inception* playing in my head?).

For orientation purposes, I'll continue this from the line on which the bitmap was created in the previous code listing. If you're confused, check the sample code for this chapter.

```
final Bitmap bt = BitmapFactory.decodeStream(is);
ImageActivity.this.runOnUiThread(new Runnable() {
public void run() {
    ImageView iv = (ImageView)findViewById(R.id.remote_image);
        iv.setImageBitmap(bt);
        }
    });
//All the close brackets omitted to save space
```

Remember, we're already running in a thread, so accessing just `this` will refer to the thread itself. I, on the other hand, need to invoke a method on the activity. Calling `ImageActivity.this` provides a pointer to the outer `Activity` class in which we've spun up this hacky code and will thus allow us to call `runOnUiThread`. Further, because I want to access the recently created bitmap in a different thread, I'll need to make the bitmap declaration `final` or the compiler will get cranky with us.

When you call `runOnUiThread`, Android will schedule this work to be done as soon as the main thread is free from other tasks. Once back on the main thread, all the same "don't be a hog" rules again apply.

THERE MUST BE A BETTER WAY!

If you're looking at this jumbled, confusing, un-cancelable code and thinking to yourself, "Self. There must be a cleaner way to do this," you'd be right. There are many ways to handle long-running tasks; I'll show you what I think are the two most useful. One is the `AsyncTask`, a simple way to do an easy action within an activity. The other, `IntentService`, is more complicated but much better at handling repetitive work that can span multiple activities.

THE ASYNCTASK

At its core, the AsyncTask is an abstract class that you extend and that provides the basic framework for a time-consuming asynchronous task.

The best way to describe the AsyncTask is to call it a working thread sandwich. That is to say, it has three major methods for which you must provide implementation.

1. onPreExecute takes place on the main thread and is the first slice of bread. It sets up the task, prepares a loading dialog, and warns the user that something is about to happen.

2. doInBackground is the meat of this little task sandwich. This method is guaranteed by Android to run on a separate background thread. This is where the majority of your work takes place.

3. onPostExecute will be called once your work is finished (again, on the main thread), and the results produced by the background method will be passed to it. This is the other slice of bread.

That's the gist of the asynchronous task. There are more-complicated factors that I'll touch on in just a minute, but this is one of the fundamental building blocks of the Android platform (given that all hard work must be taken off the main thread).

Take a look at one in action, then we'll go over the specifics of it:

```
private class ImageDownloader
extends AsyncTask<String, Integer, Bitmap>{
protected void onPreExecute(){
        //Setup is done here
    }
    @Override
    protected Bitmap doInBackground(String... params) {
        // TODO Auto-generated method stub
        try{
            URL url = new URL(params[0]);
```

```
            HttpURLConnection httpCon =
            (HttpURLConnection)url.openConnection();
            if(httpCon.getResponseCode() != 200)
                throw new Exception("Failed to connect");
            InputStream is = httpCon.getInputStream();
            return BitmapFactory.decodeStream(is);
        }catch(Exception e){
            Log.e("Image","Failed to load image",e);
        }
        return null;
    }
    protected void onProgressUpdate(Integer... params){
        //Update a progress bar here, or ignore it, it's up to you
    }
    protected void onPostExecute(Bitmap img){
        ImageView iv = (ImageView)findViewById(R.id.remote_image);
        if(iv!=null && img!=null){
            iv.setImageBitmap(img);
        }
    }
        protected void onCancelled(){
        }
    }
```

That, dear readers, is an asynchronous task that will download an image at the end of any URL and display it for your pleasure (provided you have an image view onscreen with the ID `remote_image`). Here is how you'd kick off such a task from the `onCreate` method of your activity.

```
public void onCreate(Bundle extras){
    super.onCreate(extras);
    setContentView(R.layout.image_layout);
    id = new ImageDownloader();
    id.execute("http://wanderingoak.net/bridge.png");
}
```

Once you call `execute` on the `ImageDownloader`, it will download the image, process it into a bitmap, and display it to the screen. That is, assuming your `image_layout.xml` file contains an `ImageView` with the ID `remote_image`.

HOW TO MAKE IT WORK FOR YOU

The `AsyncTask` requires that you specify three generic type arguments (if you're unsure about Java and generics, do a little Googling before you press on) as you declare your extension of the task.

- The type of parameter that will be passed into the class. In this example `AsyncTask` code, I'm passing one string that will be the URL, but I could pass several of them. The parameters will always be referenced as an array no matter how many of them you pass in. Notice that I reference the single URL string as `params[0]`.
- The object passed between the `doInBackground` method (*off* the main thread) and the `onProgressUpdate` method (which will be called *on* the main thread). It doesn't matter in the example, because I'm not doing any progress updates in this demo, but it'd probably be an integer, which would be either the percentage of completion of the transaction or the number of bytes transferred.
- The object that will be returned by the `doInBackground` method to be handled by the `onPostExecute` call. In this little example, it's the bitmap we set out to download.

Here's the line in which all three objects are declared:

```
private class ImageDownloader extends
    AsyncTask<String, Integer, Bitmap>{
```

In this example, these are the classes that will be passed to your three major methods.

ONPREEXECUTE

```
protected void onPreExecute(){
}
```

`onPreExecute` is usually when you'll want to set up a loading dialog or a loading spinner in the corner of the screen (I'll discuss dialogs in depth later). Remember, `onPreExecute` is called on the main thread, so don't touch the file system or network at all in this method.

DOINBACKGROUND

```
protected Bitmap doInBackground(String... params) {
}
```

This is your chance to make as many network connections, file system accesses, or other lengthy operations as you like without holding up the phone. The class of object passed to this method will be determined by the first generic object in your `AsyncTask`'s class declaration. Although I'm using only one parameter in the code sample, you can actually pass any number of parameters (as long as they derive from the saved class) and you'll have them at your fingertips when `doInBackground` is called. Once your long-running task has been completed, you'll need to return the result at the end of your function. This final value will be passed into another method called back on the main UI thread.

BEWARE LOADING **DIALOGS**

Remember that mobile applications are not like their web or desktop counterparts. Your users will typically be using their phones when they're away from a conventional computer. This means, usually, that they're already waiting for something: a bus, that cup of expensive coffee, their friend to come back from the bathroom, or a boring meeting to end. It's very important, therefore, to keep them from having to wait on anything within your application. Waiting for your mobile application to connect while you're already waiting for something else can be a frustrating experience. Do what you can to limit users' exposure to full-screen loading dialogs. They're unavoidable sometimes, but minimize them whenever possible.

SHOWING YOUR PROGRESS

There's another aspect of the `AsyncTask` that you should be aware of even though I haven't demonstrated it. From within `doInBackground`, you can send progress updates to the user interface. `doInBackground` isn't on the main thread, so if you'd like to update a progress bar or change the state of something on the screen, you'll have to get back on the main thread to make the change.

Within the `AsyncTask`, you can do this during the `doInBackground` method by calling `publishProgress` and passing in any number of objects deriving from the second class in the `AsyncTask` declaration (in the case of this example, an integer). Android will then, on the main thread, call your declared `onProgressUpdate` method and hand over any classes you passed to `publishProgress`. Here's what the method looks like in the `AsyncTask` example:

```
protected void onProgressUpdate(Integer... params){
    //Update a progress bar here, or ignore it, it's up to you
}
```

As always, be careful when doing UI updates, because if the activity isn't currently onscreen or has been destroyed, you could run into some trouble.

ONPOSTEXECUTE

The work has been finished or, in the example, the image has been downloaded. It's time to update the screen with what I've acquired. At the end of `doInBackground`, if successful, I return a loaded bitmap to the `AsyncTask`. Now Android will switch to the main thread and call `onPostExecute`, passing the class I returned at the end of `doInBackground`. Here's what the code for that method looks like:

```
protected void onPostExecute(Bitmap img){
    ImageView iv = (ImageView)findViewById(R.id.remote_image);
    if(iv!=null && img!=null){
        iv.setImageBitmap(img);
    }
}
```

I take the bitmap downloaded from the website, retrieve the image view into which it's going to be loaded, and set it as that view's bitmap to be rendered. There's an error case I haven't correctly handled here. Take a second to look back at the original code and see if you can spot it.

A FEW IMPORTANT CAVEATS

Typically, an `AsyncTask` is started from within an activity. However, you must remember that activities can have short life spans. Recall that, by default, Android destroys and re-creates any activity each time you rotate the screen. Android will also destroy your activity when the user backs out of it. You might reasonably ask, "If I start an `AsyncTask` from within an activity and then that activity is destroyed, what happens?" You guessed it: very bad things. Trying to draw to an activity that's already been removed from the screen can cause all manner of havoc (usually in the form of unhandled exceptions).

It's a good idea to keep track of any `AsyncTask`s you've started, and when the activity's `onDestroy` method is called, make sure to call `cancel` on any lingering `AsyncTask`.

There are a few cases in which the `AsyncTask` is perfect for the job:

- Downloading small amounts of data specific to one particular activity
- Loading files from an external storage drive (usually an SD card)

Make sure, basically, that the data you're moving with the AsyncTask pertains to only one activity, because your task generally shouldn't span more than one. You can pass it between activities if the screen has been rotated, but this can be tricky.

There are a few cases when it's not a good idea to use an AsyncTask:

- Any acquired data that may pertain to more than one activity shouldn't be acquired through an AsyncTask. Both an image that might be shown on more than one screen and a list of messages in a Twitter application, for example, would have relevance outside a single activity.
- Data to be posted to a web service is also a bad idea to put on an AsyncTask for the following reason: Users will want to fire off a post (posting a photo, blog, tweet, or other data) and do something else, rather than waiting for a progress bar to clear. By using an AsyncTask, you're forcing them to wait around for the posting activity to finish.
- Last, be aware that there is some overhead for the system in setting up the AsyncTask. This is fine if you use a few of them, but it may start to slow down your main thread if you're firing off hundreds of them.

You might be curious as to exactly what you should use in these cases. I'm glad you are, because that's exactly what I'd like to show you next.

THE INTENTSERVICE

The IntentService is an excellent way to move large amounts of data around without relying on any specific activity or even application. The AsyncTask will always take over the main thread at least twice (with its pre- and post-execute methods), and it must be owned by an activity that is able to draw to the screen. The IntentService has no such restriction. To demonstrate, I'll show you how to download the same image, this time from the IntentService rather than the AsyncTask.

DECLARING A SERVICE

Services are, essentially, classes that run in the background with no access to the screen. In order for the system to find your service when required, you'll need to declare it in your manifest, like so:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.haseman.Example"
        android:versionCode="1"
        android:versionName="1.0">
    <application
        android:name="MyApplication"
        android:icon="@drawable/icon"
        android:label="@string/app_name">
    <!—Rest of the application declarations go here -->
        <service android:name=".ImageIntentService"/>
    </application>
</manifest>
```

At a minimum, you'll need to have this simple declaration. It will then allow you to (as I showed you earlier with activities) explicitly launch your service. Here's the code to do exactly that:

```
Intent i = new Intent(this, ImageIntentService.class);
i.putExtra("url", getIntent().getExtras().getString("url"));
startService(i);
```

At this point, the system will construct a new instance of your service, call its onCreate method, and then start firing data at the IntentService's handleIntent method. The intent service is specifically constructed to handle large amounts of work and processing off the main thread. The service's onCreate method *will* be called on the main thread, but subsequent calls to handleIntent are guaranteed by Android to be on a background thread (and this is where you should put your long-running code in any case).

Right, enough gabbing. Let me introduce you to the ImageIntentService. The first thing you'll need to pay attention to is the constructor:

```
public class ImageIntentService extends IntentService{
    public ImageIntentService() {
        super("ImageIntentService");
    }
```

Notice that the constructor you must declare has no string as a parameter. The parent's constructor that you must call, however, must be passed a string. Eclipse will make it seem that you must declare a constructor with a string when, in reality, you must declare it without one. This simple mistake can cause you several hours of intense face-to-desk debugging.

Once your service exists, and before anything else runs, the system will call your onCreate method. onCreate is an excellent time to run any housekeeping chores you'll need for the rest of the service's tasks (more on this when I show you the image downloader).

At last, the service can get down to doing some heavy lifting. Once it has been constructed and has had its onCreate method called, it will then receive a call to handleIntent for each time any other activity has called startService.

FETCHING IMAGES

The main difference between fetching images and fetching smaller, manageable data is that larger data sets (such as images or larger data retrievals) should not be bundled into a final broadcast intent (another major difference to the AsyncTask). Also, keep in mind that the service has no direct access to any activity, so it cannot

ever access the screen on its own. Instead of modifying the screen, the IntentService will send a broadcast intent alerting all listeners that the image download is complete. Further, since the service cannot pass the actual image data along with that intent, you'll need to save the image to the SD card and include the path to that file in the final completion broadcast.

THE SETUP

Before you can use the external storage to cache the data, you'll need to create a cache folder for your application. A good place to check is when the IntentService's onCreate method is called:

```
public void onCreate(){
    super.onCreate();
    String tmpLocation =
        Environment.getExternalStorageDirectory().getPath()
        + CACHE_FOLDER;
    cacheDir = new File(tmpLocation);
    if(!cacheDir.exists()){
        cacheDir.mkdirs();
    }
}
```

Using Android's environment, you can determine the correct prefix for the external file system. Once you know the path to the eventual cache folder, you can then make sure the directory is in place. Yes, I know I told you to avoid file-system contact while on the main thread (and onCreate is called on the main thread), but checking and creating a directory is a small enough task that it should be all right. I'll leave this as an open question for you as you read through the rest of this chapter: Where might be a better place to put this code?

A **NOTE** ON **FILE SYSTEMS**

Relying on a file-system cache has an interesting twist with Android. On most phones, the internal storage space (used to install applications) is incredibly limited. You should not, under any circumstances, store large amounts of data anywhere on the local file system. Always save it to a location returned from `getExternalStorageDirectory`.

When you're saving files to the SD card, you must also be aware that nearly all pre-2.3 Android devices can have their SD cards removed (or mounted as a USB drive on the user's laptop). This means you'll need to gracefully handle the case where the SD card is missing. You'll also need to be able to forgo the file-system cache on the fly if you want your application to work correctly when the external drive is missing. There are a lot of details to be conscious of while implementing a persistent storage cache, but the benefits (offline access, faster start-up times, fewer app-halting loading dialogs) make it more than worth your effort.

THE FETCH

Now that you've got a place to save images as you download them, it's time to implement the image fetcher. Here's the `handleIntent` method:

```
protected void onHandleIntent(Intent intent) {
    String remoteUrl = intent.getExtras().getString("url");
    String location;
    String filename =
        remoteUrl.substring(
        remoteUrl.lastIndexOf(File.separator)+1);
    File tmp = new File(cacheDir.getPath()
            + File.separator +filename);
    if(tmp.exists()){
        location = tmp.getAbsolutePath();
        notifyFinished(location, remoteUrl);
```

```
            stopSelf();
            return;
        }
        try{
            URL url = new URL(remoteUrl);
            HttpURLConnection httpCon =
                (HttpURLConnection)url.openConnection();
            if(httpCon.getResponseCode() != 200)
                throw new Exception("Failed to connect");
            InputStream is = httpCon.getInputStream();
            FileOutputStream fos = new FileOutputStream(tmp);
            writeStream(is, fos);
            fos.flush(); fos.close();
            is.close();
            location = tmp.getAbsolutePath();
            notifyFinished(location, remoteUrl);
        }catch(Exception e){
            Log.e("Service","Failed!",e);
        }
    }
```

This is a lot of code. Fortunately, most of it is stuff you've seen before.

First, you retrieve the URL to be downloaded from the `Extras` bundle on the intent. Next, you determine a cache file name by taking the last part of the URL. Once you know what the file will eventually be called, you can check to see if it's already in the cache. If it is, you're finished, and you can notify the system that the image is available to load into the UI.

If the file isn't cached, you'll need to download it. By now you've seen the `HttpUrlConnection` code used to download an image at least once, so I won't bore you by covering it. Also, if you've written any Java code before, you probably know how to write an input stream to disk.

THE CLEANUP

At this point, you've created the cache file, retrieved it from the web, and written it to the aforementioned cache file. It's time to notify anyone who might be listening that the image is available. Here's the contents of the `notifyFinished` method that will tell the system both that the image is finished and where to get it.

```
public static final String TRANSACTION_DONE =
                "com.haseman.TRANSACTION_DONE";
private void notifyFinished(String location, String remoteUrl){
    Intent i = new Intent(TRANSACTION_DONE);
    i.putExtra("location", location);
    i.putExtra("url", remoteUrl);
    ImageIntentService.this.sendBroadcast(i);
}
```

Anyone listening for the broadcast intent `com.haseman.TRANSACTION_DONE` will be notified that an image download has finished. They will be able to pull both the URL (so they can tell if it was an image it actually requested) and the location of the cached file.

RENDERING THE DOWNLOAD

In order to interact with the downloading service, there are two steps you'll need to take. You'll need to start the service (with the URL you want it to fetch). Before it starts, however, you'll need to register a listener for the result broadcast. You can see these two steps in the following code:

```
public void onCreate(Bundle extras){
    super.onCreate(extras);
    setContentView(R.layout.image_layout);
    IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction(ImageIntentService.TRANSACTION_DONE);
    registerReceiver(imageReceiver, intentFilter);
```

```
    Intent i = new Intent(this, ImageIntentService.class);
    i.putExtra("url",
getIntent().getExtras().getString("url"));
    startService(i);
    pd = ProgressDialog.show(this, "Fetching Image",
"Go intent service go!");
}
```

This code registered a receiver (so you can take action once the download is finished), started the service, and, finally, showed a loading dialog to the user.

Now take a look at what the imageReceiver class looks like:

```
private BroadcastReceiver imageReceiver = new BroadcastReceiver() {
@Override
    public void onReceive(Context context, Intent intent) {
        String location = intent.getExtras().getString("location");
        if(location == null || location.length() ==0){
            Toast.makeText(context, "Failed to download image",
                    Toast.LENGTH_LONG).show();
        }
        File imageFile = new File(location);
        if(!imageFile.exists()){
            pd.dismiss();
            Toast.makeText(context,
                "Unable to Download file :-(",
                Toast.LENGTH_LONG);
            return;
        }
        Bitmap b = BitmapFactory.decodeFile(location);
```

```
        ImageView iv = (ImageView)findViewById(R.id.remote_image);
        iv.setImageBitmap(b);
        pd.dismiss();
    }
};
```

This is a custom extension of the BroadcastReceiver class. This is what you'll need to declare inside your activity in order to correctly process events from the IntentService. Right now, there are two problems with this code. See if you can recognize them.

First, you'll need to extract the file location from the intent. You do this by looking for the "location" extra. Once you've verified that this is indeed a valid file, you'll pass it over to the BitmapFactory, which will create the image for you. This bitmap can then be passed off to the ImageView for rendering.

Now, to the things done wrong (stop reading if you haven't found them yet. No cheating!). First, the code is not checking to see if the intent service is broadcasting a completion intent for exactly the image originally asked for (keep in mind that one service can service requests from any number of activities).

Second, the bitmap is loading from the SD card. . .on the main thread! Exactly one of the things I've been warning you NOT to do.

CHECKING YOUR WORK

Android, in later versions of the SDK tools, has provided a way to check if your application is breaking the rules and running slow tasks on the main thread. You can, in any activity, call StrictMode.enableDefaults, and this will begin to throw warnings when the system spots main thread violations. StrictMode has many different configurations and settings, but enabling the defaults and cleaning up as many errors as you can will work wonders for the speed of your application.

THE **LOADER**

`Loader` is a new class that comes both in Honeycomb and in the Android Compatibility library. Sadly, there is not enough space in this chapter to cover it in detail, but I will say that it's an excellent tool to explore if you must do heavy lifting off the main thread repeatedly. It, like `AsyncTask`, is usually bound to an activity, but it is much better suited to handle situations where a single task must be performed many times. It's great for loading cursors (with the `CursorLoader` subclass) and for other tasks, like downloading individual list items for a `ListView`. Check the documentation for how best to use this new and powerful class.

WRAPPING UP

That about covers us on how to load data. Remember, loading from the SD card, network transactions, and longer processing tasks MUST be performed off the main thread, or your application, and users, will suffer. You can, as I've shown you in this chapter, use a simple thread, an `AsyncTask`, or an `IntentService` to retrieve and process your data. But remember, too, that any action modifying any view or object onscreen must be carried out on the main thread (or Android will throw angry exceptions at you).

Further, keep in mind that these three methods are only a few of many possible background data fetching patterns. `Loaders`, `Workers`, and `ThreadPools` are all other alternatives that might suit your application better than the examples I've given.

Follow the simple rules I've outlined here, and your app will be fast, it will be responsive to your users, and it will avoid the dreaded App Not Responding notification of doom. Correct use and avoidance of the main thread is critical to producing a successful application.

If you're more interested in building lists out of complex data from remote sources, the next chapter should give you exactly what you're looking for. I'll be showing you how to render a list of Twitter messages to a menu onscreen.

I'll leave you with a final challenge: Enable Android's strict mode and move the little file accesses I've left in this chapter's sample code off the main thread. It should be a good way to familiarize yourself with the process before you undertake it on your own.

This page intentionally left blank

INDEX

: (colon), using with services, 161

A

AbsoluteLayout, 78–82
action bar
  action views, 236
  adding icons to, 233–234
  adding tabs, 235
  delete icon, 233
  drop-down list action view, 236
  features of, 232
  icon clicks, 234–235
  showing, 232
activities. *See also* intents
  bundle objects, 35
  colliding, 44–45
  considering for applications, 49
  constructor, 31
  creating, 24–27
  data retention methods, 35
  destroying, 32, 36
  getting intents, 31
  Intent class, 37
  intents, 29–31, 37
  key handling method, 29
  launching, 28–30
  listening for key events, 28
  locating, 25
  NewActivity class, 39
  onCreate method, 24, 31–33
  onDestroy method, 31, 34
  onKeyDown method, 28–30, 40
  onPause method, 31, 34
  onResume method, 31
  OnRetainNonConfigurationInstance method, 35
  onSaveInstanceState method, 35–36
  onStart method, 31
  onStop method, 31, 34
  public void onCreate(bundle icicle), 32–33
  public void onResume(), 33
  public void onStart method, 33
  receiving events, 41
  screen layout, 27–29
  separating from layout files, 178
  setContentView method, 28, 33
  StrictMode.enableDefaults, 120
  TextView ID, 27
Activity class, creating, 25–27
activity declaration, android:name tag, 23
Adapter class
  getCount method, 138
  getItem method, 138
  getItemId method, 138
  GetView method, 139
  interaction with ListView class, 126, 144
adapters, customizing, 138–140
adb pull /data/anr/traces.txt command line, 102
ADT plug-in, adding to Eclipse, 8–9
AIDL (Android Interface Definition Language), 160–162
Android, older versions of, 182–184
android create project command, 16
Android Developers website, 4
Android folder, displaying, 14
Android phone
  USB debugging, 12
  using, 12
Android projects
  creating, 14–16
  creating from command line, 16
  DDMS perspective, 17
  Java package, 15
  naming, 15
  naming activities, 15
  naming applications, 15
  running, 17
  selecting, 14
  selecting version of, 15
Android SDK
  downloading, xiv, 4
  installing for Linux users, 6
  installing for Mac users, 5–6
  installing for Windows users, 6

Android SDK Manager
  described, xv
  locating, xv
  using, 6–7
Android Virtual Device (AVD), configuring, 11–12
AndroidManifest.xml file
  <manifest> declaration, 22
  package definition, 22
android:name tag, 23
ANR crashes, tracking down, 102
ant install command, 16
API levels, monitoring, 184
APK file, watching size of, 76
Application class
  accessing, 50–51
  accessing variables, 51
  activities, 49
  adding data to, 50
  customizing, 48–50
  default declaration, 48
  getApplication method, 50
applications
  minimum SDK value, 242
  names, 48–49
  preventing debugging, 240
  updating, 241
apps, limiting access to, 180–181
ArrayAdapter class, creating and populating, 131–132
AsyncTask class, 106–112
  avoiding use of, 112
  doInBackground method, 106, 109
  keeping track of, 111
  onPostExecute method, 106, 111
  onPreExecute method, 106, 109
  publishProgress method, 110
  showing progress, 110
  starting within activities, 111
  type arguments, 108–109
  using, 111–112
audio, playing in services, 201–204
AVD (Android Virtual Device), configuring, 11–12

B

binder service communication, 160–165
  binder and AIDL stub, 162–164
  creating services, 161–162
bitmaps, fetching and displaying, 100–101
BroadcastReceiver
  creating for intents, 41–43
  registering, 42–43
  self-contained, 44
builds
  crash reports, 246
  submitting, 246
  updating, 246
button bar layout, 87
button_layout.xml file, creating, 172, 174–175
buttons
  adding to services, 152
  layout XML, 170–171

C

cache folder, creating for images, 115
call state, watching, 205
cd command, 16
classes
  Activity, 25–27
  imageReceiver, 119–120
  Intent, 37
  Loader, 121
click events, reacting to, 133
click listeners
  adding to buttons, 65
  calling for views, 62
  registering with views, 63
  setting, 65
colon (:), using with services, 161
command line, creating projects from, 16
communication. *See also* services
  binder service, 160–165
  intent-based, 150–159
compatibility library, using with fragments, 230–231
content observer, registering, 154

ContentFragment class, 224–225
ContentObserver, using with services, 158
ContentProvider
  cursor for, 159
  registering observer with, 154
cursor loader, using for music playback, 199
cursor.close, calling on cursors, 159
cursors
  closing for media, 193
  moving to media, 192
custom views. *See also* extended views; views
  adding to XML, 70
  declaring class for, 65–66
  extending, 66, 68–69

D

data, fetching and displaying, 100–101
DDMS (Dalvik Debug Monitor Server), xv
  perspective, opening, 17
debugging
  layout issues, 179
  preventing, 240
dialogs, beware of loading, 110
drawable folders
  contents of, 71
  referencing, 76
  using, 76

E

Eclipse IDE, xiv
  adding Android plug-in to, 8–9
  backing up keystore file, 244–246
  creating activities in, 25–27
  creating emulator, 10–13
  creating views, 55
  declaring services, 114
  downloading, 4
  exporting signed build, 243–244
  IMusicService.java file, 161
  installing, 5
  locating Android SDK, 9–10
  Zipalign tool, 245
emulator
  creating, 10–13
  troubleshooting, 18–19
exceptions, handling, 137
exporting release build
  release build, 243
  signed build, 243–244
extended views. *See also* custom views; views
  changing colors, 67
  creating instances, 68
  customizing, 66–68
  ForegroundColorSpan, 66–67
  using, 68–70

F

file system cache, relying on, 116
files, 22
  directory, 23
  locating, 23
  saving to SD cards, 116
folders, 22
ForegroundColorSpan, using with extended views, 66–67
FragmentActivity class, 226
FragmentManager, 229–230
fragments
  backward compatibility, 230–231
  compatibility library, 230–231
  content view for FragmentActivity, 226
  ContentFragment class, 224–225
  creating, 224–225
  declaring in XML layout, 225–226
  DemoListFragment, 228–229
  features of, 222
  layouts, 224–225
  lifecycle, 222–223
  onAttach method, 222
  onCreate method, 222
  onCreateView method, 222
  onDestroy method, 223

onDestroyView method, 223
onDetach method, 223
onPause method, 223
onResume method, 222
onStart method, 222
onStop method, 223
placing onscreen, 228–230
showing, 225–230
text view, 225, 227

G

GeoPoints, using with maps, 219
getApplication method, 50
Google Maps library, 214, 216
gray background, adding to RelativeLayout, 95–96

H

hierarchy viewer, locating, xv
Honeycomb
action bar, 232
action views, 236
FragmentActivity class, 226
FragmentManager, 229
Navigation, 232
SetShowAsAction, 234

I

Ice Cream Sandwich
action bar, 232
FragmentActivity class, 226
Navigation, 232
icon clicks, reacting to, 234–235
icons, adding to action bar, 233–234
image fetcher
handleIntent method, 116–117
implementing, 116–117
image uploading, automatic, 150–151
ImageIntentService, 114
imageReceiver class, 119–120
images
cache folder, 115
downloading and displaying, 100–101
external storage, 115
fetching, 114–120
listener for result broadcast, 118–119
notifyFinished method, 118
rendering download, 118–120
<include> tag, using for small changes, 172–176
installing
Android SDK for Linux users, 6
Android SDK for Mac users, 5–6
Android SDK for Windows users, 6
Eclipse IDE, 5
Intent class manifest registration, 37–38
intent filters, registering for, 40
intent-based communication, 150–159
auto image uploading, 150–151
declaring services, 151
getting services, 151
going to foreground, 155–157
observing content changes, 158–159
spinning up services, 154–155
starting services, 152–154
intents. *See also* activities
adding, 38–40
BroadcastReceiver, 41–43
creating, 29–30
features of, 37
getting for activities, 31
listening for, 41–45
listening for information, 43
moving data, 45–47
receivers, 41–43
receiving, 37
registering receivers, 42–43
retrieving and using strings, 46–47
reviewing, 47
self-contained BroadcastReceivers, 44
stopping listening, 43
toasts, 42

IntentService
declaring services, 113–114
fetching images, 114–120

J

Java
views in, 56–58
versus XML layouts, 60
JSONArray object, using with list views, 139–140

K

key, creating, 244–245
keystore file
backing up, 244–245
creating, 244–245

L

layout files, separating from activities, 178
layout folders, 170–176
adding suffixes to, 177
buttons, 170–171
contents of, 71, 75–76
<include> tag, 172–176
MVC (Model-View-Controller), 75
specifying, 172
layout issues, debugging, 179
layout-land folder
creating, 172
defining screens in, 177–178
layouts
AbsoluteLayout, 78–82
button bar, 87
height and width values, 55, 57, 86
LinearLayout, 82–89
nesting, 84
RelativeLayout, 90–96
ViewGroup, 77–78
XML versus Java, 60
LinearLayouts, 82–89, 130
button bar layout, 87
layout of children, 84
nesting layouts, 84
orientation, 86
padding option, 88–89
versus RelativeLayouts, 84, 89
using, 87
list element rows, recycling, 144
List Fragment, 126
list views
building, 141–142
custom layout view, 142–143
fetching data, 138
getting Twitter data, 136–138
getTwitterFeed, 138
getView code, 142
handling exceptions, 137
interaction with Adapter class, 144
JSONArray object, 139–140
JSONObject, 142
ListActivity class, 135–136, 139–140
main layout view, 134–135
onCreate method, 135
TextViews, 142–143
ListActivity class, 139–140
creating, 128–130
IDs, 129
XML layout file, 128–129
ListView class
custom adapter, 138–140
described, 126
Loader class
described, 121
using for music playback, 200–201
location service
distanceBetween method, 212
finding supplier, 211
getBestProvider method, 211
getLastKnownLocation, 213
LocationListener interface, 212

LocationManager object, 212
onLocationChanged method, 212
registering for updates, 211–212
using, 211
locations
adding permission to manifest, 210
getting for devices, 210
<uses-permission> tag, 210
logging, disabling, 244

M

main menu
ArrayAdapter class, 131–132
click events, 133
data, 127
list items, 130–131
ListActivity class, 128–130
main thread. *See also* thread violations
being on, 102
fetching data, 100–101
getting back on, 104–105
getting off, 103–105
Loader class, 121
recommendations, 102
manifest, 22
map key, getting, 217
MapActivity class
availability of, 214
creating, 215–216
MapControl class, 217–218
maps
manifest additions for, 214–215
using GeoPoints with, 219
MapView class
availability of, 214
creating, 216–217
testing, 217–218
value for apiKey field, 216–217
media
ContentProvider, 190
ContentResolver, 191
Cursor object, 191
loading, 192–193
moving cursor to, 192
onErrorListener, 194
playing, 192–193
playNextVideo, 192
searching SD cards for, 191
media players
cleanup, 204–205
onDestroy method, 204–205
MediaPlayer states
Idle, 195
Initialized, 195
Playing, 195
Prepared, 195
MediaScanner, 191
menu list items, text view file, 130–131
<merge> tag, wrapping views in, 176
methodNotFoundException, 184
Model-View-Controller (MVC), 75
movie playback
adding VideoView, 188–189
cleanup, 193
closing cursors, 193
onDestroy method, 193
process, 188
setting up for VideoView, 189–190
music playback
audio focus, 205
cleanup, 197, 204–205
closing cursors, 204–205
crashing service, 205
cursor loader, 199
finding recent track, 199–201
headphone controls, 205
icon in notification area, 203
interruptions, 205–206
Loader class, 200–201
missing SD card, 206
onDestroy method, 197, 204
phone calls, 205

music playback (*continued*)
  playing audio in services, 201–203
  setDataSource, 201–202
  setForegroundState method, 203
  sounds, 196–197
  stop method, 204
music service, binding to, 198–199
MusicExampleActivity, 198
MVC (Model-View-Controller), 75

N

New York City, map of, 218
NewActivity class, 39
Next button, creating with RelativeLayout, 93–94
Notification object, creating for services, 156
notification pull-down, creating for services, 157

O

OnClickListener, using with views, 62–65
onCreate method, 24
  ListView class, 135
  using with views, 63
onDestroy method, using with activities, 34
onErrorListener, using with media, 194
onKeyDown method, using with activities, 28–30, 40
onPause method, using with activities, 34
OnRetainNonConfigurationInstance method, 35
onSaveInstanceState method, 35–36
onStop method, using with activities, 34

P

packages
  downloading, 6–7
  naming, 240–241
packaging
  and signing, 243–245
  and versioning, 240–242
padding
  LinearLayouts, 88–89
  RelativeLayouts, 93
permission, adding to manifest, 210
phone's call state, watching, 205
photo listening service
  registering for media notification, 154
  starting, 153
  stopping, 153
photos, uploading, 159
playNextVideo, 192
preferences, saving usernames to, 182
projects
  creating, 14–16
  creating from command line, 16
  DDMS perspective, 17
  Java package, 15
  naming, 15
  naming activities, 15
  naming applications, 15
  running, 17
  selecting, 14
  selecting version of, 15
public void onResume method, using with activities, 33
public void onStart method, using with activities, 33

R

reflection
  accessing SDK methods with, 183–184
  benefits of, 184
  methodNotFoundException, 184
RelativeLayouts, 90–96
  gray background, 95–96
  versus LinearLayouts, 84, 89
  Next button, 93–94
  padding declaration, 93
  <RelativeLayout> declaration, 92
  using, 90–96
release build, exporting, 243
res/ folder
  contents of, 71
  layout folders, 170–177
resources, finding, 59

R.javafile
  code, 72
  creation of, 71

S

saving files to SD cards, 116
screen layout, creating for activities, 27–29
screen sizes, handling, 75, 89
screens, defining in layout-land folder, 177–178
SD card, saving files to, 116
SDK (software development kit)
  downloading, xiv, 4
  installing for Linux users, 6
  installing for Mac users, 5–6
  installing for Windows users, 6
SDK methods, accessing with reflection, 183–184
SDK value, setting, 242
SDK version number
  declaring support for, 181
  finding, 184
Service class
  described, 148
  onBind method, 151
ServiceExampleActivity, 152–153
services. *See also* communication
  binding and communicating with, 164–165
  bringing into foreground, 155
  colon (:) in process, 161
  ContentObserver, 158
  Context.stopService, 149
  creating, 161–162
  creating notifications, 155–156
  cursor for ContentProvider, 159
  declaring, 113–114, 151
  getting, 151
  ImageIntentService, 114
  IMusicService.Stub class, 164
  keeping running, 149
  lifecycle, 148
  main thread, 149
  Notification object, 156
  notification pull-down, 157
  onBind method, 148
  onClickListener, 164
  onCreate method, 148
  onDestroy method, 149
  onStartCommand method, 148
  setForegroundState method, 155–156
  shutting down, 149
  as singletons, 148
  Start and Stop buttons, 152
  startForeground method, 149
  starting, 152–154
  stopSelf method, 149
setContentView method, 28, 33, 55
setForegroundState method
  music playback, 203
  using, 155–156
SharedPreferences, apply method, 182
signed build
  exporting, 243–244
  keystore file, 244
sound effects, playing, 196–197
Start and Stop buttons, adding to services, 152
StrictMode.enableDefaults, 120

T

tabs, adding to action bars, 235
text view
  customizing, 65–66
  grabbing instance of, 59–60
TextView class, 142–143
TextView ID, creating for activities, 27
thread violations, spotting, 120. *See also* main thread
Toast API, 42
troubleshooting emulator, 18–19
Twitter data, creating for list views, 136–138
Twitter feed
  displaying, 143
  downloading, 143
  parsing, 143
TwitterAsyncTask, 136–138

U

UI (user interface)
- AbsoluteLayout, 78–82
- altering at runtime, 58–60
- changing visibility of views, 61–65
- creating views, 54–58
- customizing views, 65
- drawable folders, 76
- finding resources, 59
- identifying views, 58–59
- layout folders, 74–76
- LinearLayout, 82–90
- RelativeLayout, 90–96
- resource folder, 71–73
- values folder, 73–74
- View class, 54
- ViewGroup, 77–78

USB debugging, enabling, 12
usernames, saving to preferences, 182
<uses> tag, 180
uses-sdkfield, including, 242

V

values folders
- arrays, 73
- colors, 74
- contents of, 71
- creating, 74
- dimensions, 74
- strings, 73
- styles, 74

version, selecting for projects, 15
versioning
- versionCode field, 241
- versionName field, 241

video player, creating, 188–190
VideoView
- adding for movies, 188–189
- extending OnCompletionListener, 189
- implementing onCompletion method, 189
- setting up for, 189–190

view clicks, tracking, 62
ViewGroup
- dip value, 78
- dp value, 78
- match_parent value, 78
- px value, 78
- using with layouts, 77–78
- wrap_content value, 78

views. *See also* custom views; extended views
- assigning IDs, 58–59
- bringing in from XML files, 176
- centering between objects, 95
- changing visibility of, 61
- click listeners, 62
- creating, 54–58
- findViewByID, 58–60
- GONE visibility setting, 61
- identifying, 58–59
- INVISIBLE visibility setting, 61
- keeping, 60
- laying out, 75
- LinearLayouts, 130
- OnClickListener, 62–65
- onCreate method, 63
- retrieving, 59–60
- setVisibility, 61
- VISIBLE visibility setting, 61
- wrapping in <merge> tag, 176
- in XML, 54–55

views in Java, 56–58
- dip value, 57
- dp value, 57

fill_parent value, 57
match_parent value, 57
px value, 57
wrap_content value, 57

W

websites
ActionBar documentation, 236
Android Developers, 4
Eclipse IDE, 4

X

XML (Extensible Markup Language)
setContentView method, 55
views in, 54–55
XML files
bringing in views from, 176
packed binary format, 73
referencing resources in, 73
XML versus Java layouts, 60

Z

Zipalign tool, accessing and using, 245