Beginning Android Programming

DEVELOP AND DESIGN

Kevin Grant
Chris Haseman

Beginning Android Programming

DEVELOP AND DESIGN

Kevin Grant and Chris Haseman

PEACHPIT PRESS
WWW.PEACHPIT.COM

Beginning Android Programming: Develop and Design
Kevin Grant and Chris Haseman

Peachpit Press
www.peachpit.com

To report errors, please send a note to errata@peachpit.com
Peachpit Press is a division of Pearson Education.

Editor: Clifford Colby
Development editor: Robyn Thomas
Production editor: Danielle Foster
Copyeditor: Scout Festa
Technical editors: Matthew Brochstein and Vijay Penemetsa
Cover design: Aren Straiger
Interior design: Mimi Heft
Compositor: Danielle Foster
Indexer: Valerie Haynes Perry

ISBN-13: 978-0-321-95656-9
ISBN-10: 0-321-95656-7

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

To my love, Erica, who's encouraged me to dream bigger than I've ever imagined;
my mother, J'nette, who is my best friend and biggest fan;
and my grandmother, Helene, who always supported me in all of my endeavors.

—Kevin Grant

ACKNOWLEDGMENTS

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

Cliff C. for getting me on board (and always letting me know the weather). Robyn T. for her diligence in keeping us all on time, and deleting all of my superfluous words. Scout F. for her tolerance of my grammar. Matthew B. for helping out while taking care of his new baby boy. Vijay P. for coming in under a tight deadline and working till the end. The mobile team at Tumblr for their encouragement (cleverly disguised as snark). The Android team at Google for building great new tools and making every release feel like a birthday. Most of all, Peachpit for giving me the opportunity to write for you.

ABOUT THE AUTHORS

Kevin Grant is an Android Engineer at Tumblr, a creative blogging platform in New York City, where he focuses on application design, implementing the latest design paradigms, and pushing the boundaries of the Android framework.

He began developing for Android in 2009, performing research at the University of Nevada, Reno. After graduating, he was employed in Malmö, Sweden, where he further honed his mobile skills in the Scandinavian startup scene.

Chris Haseman has been writing mobile software in various forms since 2003. He was involved in several large projects, from MMS messaging to Major League Baseball. More recently, he was an early Android engineer behind the doubleTwist media player and is now the Engineering Manager for the Mobile team at Tumblr. He lives in Manhattan with his wife, Meghan, and constantly debates shaving his beard.

CONTENTS

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.

We don't mean to alarm anyone right off the bat here, but let's 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 you might find yourself running to the Internet for answers. 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.

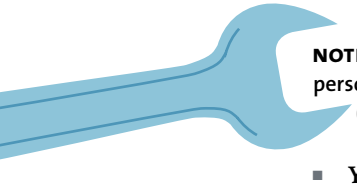
This is where our job comes in. We're here to make the process of learning to write amazing Android software as simple as possible.

Who are we to ask such things of you? Chris Haseman has been writing mobile software in a professional capacity for ten years, and for five of those years, he's been developing software for Android. He's also written code that runs on millions of handsets throughout the world. Also, he has a beard. We all know that people with ample facial hair appear to be more authoritative on all subjects.

Kevin Grant has been developing for Android since its inception and has worked on a breadth of user-facing products, developing beautiful and intuitive interfaces for millions of users. While he doesn't have a beard, we all know that people with a perpetual five o'clock shadow know how to get things done.

From here on out, we're going to take this conversation into the first person. We banter enough amongst ourselves—it's not necessary to confuse you in the process. So without further ado, 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).

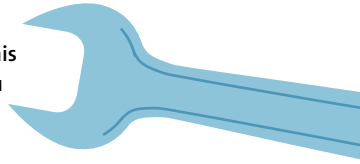
- **You're fluent in Java.** Notice that I say *fluent,* not *expert.* Because you'll be writing usable applications (rather than production libraries, at least to start), I expect you to know the differences between classes and interfaces. You should be able to handle threads and concurrency without 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.

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



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.

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.

WELCOME TO ANDROID

Eclipse and Android Studio are the two supported integrated development environments (IDEs) for Android development, and you need only one to follow along with the examples in this book. There are, however, a few other tools you should be aware of that will be very useful now and in your future work with Android. While you may not use all these tools until you're getting ready to ship an application, it will be helpful to know about them when the need arises.

ECLIPSE (ADT BUNDLE)

Eclipse was the first publicly available IDE for Android and has been in use since 2008. Previous iterations required a complicated setup process that involved downloading multiple pieces and duct-taping them together. Now, with the debut of ADT Bundle, the process is much easier. Everything you need to build an Android application in Eclipse is in one convenient bundle, preconfigured to get you up and running in under five minutes.

ANDROID STUDIO

A spinoff of the popular Java IDE Intellij, Android Studio is Google's newest solution to many of our Android development woes. With Android Studio, Android receives a new unified build system, Gradle, which is fully integrated to allow the utmost flexibility in your development process. It may be a little rough around the edges, and it may take a little extra elbow grease, but you'll find that the time invested will pay off in the long run.

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 that will help you find and diagnose problems and streamline your applications. Whether you use Eclipse or Android Studio, the Android SDK comes preconfigured and is identical for both IDEs.

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.

ANDROID VIRTUAL DEVICE MANAGER

Android Virtual Device Manager is for those developers who prefer to develop on an emulator rather than an actual device. It's a little slow, but you can run an Android emulator for any version of Android, at any screen size. It's perfect for testing screen sizes, screen density, and operating system versions across a plethora of configurations.

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`.

MONITOR

Also known as DDMS (Dalvik Debug Monitor Server), Monitor 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. This tool is very much the Swiss Army knife of your Android toolkit. Along with being a standalone application, both Eclipse and Android Studio users can access this tool from directly within their programs.

GRADLE

This is the new build system in Android Studio. The beauty of Gradle is that whether you press "Build" from within the IDE or build from the command line, you are building with the same system. For general use, there aren't many commands you will need to know, but I cover basic and advanced Gradle usage at the end of the book.

CHAPTER 4

Acquiring Data

Although 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);
      iv.setImageBitmap(bitmap);

   }catch(Exception e){
      Log.e("ImageFetching","Didn't work!",e);
   }
}
```

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.

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

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. Starting in Android Honeycomb (3.0), if you open any network connections on the main thread, your application will crash. If you're hogging too much of the main thread's time with long-running operations, such as calculating pi or finding the next prime number, your application will produce this disastrous dialog box (**Figure 4.1**) on top of your application.

This dialog box 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.

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.

You can also check DDMS and look at the logcat tab. If you are performing network requests on the main thread, you can look for a `NetworkOnMainThreadException`, which should help you identify the location in your code where the error is originating.

WHAT NOT TO DO

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

- Anything involving the network
- Any task requiring a read from or write to the file system
- Heavy processing of any kind (such as image or movie modification)
- Any task that blocks 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 bitmap = 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. I don't want to indent the code into the following page, so I'll show the code beginning 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?).

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 `Image Activity.this` provides a reference 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 can provide implementation.

- 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.
- doInBackground is the meat of this little task sandwich (and is also required). This method is guaranteed by Android to run on a separate background thread. This is where the majority of your work takes place.
- 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, and then we'll go over the specifics of it:

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

   Override
   protected void onPreExecute(){
      //Setup is done here
   }

   @Override
   protected Bitmap doInBackground(String... params) {
      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;
   }
   @Override
   protected void onProgressUpdate(Integer... params){
      //Update a progress bar here, or ignore it, it's up to you
   }
   @Override
   protected void onPostExecute(Bitmap img){
      ImageView iv = (ImageView) findViewById(R.id.remote_image);
      if(iv!=null && img!=null){
         iv.setImageBitmap(img);
      }
   }

   @Override
   protected void onCancelled(){
      // Handle what you want to do if you cancel this task
   }
}
```

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);

   ImageDownloader imageDownloader = new ImageDownloader();
   imageDownloader.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 OF 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. The section "A Few Important Caveats" discusses the "bad things" that can happen.

ONPOSTEXECUTE

The work has been finished, or, as 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 `AsyncTasks` you've started, and when the activity's `onDestroy` method is called, make sure to call `cancel` on any lingering `AsyncTask`.

There are two 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 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.peachpit.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. Your IDE will let you know 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();
   }
}
```

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.

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?

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 `onHandleIntent` 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.peachpit.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.peachpit.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 box 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(TextUtils.isEmpty(location){
         String failedString = "Failed to download image";
         Toast.makeText(context, failedString , Toast.LENGTH_LONG).show();
      }

      File imageFile = new File(location);
      if(!imageFile.exists()){
         pd.dismiss();
```

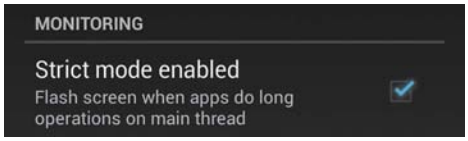


FIGURE 4.2 Developer option for enabling strict mode

```
        String downloadFail = "Unable to Download file :-(";
        Toast.makeText(context, downloadFail, 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 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. The easiest way to accomplish this is by enabling the setting in your developer options (**Figure 4.2**). If you want more fine-grained control of when it's enabled (or you're on a Gingerbread phone), you can, in any activity, call `StrictMode.enableDefaults()`. 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. The `CursorLoader` subclass is great for loading cursors from your application's `ContentProvider`, and for tasks like downloading individual list items for a `ListView`, there is an `AsyncTaskLoader`. Check the documentation for how best to use this new and powerful class.

WRAPPING UP

That about covers 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, it shouldn't crash (ha!), 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 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

NUMBERS

SYMBOL

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

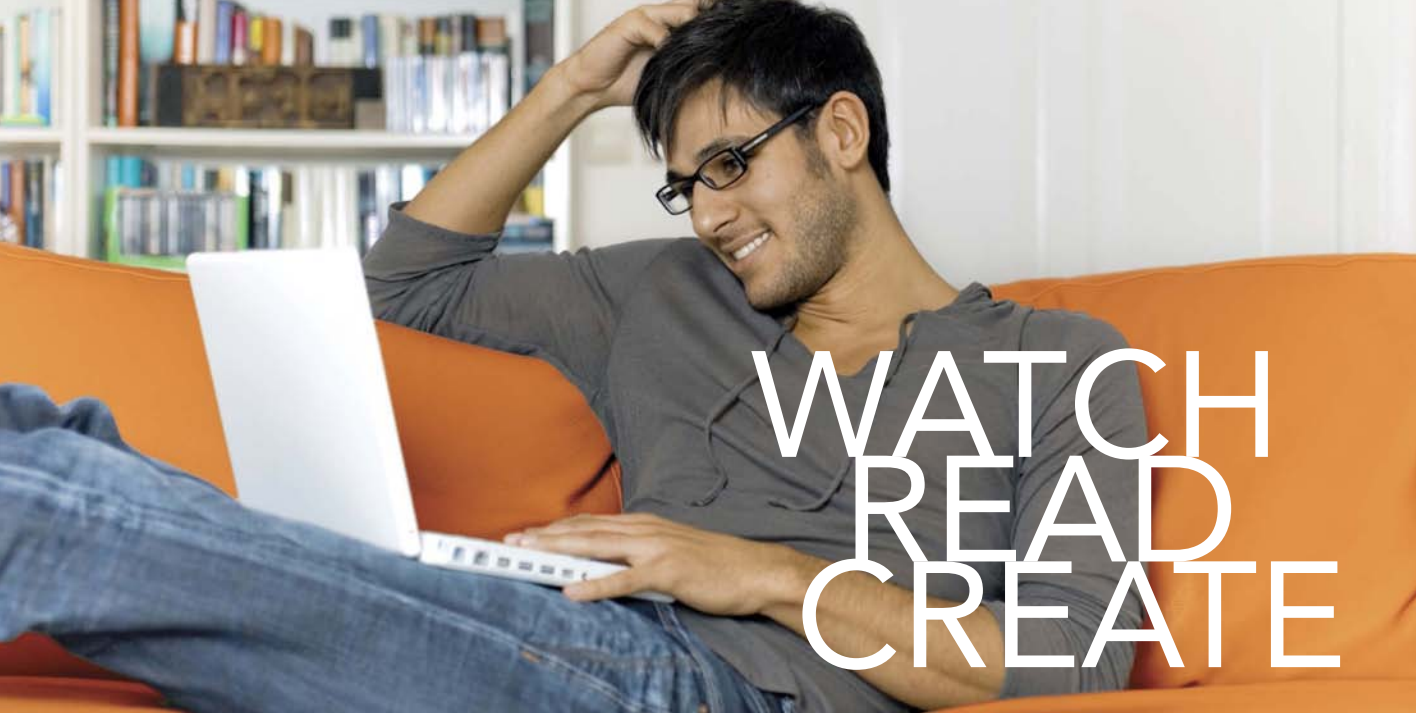
U

V

W

X

This page intentionally left blank



Unlimited online access to all Peachpit, Adobe Press, Apple Training, and New Riders videos and books, as well as content from other leading publishers including: O'Reilly Media, Focal Press, Sams, Que, Total Training, John Wiley & Sons, Course Technology PTR, Class on Demand, VTC, and more.

No time commitment or contract required!
Sign up for one month or a year.
All for $19.99 a month

SIGN UP TODAY

peachpit.com/creativeedge

creative edge