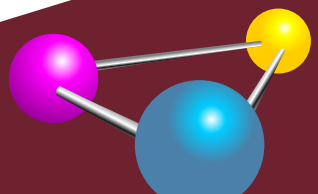
Version 1.0

ebook updates at commonsware.com!

The Busy Coder's Guide to Android Development

Mark L. Murphy



CommonsWare

The Busy Coder's Guide to Android Development

by Mark L. Murphy

The Busy Coder's Guide to Android Development
by Mark L. Murphy

Printing History:
Jul 2008: Version 1.0 ISBN: 978-0-9816780-0-9

Table of Contents

Preface

Welcome to the Book!

Thanks!

Thanks for your interest in developing applications for Android! Increasingly, people will access Internet-based services using so-called "non-traditional" means, such as mobile devices. The more we do in that space now, the more that people will help invest in that space to make it easier to build more powerful mobile applications in the future. Android is new – at the time of this writing, there are no shipping Android-powered devices – but it likely will rapidly grow in importance due to the size and scope of the Open Handset Alliance.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

Prerequisites

If you are interested in programming for Android, you will need at least basic understanding of how to program in Java. Android programming is done using Java syntax, plus a class library that resembles a subset of the Java SE library (plus Android-specific extensions). If you have not programmed in Java before, you probably should quick learn how that works before attempting to dive into programming for Android.

The book does not cover in any detail how to download or install the Android development tools, either the Eclipse IDE flavor or the standalone flavor. The Android Web site covers this quite nicely. The material in the book should be relevant whether you use the IDE or not. You should download, install, and test out the Android development tools from the Android Web site before trying any of the examples listed in this book.

Some chapters may reference material in previous chapters, though usually with a link back to the preceding section of relevance.

Warescription

This book will be published both in print and in digital (ebook) form. The ebook versions of all CommonsWare titles are available via an annual subscription – the Warescription.

The Warescription entitles you, for the duration of your subscription, to ebook forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and Kindle; other ebook formats will be added based on interest and the openness of the format.

Each subscriber gets personalized editions of all editions of each title: both those mirroring printed editions and in-between updates that are only available in ebook form. That way, your ebooks are never out of date for long, and you can take advantage of new material as it is made available instead of having to wait for a whole new print edition. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

From time to time, subscribers will also receive access to subscriber-only online material, both short articles and not-yet-published new titles.

Also, if you own a print copy of a CommonsWare book, and it is in good clean condition with no marks or stickers, you can exchange that copy for a discount off the Warescription price.

If you are interested in a Warescription, visit the Warescription section of the CommonsWare Web site.

Book Bug Bounty

Find a problem in one of our books? Let us know!

Be the first to report a unique concrete problem, and we'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the coupon to a friend, colleague, or some random person you meet on the subway.

By "concrete" problem, we mean things like:

- Typographical errors
- Sample applications that do not work as advertised, in the environment described in the book
- Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Each book has an errata page on the CommonsWare Web site; most known problems will be listed there.

We appreciate hearing about "softer" issues as well, such as:

- Places where you think we are in error, but where we feel our interpretation is reasonable
- Places where you think we could add sample applications, or expand upon the existing material
- Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code License

The source code samples shown in this book are available for download from the CommonsWare Web site. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on **July 1, 2012**. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license, visit the Creative Commons Web site.

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

PART I – Core Concepts

CHAPTER 1

The Big Picture

Android devices, by and large, will be mobile phones. While the Android technology is being discussed for use in other areas (e.g., car dashboard "PCs"), for the most part, you can think of Android as being used on phones.

For developers, this has benefits and drawbacks.

On the plus side, circa 2008, Android-style smartphones are sexy. Offering Internet services over mobile devices dates back to the mid-1990's and the Handheld Device Markup Language (HDML). However, only in recent years have phones capable of Internet access taken off. Now, thanks to trends like text messaging and to products like Apple's iPhone, phones that can serve as Internet access devices are rapidly gaining popularity. So, working on Android applications gives you experience with an interesting technology (Android) in a fast-moving market segment (Internet-enabled phones), which is always a good thing.

The problem comes when you actually have to program the darn things.

Anyone with experience in programming for PDAs or phones has felt the pain of phones simply being *small* in all sorts of dimensions:

- Screens are small (you won't get comments like, "is that a 24-inch LCD in your pocket, or...?")

- Keyboards, if they exist, are small
- Pointing devices, if they exist, are annoying (as anyone who has lost their stylus will tell you) or inexact (large fingers and "multi-touch" LCDs are not a good mix)
- CPU speed and memory are tight compared to desktops and servers you may be used to
- You can have any programming language and development framework you want, so long as it was what the device manufacturer chose and burned into the phone's silicon
- And so on

Moreover, applications running on a phone have to deal with the fact that they're *on a phone*.

People with mobile phones tend to get very irritated when those phones don't work, which is why the "can you hear me now?" ad campaign from Verizon Wireless has been popular for the past few years. Similarly, those same people will get irritated at you if your program "breaks" their phone:

- ...by tying up the CPU such that calls can't be received
- ...by not working properly with the rest of the phone's OS, such that your application doesn't quietly fade to the background when a call comes in or needs to be placed
- ...by crashing the phone's operating system, such as by leaking memory like a sieve

Hence, developing programs for a phone is a different experience than developing desktop applications, Web sites, or back-end server processes. You wind up with different-looking tools, different-behaving frameworks, and "different than you're used to" limitations on what you can do with your program.

What Android tries to do is meet you halfway:

- You get a commonly-used programming language (Java) with some commonly used libraries (e.g., some Apache Commons APIs), with support for tools you may be used to (Eclipse)
- You get a fairly rigid and uncommon framework in which your programs need to run so they can be "good citizens" on the phone and not interfere with other programs or the operation of the phone itself

As you might expect, much of this book deals with that framework and how you write programs that work within its confines and take advantage of its capabilities.

What Androids Are Made Of

When you write a desktop application, you are "master of your own domain". You launch your main window and any child windows – like dialog boxes – that are needed. From your standpoint, you are your own world, leveraging features supported by the operating system, but largely ignorant of any other program that may be running on the computer at the same time. If you do interact with other programs, it is typically through an API, such as using JDBC (or frameworks atop it) to communicate with MySQL or another database.

Android has similar concepts, but packaged differently, and structured to make phones more crash-resistant.

Activities

The building block of the user interface is the **activity**. You can think of an activity as being the Android analogue for the window or dialog in a desktop application.

While it is possible for activities to not have a user interface, most likely your "headless" code will be packaged in the form of content providers or services, described below.

Content Providers

Content providers provide a level of abstraction for any data stored on the device that is accessible by multiple applications. The Android development model encourages you to make your own data available to other applications, as well as your own – building a content provider lets you do that, while maintaining complete control over how your data gets accessed.

Intents

Intents are system messages, running around the inside of the device, notifying applications of various events, from hardware state changes (e.g., an SD card was inserted), to incoming data (e.g., an SMS message arrived), to application events (e.g., your activity was launched from the device's main menu). Not only can you respond to intents, but you can create your own, to launch other activities, or to let you know when specific situations arise (e.g., raise such-and-so intent when the user gets within 100 meters of this-and-such location).

Services

Activities, content providers, and intent receivers are all short-lived and can be shut down at any time. Services, on the other hand, are designed to keep running, if needed, independent of any activity. You might use a service for checking for updates to an RSS feed, or to play back music even if the controlling activity is no longer operating.

Stuff At Your Disposal

Storage

You can package data files with your application, for things that do not change, such as icons or help files. You also can carve out a small bit of space on the device itself, for databases or files containing user-entered or

retrieved data needed by your application. And, if the user supplies bulk storage, like an SD card, you can read and write files on there as needed.

Network

Android devices will generally be Internet-ready, through one communications medium or another. You can take advantage of the Internet access at any level you wish, from raw Java sockets all the way up to a built-in WebKit-based Web browser widget you can embed in your application.

Multimedia

Android devices have the ability to play back and record audio and video. While the specifics may vary from device to device, you can query the device to learn its capabilities and then take advantage of the multimedia capabilities as you see fit, whether that is to play back music, take pictures with the camera, or use the microphone for audio note-taking.

GPS

Android devices will frequently have access to location providers, such as GPS, that can tell your applications where the device is on the face of the Earth. In turn, you can display maps or otherwise take advantage of the location data, such as tracking a device's movements if the device has been stolen.

Phone Services

And, of course, Android devices are typically phones, allowing your software to initiate calls, send and receive SMS messages, and everything else you expect from a modern bit of telephony technology.

CHAPTER 2

Project Structure

The Android build system is organized around a specific directory tree structure for your Android project, much like any other Java project. The specifics, though, are fairly unique to Android and what it all does to prepare the actual application that will run on the device or emulator. Here's a quick primer on the project structure, to help you make sense of it all, particularly for the sample code referenced in this book.

Root Contents

...

The Sweat Off Your Brow

...

And Now, The Rest of the Story

...

What You Get Out Of It

...

CHAPTER 3

Inside the Manifest

The foundation for any Android application is the manifest file: `AndroidManifest.xml` in the root of your project. Here is where you declare what all is inside your application – the activities, the services, and so on. You also indicate how these pieces attach themselves to the overall Android system; for example, you indicate which activity (or activities) should appear on the device's main menu (a.k.a., launcher).

When you create your application, you will get a starter manifest generated for you. For a simple application, offering a single activity and nothing else, the auto-generated manifest will probably work out fine, or perhaps require a few minor modifications. On the other end of the spectrum, the manifest file for the Android API demo suite is over 1,000 lines long. Your production Android applications will probably fall somewhere in the middle.

Most of the interesting bits of the manifest will be described in greater detail in the chapters on their associated Android features. For example, the `service` element will be described in greater detail in the chapter on creating services. For now, we just need to understand what the role of the manifest is and its general overall construction.

In The Beginning, There Was the Root, And It Was Good

...

Permissions, Instrumentations, and Applications (Oh, My!)

...

Your Application Does Something, Right?

...

PART II – Activities

CHAPTER 4

Creating a Skeleton Application

Every programming language or environment book starts off with the ever-popular "Hello, World!" demonstration: just enough of a program to prove you can build things, not so much that you cannot understand what is going on. However, the typical "Hello, World!" program has no interactivity (e.g., just dumps the words to a console), and so is really boring.

This chapter demonstrates a simple project, but one using Advanced Push-Button Technology™ and the current time, to show you how a simple Android activity works.

Begin at the Beginning

To work with anything in Android, you need a project. With ordinary Java, if you wanted, you could just write a program as a single file, compile it with `javac`, and run it with `java`, without any other support structures. Android is more complex, but to help keep it manageable, Google has supplied tools to help create the project. If you are using an Android-enabled IDE, such as Eclipse with the Android plugin, you can create a project inside of the IDE (e.g., select **File** > **New** > **Project**, then choose **Android** > **Android Project**).

If you are using tools that are not Android-enabled, you can use the `activityCreator.py` script, found in the `tools/` directory in your SDK installation. Just pass `activityCreator.py` the package name of the activity

you want to create and a `--out` switch indicating where the project files should be generated. For example:

```
./activityCreator.py --out /path/to/my/project/dir \
com.commonsware.android.Now
```

You will wind up with a handful of pre-generated files, as described in a previous chapter.

For the purposes of the samples shown in this book, you can download their project directories in a ZIP file on the CommonsWare Web site. These projects are ready for use; you do not need to run `activityCreator.py` on those unpacked samples.

The Activity

Your project's `src/` directory contains the standard Java-style tree of directories based upon the Java package you chose when you created the project (e.g., `com.commonsware.android` results in `src/com/commonsware/android/`). Inside the innermost directory you should find a pre-generated source file named `Now.java`, which where your first activity will go.

Open `Now.java` in your editor and paste in the following code:

```
package com.commonsware.android.skeleton;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;

public class Now extends Activity implements View.OnClickListener {
  Button btn;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    btn = new Button(this);
    btn.setOnClickListener(this);
```

```
    updateTime();
    setContentView(btn);
  }

  public void onClick(View view) {
    updateTime();
  }

  private void updateTime() {
    btn.setText(new Date().toString());
  }
}
```

Or, if you download the source files off the Web site, you can just use the `Now` project directly.

Dissecting the Activity

Let's examine this piece by piece:

```
package com.commonsware.android.skeleton;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import java.util.Date;
```

The package declaration needs to be the same as the one you used when creating the project. And, like any other Java project, you need to import any classes you reference. Most of the Android-specific classes are in the `android` package.

Remember that not every Java SE class is available to Android programs! Visit the Android class reference to see what is and is not available.

```
public class Now extends Activity implements View.OnClickListener {
  Button btn;
```

Activities are public classes, inheriting from the `android.Activity` base class. In this case, the activity holds a button (`btn`). Since, for simplicity, we want

to trap all button clicks just within the activity itself, we also have the activity class implement `OnClickListener`.

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);

  btn = new Button(this);
  btn.setOnClickListener(this);
  updateTime();
  setContentView(btn);
}
```

The `onCreate()` method is invoked when the activity is started. The first thing you should do is chain upward to the superclass, so the stock Android activity initialization can be done.

In our implementation, we then create the button instance (`new Button(this)`), tell it to send all button clicks to the activity instance itself (via `setOnClickListener()`), call a private `updateTime()` method (see below), and then set the activity's content view to be the button itself (via `setContentView()`).

We will discuss that magical `Bundle icicle` in a later chapter. For the moment, consider it an opaque handle that all activities receive upon creation.

```
public void onClick(View view) {
  updateTime();
}
```

In Swing, a `JButton` click raises an `ActionEvent`, which is passed to the `ActionListener` configured for the button. In Android, a button click causes `onClick()` to be invoked in the `OnClickListener` instance configured for the button. The listener is provided the view that triggered the click (in this case, the button). All we do here is call that private `updateTime()` method:

```
private void updateTime() {
  btn.setText(new Date().toString());
}
```

When we open the activity (`onCreate()`) or when the button is clicked (`onClick()`), we update the button's label to be the current time via `setText()`, which functions much the same as the `JButton` equivalent.

Building and Running the Activity

To build the activity, either use your IDE's built-in Android packaging tool, or run ant in the base directory of your project. Then, to run the activity:

- Launch the emulator (e.g., run `tools/emulator` from your Android SDK installation)
- Install the package (e.g., run `tools/adb install /path/to/this/example/bin/Now.apk` from your Android SDK installation)
- View the list of installed applications in the emulator and find the "Now" application

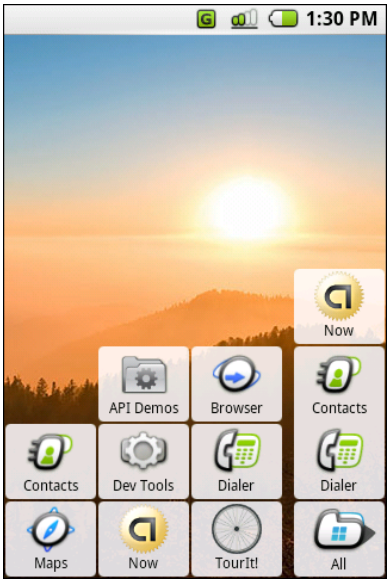


Figure 1. The Android application "launcher"

- Open that application

You should see an activity screen akin to:

Figure 2. The Now demonstration activity

Clicking the button – in other words, pretty much anywhere on the phone's screen – will update the time shown in the button's label.

Note that the label is centered horizontally and vertically, as those are the default styles applied to button captions. We can control that formatting, which will be covered in a later chapter.

After you are done gazing at the awesomeness of Advanced Push-Button Technology™, you can click the back button on the emulator to return to the launcher.

CHAPTER 5

Using XML-Based Layouts

While it is technically possible to create and attach widgets to our activity purely through Java code, the way we did in the preceding chapter, the more common approach is to use an XML-based layout file. Dynamic instantiation of widgets is reserved for more complicated scenarios, where the widgets are not known at compile-time (e.g., populating a column of radio buttons based on data retrieved off the Internet).

With that in mind, it's time to break out the XML and learn out to lay out Android activity views that way.

What Is an XML-Based Layout?

...

Why Use XML-Based Layouts?

...

OK, So What Does It Look Like?

...

What's With the @ Signs?

...

And We Attach These to the Java...How?

...

The Rest of the Story

...

CHAPTER 6

Employing Basic Widgets

Every GUI toolkit has some basic widgets: fields, labels, buttons, etc. Android's toolkit is no different in scope, and the basic widgets will provide a good introduction as to how widgets work in Android activities.

Assigning Labels

...

Button, Button, Who's Got the Button?

...

Fleeting Images

...

Fields of Green. Or Other Colors.

...

Just Another Box to Check

...

Turn the Radio Up

...

It's Quite a View

...

Useful Properties

...

Useful Methods

...

CHAPTER 7

Working with Containers

Containers pour a collection of widgets (and possibly child containers) into specific layouts you like. If you want a form with labels on the left and fields on the right, you will need a container. If you want OK and Cancel buttons to be beneath the rest of the form, next to one another, and flush to right side of the screen, you will need a container. Just from a pure XML perspective, if you have multiple widgets (beyond `RadioButton` widgets in a `RadioGroup`), you will need a container just to have a root element to place the widgets inside.

Most GUI toolkits have some notion of layout management, frequently organized into containers. In Java/Swing, for example, you have layout managers like `BoxLayout` and containers that use them (e.g., `Box`). Some toolkits stick strictly to the box model, such as XUL and Flex, figuring that any desired layout can be achieved through the right combination of nested boxes.

Android, through LinearLayout, also offers a "box" model, but in addition supports a range of containers providing different layout rules. In this chapter, we will look at three commonly-used containers: `LinearLayout` (the box model), `RelativeLayout` (a rule-based model), and `TableLayout` (the grid model), along with `ScrollView`, a container designed to assist with implementing scrolling containers. In the next chapter, we will examine some more esoteric containers.

Thinking Linearly

…

Concepts and Properties

…

Orientation

…

Fill Model

…

Weight

…

Gravity

…

Padding

…

Example

…

All Things Are Relative

…

Concepts and Properties

…

Positions Relative to Container

…

Relative Notation in Properties

…

Positions Relative to Other Widgets

…

Order of Evaluation

…

Example

…

Tabula Rasa

…

Concepts and Properties

…

Putting Cells in Rows

…

Non-Row Children of TableLayout

…

Stretch, Shrink, and Collapse

…

Example

…

Scrollwork

…

CHAPTER 8

Using Selection Widgets

Back in the chapter on basic widgets, you saw how fields could have constraints placed upon them to limit possible input, such as numeric-only or phone-number-only. These sorts of constraints help users "get it right" when entering information, particularly on a mobile device with cramped keyboards.

Of course, the ultimate in constrained input is to select a choice from a set of items, such as the radio buttons seen earlier. Classic UI toolkits have listboxes, comboboxes, drop-down lists, and the like for that very purpose. Android has many of the same sorts of widgets, plus others of particular interest for mobile devices (e.g., the `Gallery` for examining saved photos).

Moreover, Android offers a flexible framework for determining what choices are available in these widgets. Specifically, Android offers a framework of data adapters that provide a common interface to selection lists ranging from static arrays to database contents. Selection views – widgets for presenting lists of choices – are handed an adapter to supply the actual choices.

Adapting to the Circumstances

...

Using ArrayAdapter

…

Other Key Adapters

…

Lists of Naughty and Nice

…

Spin Control

…

Grid Your Lions (Or Something Like That…)

…

Fields: Now With 35% Less Typing!

…

Galleries, Give Or Take The Art

…

CHAPTER 9

Employing Fancy Widgets and Containers

The widgets and containers covered to date are not only found in many GUI toolkits (in one form or fashion), but also are widely used in building GUI applications, whether Web-based, desktop, or mobile. The widgets and containers in this chapter are a little less widely used, though you will likely find many to be quite useful.

Pick and Choose

...

Time Keeps Flowing Like a River

...

Making Progress

...

Putting It On My Tab

…

The Pieces

…

The Idiosyncrasies

…

Wiring It Together

…

Other Containers of Note

…

CHAPTER 10

Applying Menus

Like applications for the desktop and some mobile operating systems, such as PalmOS and Windows Mobile, Android supports activities with "application" menus. Some Android phones will have a dedicated menu key for popping up the menu; others will offer alternate means for triggering the menu to appear.

Also, as with many GUI toolkits, you can create "context menus". On a traditional GUI, this might be triggered by the right-mouse button. On mobile devices, context menus typically appear when the user "taps-and-holds" over a particular widget. For example, if a `TextView` had a context menu, and the device was designed for finger-based touch input, you could push the `TextView` with your finger, hold it for a second or two, and a pop-up menu will appear for the user to choose from.

Where Android differs from most other GUI toolkits is in terms of menu construction. While you can add items to the menu, you do not have full control over the menu's contents, nor the timing of when the menu is built. Part of the menu is system-defined, and that portion is managed by the Android framework itself.

Flavors of Menu

...

Menus of Options

...

Menus in Context

...

Taking a Peek

...

CHAPTER 11

Embedding the WebKit Browser

Other GUI toolkits let you use HTML for presenting information, from limited HTML renderers (e.g., Java/Swing, wxWidgets) to embedding Internet Explorer into .NET applications. Android is much the same, in that you can embed the built-in Web browser as a widget in your own activities, for displaying HTML or full-fledged browsing. The Android browser is based on WebKit, the same engine that powers Apple's Safari Web browser.

The Android browser is sufficiently complex that it gets its own Java package (`android.webkit`), though using the `WebView` widget itself can be simple or powerful, based upon your requirements.

A Browser, Writ Small

...

Loading It Up

...

Navigating the Waters

...

Entertaining the Client

...

Settings, Preferences, and Options (Oh, My!)

...

CHAPTER 12

Showing Pop-Up Messages

Sometimes, your activity (or other piece of Android code) will need to speak up.

Not every interaction with Android users will be neat, tidy, and containable in activities composed of views. Errors will crop up. Background tasks may take way longer than expected. Something asynchronous may occur, such as an incoming message. In these and other cases, you may need to communicate with the user outside the bounds of the traditional user interface.

Of course, this is nothing new. Error messages in the form of dialog boxes have been around for a very long time. More subtle indicators also exist, from task tray icons to bouncing dock icons to a vibrating cell phone.

Android has quite a few systems for letting you alert your users outside the bounds of an `Activity`-based UI. One, notifications, is tied heavily into intents and services and, as such, is covered in a later chapter. In this chapter, you will see two means of raising pop-up messages: toasts and alerts.

Raising Toasts

...

Alert! Alert!

...

Checking Them Out

...

CHAPTER 13

Dealing with Threads

Ideally, you want your activities to be downright snappy, so your users don't feel that your application is sluggish. Responding to user input quickly (e.g., 200ms) is a fine goal. At minimum, though, you need to make sure you respond within 5 seconds, lest the `ActivityManager` decide to play the role of the Grim Reaper and kill off your activity as being non-responsive.

Of course, your activity might have real work to do, which takes non-negligible amounts of time. There are two ways of dealing with this:

1. Do expensive operations in a background service, relying on notifications to prompt users to go back to your activity
2. Do expensive work in a background thread

Android provides a veritable cornucopia of means to set up background threads yet allow them to safely interact with the UI on the UI thread. These include `Handler` objects, posting `Runnable` objects to the `View`, and using `UIThreadUtilities`.

Getting Through the Handlers

...

Messages

...

Runnables

...

Running In Place

...

Utilities (And I Don't Mean Water Works)

...

And Now, The Caveats

...

CHAPTER 14

Handling Activity Lifecycle Events

While this may sound like a broken record...please remember that Android devices, by and large, are phones. As such, some activities are more important that others – taking a call is probably more important to users than is playing Sudoku. And, since it is a phone, it probably has less RAM than does your current desktop or notebook.

As a result, your activity may find itself being killed off because other activities are going on and the system needs your activity's memory. Think of it as the Android equivalent of the "circle of life" – your activity dies so others may live, and so on. You cannot assume that your activity will run until you think it is complete, or even until the user thinks it is complete.

This is one example – perhaps the most important example – of how an activity's lifecycle will affect your own application logic. This chapter covers the various states and callbacks that make up an activity's lifecycle and how you can hook into them appropriately.

Schroedinger's Activity

...

Life, Death, and Your Activity

…

onCreate() and onCompleteThaw()

…

onStart(), onRestart(), and onResume()

…

onPause(), onFreeze(), onStop(), and onDestroy()

…

PART III – Data Stores, Network Services, and APIs

CHAPTER 15

Using Preferences

Android has many different ways for you to store data for long-term use by your activity. The simplest to use is the preferences system.

Android allows activities and applications to keep preferences, in the form of key/value pairs (akin to a `Map`), that will hang around between invocations of an activity. As the name suggests, the primary purpose is for you to store user-specified configuration details, such as the last feed the user looked at in your feed reader, or what sort order to use by default on a list, or whatever. Of course, you can store in the preferences whatever you like, so long as it is keyed by a `String` and has a primitive value (`boolean`, `String`, etc.)

Preferences can either be for a single activity or shared among all activities in an application. Eventually, preferences might be shareable across applications, but that is not supported as of the time of this writing.

Getting What You Want

...

Stating Your Preference

...

A Preference For Action

...

CHAPTER 16

Accessing Files

While Android offers structured storage, via preferences and databases, sometimes a simple file will suffice. Android offers two models for accessing files: one for files pre-packaged with your application, and one for files created on-device by your application.

You And The Horse You Rode In On

...

Readin' 'n Writin'

...

CHAPTER 17

Working with Resources

Resources are static bits of information held outside the Java source code. You have seen one type of resource – the layout – frequently in the examples in this book. There are many other types of resource, such as images and strings, that you can take advantage of in your Android applications.

The Resource Lineup

...

String Theory

...

Plain Strings

...

String Formats

...

Styled Text

…

Styled Formats

…

Got the Picture?

…

XML: The Resource Way

…

Miscellaneous Values

…

Dimensions

…

Colors

…

Arrays

…

Different Strokes for Different Folks

...

CHAPTER 18

Managing and Accessing Local Databases

SQLite is a very popular embedded database, as it combines a clean SQL interface with a very small memory footprint and decent speed. Moreover, it is public domain, so everyone can use it. Lots of firms (Adobe, Apple, Google, Sun, Symbian) and open source projects (Mozilla, PHP, Python) all ship products with SQLite.

For Android, SQLite is "baked into" the Android runtime, so every Android application can create SQLite databases. Since SQLite uses a SQL interface, it is fairly straightforward to use for people with experience in other SQL-based databases. However, its native API is not JDBC, and JDBC might be too much overhead for a memory-limited device like a phone, anyway. Hence, Android programmers have a different API to learn – the good news being is that it is not that difficult.

This chapter will cover the basics of SQLite use in the context of working on Android. It by no means is a thorough coverage of SQLite as a whole. If you want to learn more about SQLite and how to use it in other environment than Android, a fine book is The Definitive Guide to SQLite by Michael Owens.

Activities will typically access a database via a content provider or service. As such, this chapter does not have a full example. You will find a full example

of a content provider that accesses a database in the Building a Content Provider chapter.

A Quick SQLite Primer

...

Start at the Beginning

...

Setting the Table

...

Makin' Data

...

What Goes Around, Comes Around

...

Raw Queries

...

Regular Queries

...

Building with Builders

…

Using Cursors

…

Change for the Sake of Change

…

Making Your Own Cursors

…

Data, Data, Everywhere

…

CHAPTER 19

Leveraging Java Libraries

Java has as many, if not more, third-party libraries than any other modern programming language. Here, "third-party libraries" refer to the innumerable JARs that you can include in a server or desktop Java application – the things that the Java SDKs themselves do not provide.

In the case of Android, the Dalvik VM at its heart is not precisely Java, and what it provides in its SDK is not precisely the same as any traditional Java SDK. That being said, many Java third-party libraries still provide capabilities that Android lacks natively and therefore may be of use to you in your project, for the ones you can get working with Android's flavor of Java.

This chapter explains what it will take for you to leverage such libraries and the limitations on Android's support for arbitrary third-party code.

The Outer Limits

...

Ants and Jars

...

CHAPTER 20

Communicating via the Internet

The expectation is that most, if not all, Android devices will have built-in Internet access. That could be WiFi, cellular data services (EDGE, 3G, etc.), or possibly something else entirely. Regardless, most people – or at least those with a data plan or WiFi access – will be able to get to the Internet from their Android phone.

Not surprisingly, the Android platform gives developers a wide range of ways to make use of this Internet access. Some offer high-level access, such as the integrated WebKit browser component we saw in an earlier chapter. If you want, you can drop all the way down to using raw sockets. Or, in between, you can leverage APIs – both on-device and from 3rd-party JARs – that give you access to specific protocols: HTTP, XMPP, SMTP, and so on.

The emphasis of this book is on the higher-level forms of access: the WebKit component and Internet-access APIs, as busy coders should be trying to reuse existing components versus rolling one's own on-the-wire protocol wherever possible.

REST and Relaxation

...

HTTP Operations via Apache Commons

…

Parsing Responses

…

Stuff To Consider

…

Email over Java

…

PART IV – Intents

CHAPTER 21

Creating Intent Filters

Up to now, the focus of this book has been on activities opened directly by the user from the device's launcher. This, of course, is the most obvious case for getting your activity up and visible to the user. And, in many cases it is the primary way the user will start using your application.

However, remember that the Android system is based upon lots of loosely-coupled components. What you might accomplish in a desktop GUI via dialog boxes, child windows, and the like are mostly supposed to be independent activities. While one activity will be "special", in that it shows up in the launcher, the other activities all need to be reached...somehow.

The "how" is via intents.

An intent is basically a message that you pass to Android saying, "Yo! I want to do...er...something! Yeah!" How specific the "something" is depends on the situation – sometimes you know exactly what you want to do (e.g., open up one of your other activities), and sometimes you don't.

In the abstract, Android is all about intents and receivers of those intents. So, now that we are well-versed in creating activities, let's dive into intents, so we can create more complex applications while simultaneously being "good Android citizens".

What's Your Intent?

…

Pieces of Intents

…

Stock Options

…

Intent Routing

…

Stating Your Intent(ions)

…

Narrow Receivers

…

CHAPTER 22

Launching Activities and Sub-Activities

As discussed previously, the theory behind the Android UI architecture is that developers should decompose their application into distinct activities, each implemented as an `Activity`, each reachable via intents, with one "main" activity being the one launched by the Android launcher. For example, a calendar application could have activities for viewing the calendar, viewing a single event, editing an event (including adding a new one), and so forth.

This, of course, implies that one of your activities has the means to start up another activity. For example, if somebody clicks on an event from the view-calendar activity, you might want to show the view-event activity for that event. This means that, somehow, you need to be able to cause the view-event activity to launch and show a specific event (the one the user clicked upon).

This can be further broken down into two scenarios:

1. You know what activity you want to launch, probably because it is another activity in your own application
2. You have a content `Uri` to...something, and you want your users to be able to do...something with it, but you do not know up front what the options are

This chapter covers the first scenario; the next chapter handles the second.

Peers and Subs

...

Start 'Em Up

...

Make an Intent

...

Make the Call

...

CHAPTER 23

Finding Available Actions via Introspection

Sometimes, you know just what you want to do, such as display one of your other activities.

Sometimes, you have a pretty good idea of what you want to do, such as view the content represented by a `Uri`, or have the user pick a piece of content of some MIME type.

Sometimes, you're lost. All you have is a content `Uri`, and you don't really know what you can do with it.

For example, suppose you were creating a common tagging subsystem for Android, where users could tag pieces of content – contacts, Web URLs, geographic locations, etc. Your subsystem would hold onto the `Uri` of the content plus the associated tags, so other subsystems could, say, ask for all pieces of content referencing some tag.

That's all well and good. However, you probably need some sort of maintenance activity, where users could view all their tags and the pieces of content so tagged. This might even serve as a quasi-bookmark service for items on their phone. The problem is, the user is going to expect to be able to do useful things with the content they find in your subsystem, such as dial a contact or show a map for a location.

The problem is, you have absolutely no idea what is all possible with any given content `Uri`. You probably can view any of them, but can you edit them? Can you dial them? Since new applications with new types of content could be added by any user at any time, you can't even assume you know all possible combinations just by looking at the stock applications shipped on all Android devices.

Fortunately, the Android developers thought of this.

Android offers various means by which you can present to your users a set of likely activities to spawn for a given content `Uri`...even if you have no idea what that content `Uri` really represents. This chapter explores some of these `Uri` action introspection tools.

Pick 'Em

...

Adaptable Adapters

...

Would You Like to See the Menu?

...

Asking Around

...

PART V – Content Providers and Services

CHAPTER 24

Using a Content Provider

Any `Uri` in Android that begins with the `content://` scheme represents a resource served up by a content provider. Content providers offer data encapsulation using `Uri` instances as handles – you neither know nor care where the data represented by the `Uri` comes from, so long as it is available to you when needed. The data could be stored in a SQLite database, or in flat files, or retrieved off a device, or be stored on some far-off server accessed over the Internet.

Given a `Uri`, you can perform basic CRUD (create, read, update, delete) operations using a content provider. `Uri` instances can represent either collections or individual pieces of content. Given a collection `Uri`, you can create new pieces of content via insert operations. Given an instance `Uri`, you can read data represented by the `Uri`, update that data, or delete the instance outright.

Android lets you use existing content providers, plus create your own. This chapter covers using content providers; the next chapter will explain how you can serve up your own data using the content provider framework.

Pieces of Me

...

Getting a Handle

...

Makin' Queries

...

Adapting to the Circumstances

...

Doing It By Hand

...

Position

...

Getting Properties

...

Setting Properties

...

Give and Take

...

Beware of the BLOB!

...

CHAPTER 25

Building a Content Provider

Building a content provider is probably the most complicated and tedious task in all of Android development. There are many requirements of a content provider, in terms of methods to implement and public data members to supply. And, until you try using it, you have no great way of telling if you did any of it correctly (versus, say, building an activity and getting validation errors from the resource compiler).

That being said, building a content provider is of huge importance if your application wishes to make data available to other applications. If your application is keeping its data solely to itself, you may be able to avoid creating a content provider, just accessing the data directly from your activities. But, if you want your data to possibly be used by others – for example, you are building a feed reader and you want other programs to be able to access the feeds you are downloading and caching – then a content provider is right for you.

First, Some Dissection

...

Next, Some Typing

...

Step #1: Create a Provider Class

…

ContentProvider

…

onCreate()

…

query()

…

insert()

…

update()

…

delete()

…

getType()

…

DatabaseContentProvider

...

Step #2: Supply a Uri

...

Step #3: Declare the Properties

...

Step #4: Update the Manifest

...

Notify-On-Change Support

...

CHAPTER 26

Requesting and Requiring Permissions

In the late 1990's, a wave of viruses spread through the Internet, delivered via email, using contact information culled from Microsoft Outlook. A virus would simply email copies of itself to each of the Outlook contacts that had an email address. This was possible because, at the time, Outlook did not take any steps to protect data from programs using the Outlook API, since that API was designed for ordinary developers, not virus authors.

Nowadays, many applications that hold onto contact data secure that data by requiring that a user explicitly grant rights for other programs to access the contact information. Those rights could be granted on a case-by-case basis or a once at install time.

Android is no different, in that it requires permissions for applications to read or write contact data. Android's permission system is useful well beyond contact data, and for content providers and services beyond those supplied by the Android framework.

You, as an Android developer, will frequently need to ensure your applications have the appropriate permissions to do what you want to do with other applications' data. You may also elect to require permissions for other applications to use your data or services, if you make those available to other Android components. This chapter covers how to accomplish both these ends.

Mother, May I?

…

Halt! Who Goes There?

…

Enforcing Permissions via the Manifest

…

Enforcing Permissions Elsewhere

…

May I See Your Documents?

…

CHAPTER 27

Creating a Service

As noted previously, Android services are for long-running processes that may need to keep running even when decoupled from any activity. Examples include playing music even if the "player" activity gets garbage-collected, polling the Internet for RSS/Atom feed updates, and maintaining an online chat connection even if the chat client loses focus due to an incoming phone call.

Services are created when manually started (via an API call) or when some activity tries connecting to the service via inter-process communication (IPC). Services will live until no longer needed and if RAM needs to be reclaimed. Running for a long time isn't without its costs, though, so services need to be careful not to use too much CPU or keep radios active too much of the time, lest the service cause the device's battery to get used up too quickly.

This chapter covers how you can create your own services; the next chapter covers how you can use such services from your activities or other contexts. Both chapters will analyze the MailBuzz sample application (`MailBuzz`), with this chapter focusing mostly on the `MailBuzzService` implementation. `MailBuzzService` polls a supplied email account, either on-demand or on a stated interval, to see if new messages have arrived, at which it will post a `Notification` (as described in the chapter on notifications).

Getting Buzzed

...

Service with Class

...

When IPC Attacks!

...

Write the AIDL

...

Implement the Interface

...

Manifest Destiny

...

Where's the Remote?

...

CHAPTER 28

Invoking a Service

Services can be used by any application component that "hangs around" for a reasonable period of time. This includes activities, content providers, and other services. Notably, it does not include pure intent receivers (i.e., intent receivers that are not part of an activity), since those will get garbage collected immediately after each instance processes one incoming `Intent`.

To use a service, you need to get an instance of the AIDL interface for the service, then call methods on that interface as if it were a local object. When done, you can release the interface, indicating you no longer need the service.

In this chapter, we will look at the client side of the MailBuzz sample application (`MailBuzz`). The MailBuzz activity provides fields for the account information (server type, server, etc.), a checkbox to toggle whether polling for new mail should go on, a button to push the account information to the service, and another button to check right now for new messages.

When run, the activity looks like this:

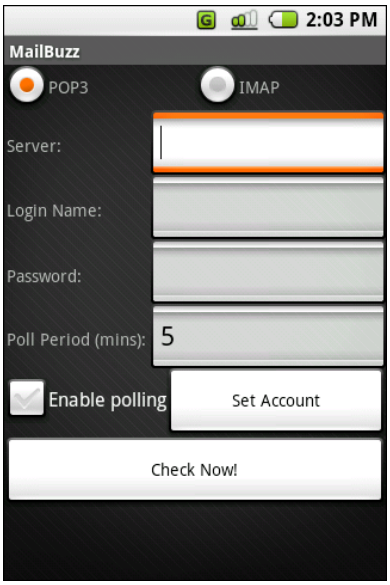


Figure 3. The MailBuzz service client

Bound for Success

...

Request for Service

...

Prometheus Unbound

...

Manual Transmission

...

CHAPTER 29

Alerting Users Via Notifications

Pop-up messages. Tray icons and their associated "bubble" messages. Bouncing dock icons. You are no doubt used to programs trying to get your attention, sometimes for good reason.

Your phone also probably chirps at you for more than just incoming calls: low battery, alarm clocks, appointment notifications, incoming text message or email, etc.

Not surprisingly, Android has a whole framework for dealing with these sorts of things, collectively called "notifications".

Types of Pestering

...

Hardware Notifications

...

Icons

...

Letting Your Presence Be Felt

...

PART VI – Other Android Capabilities

CHAPTER 30

Accessing Location-Based Services

A popular feature on current-era mobile devices is GPS capability, so the device can tell you where you are at any point in time. While the most popular use of GPS service is mapping and directions, there are other things you can do if you know your location. For example, you might set up a dynamic chat application where the people you can chat with are based on physical location, so you're chatting with those you are nearest. Or, you could automatically "geotag" posts to Twitter or similar services.

GPS is not the only way a mobile device can identify your location. Alternatives include:

- The European equivalent to GPS, called Galileo, which is still under development at the time of this writing
- Cell tower triangulation, where your position is determined based on signal strength to nearby cell towers
- Proximity to public WiFi "hotspots" that have known geographic locations

Android devices may have one or more of these services available to them. You, as a developer, can ask the device for your location, plus details on what providers are available. There are even ways for you to simulate your location in the emulator, for use in testing your location-enabled applications.

Location Providers: They Know Where You're Hiding

...

Finding Yourself

...

On the Move

...

Are We There Yet? Are We There Yet? Are We There Yet?

...

Testing...Testing...

...

CHAPTER 31

Mapping with MapView and MapActivity

One of Google's most popular services – after search, of course – is Google Maps, where you can find everything from the nearest pizza parlor to directions from New York City to San Francisco (only 2,905 miles!) to street views and satellite imagery.

Android, not surprisingly, integrates Google Maps. There is a mapping activity available to users straight off the main Android launcher. More relevant to you, as a developer, are `MapView` and `MapActivity`, which allow you to integrate maps into your own applications. Not only can you display maps, control the zoom level, and allow people to pan around, but you can tie in Android's location-based services to show where the device is and where it is going.

Fortunately, integrating basic mapping features into your Android project is fairly easy. However, there is a fair bit of power available to you, if you want to get fancy.

The Bare Bones

...

Exercising Your Control

…

Zoom

…

Center

…

Reticle

…

Traffic and Terrain

…

Follow You, Follow Me

…

Layers Upon Layers

…

Overlay Classes

…

Drawing the Overlay

…

Handling Screen Taps

…

CHAPTER 32

Playing Media

Pretty much every phone claiming to be a "smartphone" has the ability to at least play back music, if not video. Even many more ordinary phones are full-fledged MP3 players, in addition to offering ringtones and whatnot.

Not surprisingly, Android aims to match the best of them.

Android has full capability to play back and record audio and video. This includes:

- Playback of audio, such as downloaded MP3 tracks
- Showing photos
- Playing back video clips
- Voice recording through the microphone
- Camera for still pictures or video clips

Exactly how robust these capabilities will be is heavily device-dependent. Mobile device cameras range from excellent to atrocious. Screen resolutions and sizes will vary, and video playback works better on better screens. Which codecs a device manufacturer will license (e.g., what types of video can it play?) and which Bluetooth profiles a device will support (e.g., A2DP for stereo?) will also have an impact on what results any given person will have with their phone.

You as a developer can integrate media playback and recording into your applications. Recording is outside the scope of this book, in large part because the current emulator has recording limitations at this time. And, viewing pictures is mostly a matter of putting an `ImageView` widget into an activity. This chapter, therefore, focuses on playback of audio and video.

As with many advanced Android features, expect changes in future releases of their toolkit. For example, at the time of this writing, there is no built-in audio or video playback activity. Hence, you cannot just craft an `Intent` to, say, an MP3 URL, and hand it off to Android with `VIEW_ACTION` to initiate playback. Right now, you need to handle the playback yourself. It is probably safe to assume, though, that standard activities for this will be forthcoming, allowing you to "take the easy way out" if you want to play back media but do not need to control that playback much yourself.

Get Your Media On

...

Making Noise

...

Moving Pictures

...

CHAPTER 33

Handling Telephone Calls

Many, if not most, Android devices will be phones. As such, not only will users be expecting to place and receive calls using Android, but you will have the opportunity to help them place calls, if you wish.

Why might you want to?

- Maybe you are writing an Android interface to a sales management application (a la Salesforce.com) and you want to offer users the ability to call prospects with a single button click, and without them having to keep those contacts both in your application and in the phone's contacts application
- Maybe you are writing a social networking application, and the roster of phone numbers that you can access shifts constantly, so rather than try to "sync" the social network contacts with the phone's contact database, you let people place calls directly from your application
- Maybe you are creating an alternative interface to the existing contacts system, perhaps for users with reduced motor control (e.g., the elderly), sporting big buttons and the like to make it easier for them to place calls

Whatever the reason, Android has APIs to let you manipulate the phone just like any other piece of the Android system.

No, No, No – Not That IPhone...

...

What's Our Status?

...

You Make the Call!

...

CHAPTER 34

Searching with SearchManager

One of the firms behind the Open Handset Alliance – Google – has a teeny weeny Web search service, one you might have heard of in passing. Given that, it's not surprising that Android has some amount of built-in search capabilities.

Specifically, Android has "baked in" the notion of searching not only on the device for data, but over the air to Internet sources of data.

Your applications can participate in the search process, by triggering searches or perhaps by allowing your application's data to be searched.

Note that this is fairly new to the Android platform, and so some shifting in the APIs is likely. Stay tuned for updates to this chapter.

Hunting Season

...

Search Yourself

...

Craft the Search Activity

…

Update the Manifest

…

Try It Out

…

PART VII – Appendices

APPENDIX A

The TourIt Sample Application

In several chapters of this book, we used TourIt as a source of sample code for features ranging from content providers to mapping and location services. This appendix discusses the application as a whole, so you can see all facets of it from front to back.

Installing TourIt

...

Demo Location Provider

...

SD Card Image with Sample Tour

...

Running TourIt

...

Main Activity

…

Configuration Activity

…

Cue Sheet Activity

…

Map Activity

…

Tour Update Activity

…

Help Activity

…

TourIt's Manifest

…

TourIt's Content

…

Data Storage

…

Content Provider

…

Model Classes

…

TourIt's Activities

…

TourListActivity

…

TourViewActivity

…

Custom List Contents

…

Details Panel

…

TourMapActivity

…

TourEditActivity

…

HelpActivity

…

ConfigActivity

…

```
  String page=getIntent().getStringExtra(PAGE);

  if (page==null) {
    browser.loadUrl("file:///android_asset/index.html");
  }
  else {
    browser.loadUrl("file:///android_asset/"+page+".html");
  }
}
```

By default, it will load the home page. If, however, the activity was started by another activity that passed in a specific page to view, it loads that page instead.

`HelpActivity` hooks into the WebKit browser to detect clicks on links. Since the only links in the help are to other help pages, it simply loads in the requested page:

```
private class Callback extends WebViewClient {
  public boolean shouldOverrideUrlLoading(WebView view, String url) {
    view.loadUrl(url);

    return(true);
  }
}
```

ConfigActivity

The `ConfigActivity` class mostly loads data out of preferences, updates the layout's widgets to match, then reverses the process when the activity is paused (e.g., when the user clicks Close from the options menu).

The most interesting thing here is the spinner of location providers – this is covered in detail in the chapter on location services.

Keyword Index

Class

Command

Constant

Method

Property