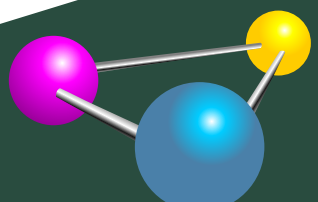
Version 0.7

digital updates at commonsware.com!

The Busy Coder's Guide to *Advanced* Android Development

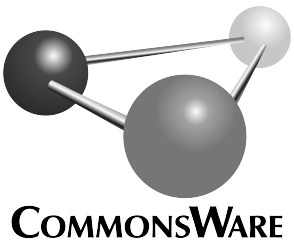
Mark L. Murphy



CommonsWare

The Busy Coder's Guide to Advanced Android Development

by Mark L. Murphy



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

Printing History:
Mar 2009: Version 0.6 ISBN: 978-0-9816780-1-6

Table of Contents

Welcome to the Warescription! ..... **ix**

Preface ..... **xi**

Welcome to the Book! xi

Prerequisites xi

Warescription xii

Book Bug Bounty xiii

Source Code License xiv

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

Lifecycle of a CommonsWare Book xv

WebView, Inside and Out **1**

Friends with Benefits 1

Turnabout is Fair Play 6

Gearing Up 9

Back To The Future 11

Crafting Your Own Views **13**

Providing Attribution 13

Tailor Your Buttons 13

The Department of State 14

Spin It Your Way 14

More Fun With ListViews....15
Giant Economy-Size Dividers....15
Choosing What Is Selectable....16
Composition for Sections....17
From Head To Toe....24
Control Your Selection....26
Create a Unified Row View....27
Configure the List, Get Control on Selection....28
Change the Row....30
Creating Drawables....35
Traversing Along a Gradient....35
A Stitch In Time Saves Nine....39
The Name and the Border....39
Padding and the Box....40
Stretchable Zones....40
Tooling....41
Using Nine-Patch Images....42
Animating Widgets....45
It's Not Just For Toons Anymore....45
A Quirky Translation....46
Mechanics of Translation....46
Imagining a Drawer....47
The Aftermath....47
Introducing DrawerLayout....48
Using the Animation....50
Fading To Black. Or Some Other Color....50
Alpha Numbers....51

Animations in XML....51
Using XML Animations....52
When It's All Said And Done....52
Hit The Accelerator....53
Animate. Set. Match....54
A Chest of Drawers....55
Playing Media....57
Get Your Media On....57
Making Noise....58
Moving Pictures....63
Using the Camera....69
Sneaking a Peek....69
The Permission....70
The SurfaceView....71
The Camera....71
Image Is Everything....73
Asking for a Format....74
Connecting the Camera Button....74
Taking a Picture....75
The Job Queue Pattern....76
Saving the Picture....77
Sensors....81
The Sixth Sense. Or Possibly the Seventh....81
Orienting Yourself....82
Steering Your Phone....85
Do "The Shake"....87

Databases and Content Providers.....93
Distributed Data.....94
SQLite: On-Device, On-Desktop.....95
Exporting a Database.....95
Loading the Exported Database.....98
Examining Your Relationships.....101
Contact Permissions.....101
Pre-Joined Data.....102
The Sample Activity.....102
Accessing People.....105
Accessing Phone Numbers.....106
Accessing Email Addresses.....107
Rummaging Through Your Phone Records.....107
Come Together, Right Now.....108
CursorWrapper.....109
Implementing a JoinCursor.....110
Using a JoinCursor.....114
Using System Services.....121
Get Alarmed.....121
Meeting the User's Preference.....121
Get Set.....121
Handling System Events.....123
Get Moving, First Thing.....123
I Sense a Connection Between Us.....123
Feeling Drained.....123
Your Own (Advanced) Services.....125
Service From Afar.....125

Service Names....126
The Service....127
The Client....129
Servicing the Service....131
Callbacks via AIDL....132
Revising the Client....133
Revising the Service....134
Reusable Components....141
Pick Up a JAR....141
The Android Build Process....142
Integrating JARs into Android....143
Putting Limits on the JAR....143
Crafting an Android-Aware JAR....144
An API with Intent....145
Sending Data in the Intent....145
Callbacks As Intents....145
Serving Your Fellow Bits....145
Pros, Cons, and Other Forms of Navel-Gazing....145
Richness of API....146
Code Duplication....146
Ease of Initial Deployment....147
Intended Form of Integration....147
Testing Your Code....149
Testing Your Instrument....149
Something Incompletely Different....149
Production Applications....151
Making Your Mark....151

To Market, To Market.....151
Wide Distribution.....151
Click Here To Download.....151
Let Your Fingers Do the Distributing.....151
Late-Breaking Updates.....152

Welcome to the Warescription!

We hope you enjoy this digital book and its updates – keep tabs on the Warescription feed off the CommonsWare site to learn when new editions of this book, or other books in your Warescription, are available.

Each Warescription digital book is licensed for the exclusive use of its subscriber and is tagged with the subscribers name. We ask that you not distribute these books. If you work for a firm and wish to have several employees have access, enterprise Warescriptions are available. Just contact us at enterprise@commonsware.com.

Also, bear in mind that eventually this edition of this title will be released under a Creative Commons license – more on this in the preface.

Remember that the CommonsWare Web site has errata and resources (e.g., source code) for each of our titles. Just visit the Web page for the book you are interested in and follow the links.

Some notes for Kindle users:

- You may wish to drop your font size to level 2 for easier reading
- Source code listings are incorporated as graphics so as to retain the monospace font, though this means the source code listings do not honor changes in Kindle font size

Preface

Welcome to the Book!

If you come to this book after having read its companion volume, The Busy Coder's Guide to Android Development, thanks for sticking with the series! CommonsWare aims to have the most comprehensive set of Android development resources (outside of the Open Handset Alliance itself), and we appreciate your interest.

If you come to this book having learned about Android from other sources, thanks for joining the CommonsWare community! Android, while aimed at small devices, is a surprisingly vast platform, making it difficult for any given book, training, wiki, or other source to completely cover everything one needs to know. This book will hopefully augment your knowledge of the ins and outs of Android-dom and make it easier for you to create "killer apps" that use the Android platform.

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

Prerequisites

This book assumes you have experience in Android development, whether from a CommonsWare resource or someplace else. In other words, you should have:

- A working Android development environment, whether it is based on Eclipse, another IDE, or just the command-line tools that accompany the Android SDK
- A strong understanding of how to create activities and the various stock widgets available in Android
- A working knowledge of the `Intent` system, how it serves as a message bus, and how to use it to launch other activities
- Experience in creating, or at least using, content providers and services

If you picked this book up expecting to learn those topics, you really need another source first, since this book focuses on other topics. While we are fans of The Busy Coder's Guide to Android Development, there are plenty of other books available covering the Android basics, blog posts, wikis, and, of course, the main Android site itself. A list of currently-available Android books can be found on the Android Programming knol.

Some chapters may reference material in previous chapters, though usually with a link back to the preceding section of relevance. Many chapters will reference material in The Busy Coder's Guide to Android Development, usually via the shorthand *BCG to Android* moniker.

Warescription

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

The Warescription entitles you, for the duration of your subscription, to digital forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and Kindle; other digital 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 digital form. That way, your digital books 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, including 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 free four-month Warescription.

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 in the current digital edition, 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. One coupon is given per email containing valid bug reports.

NOTE: Books with version numbers lower than 0.9 are ineligible for the bounty program, as they are in various stages of completion. We appreciate bug reports, though, if you choose to share them with us.

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 December 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.

Lifecycle of a CommonsWare Book

CommonsWare books generally go through a series of stages.

First are the pre-release editions. These will have version numbers below 0.9 (e.g., 0.2). These editions are incomplete, often times having but a few chapters to go along with outlines and notes. However, we make them available to those on the Warescription so they can get early access to the material.

Release candidates are editions with version numbers ending in ".9" (0.9, 1.9, etc.). These editions should be complete. Once again, they are made available to those on the Warescription so they get early access to the material and can file bug reports (and receive bounties in return!).

Major editions are those with version numbers ending in ".0" (1.0, 2.0, etc.). These will be first published digitally for the Warescription members, but will shortly thereafter be available in print from booksellers worldwide.

Versions between a major edition and the next release candidate (e.g., 1.1, 1.2) will contain bug fixes plus new material. Each of these editions should also be complete, in that you will not see any "TBD" (to be done) markers or the like. However, these editions may have bugs, and so bug reports are eligible for the bounty program, as with release candidates and major releases.

A book usually will progress fairly rapidly through the pre-release editions to the first release candidate and Version 1.0 – often times, only a few months. Depending on the book's scope, it may go through another cycle of significant improvement (versions 1.1 through 2.0), though this may take several months to a year or more. Eventually, though, the book will go into more of a "maintenance mode", only getting updates to fix bugs and deal with major ecosystem events – for example, a new release of the Android SDK will necessitate an update to all Android books.

PART I – Advanced Widgets

CHAPTER 1

WebView, Inside and Out

Android uses the WebKit browser engine as the foundation for both its Browser application and the `WebView` embeddable browsing widget. The Browser application, of course, is something Android users can interact with directly; the `WebView` widget is something you can integrate into your own applications for places where an HTML interface might be useful.

In *BCG to Android*, we saw a simple integration of a `WebView` into an Android activity, with the activity dictating what the browsing widget displayed and how it responded to links.

Here, we will expand on this theme, and show how to more tightly integrate the Java environment of an Android application with the Javascript environment of WebKit.

Friends with Benefits

When you integrate a `WebView` into your activity, you can control what Web pages are displayed, whether they are from a local provider or come from over the Internet, what should happen when a link is clicked, and so forth. And between `WebView`, `WebViewClient`, and `WebSettings`, you can control a fair bit about how the embedded browser behaves. Yet, by default, the browser itself is just a browser, capable of showing Web pages and interacting with Web sites, but otherwise gaining nothing from being hosted by an Android application.

Except for one thing: `addJavascriptInterface()`.

The `addJavascriptInterface()` method on `WebView` allows you to inject a Java object into the `WebView`, exposing its methods, so they can be called by Javascript loaded by the Web content in the `WebView` itself.

Now you have the power to provide access to a wide range of Android features and capabilities to your `WebView`-hosted content. If you can access it from your activity, and if you can wrap it in something convenient for use by Javascript, your Web pages can access it as well.

For example, Google's Gears project offers a Geolocation API, so Web pages loaded in a Gears-enabled browser can find out where the browser is located. This information could be used for everything from fine-tuning a search to emphasize local content to serving up locale-tailored advertising.

We can do much of the same thing with Android and `addJavascriptInterface()`.

In the `WebView/GeoWeb1` project, you will find a fairly simple layout (`main.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <WebView android:id="@+id/webkit"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  />
</LinearLayout>
```

All this does is host a full-screen `WebView` widget.

Next, take a look at the `GeoWebOne` activity class:

```
public class GeoWebOne extends Activity {
  private static String PROVIDER=LocationManager.GPS_PROVIDER;
  private WebView browser;
```

```
  private LocationManager myLocationManager=null;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    myLocationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE
);

    browser.getSettings().setJavaScriptEnabled(true);
    browser.addJavascriptInterface(new Locater(), "locater");
    browser.loadUrl("file:///android_asset/geoweb1.html");
  }

  @Override
  public void onResume() {
    super.onResume();
    myLocationManager.requestLocationUpdates(PROVIDER, 10000,
                                             100.0f,
                                             onLocationChange);
  }

  @Override
  public void onPause() {
    super.onPause();
    myLocationManager.removeUpdates(onLocationChange);
  }

  LocationListener onLocationChange=new LocationListener() {
    public void onLocationChanged(Location location) {
      // ignore...for now
    }

    public void onProviderDisabled(String provider) {
      // required for interface, not used
    }

    public void onProviderEnabled(String provider) {
      // required for interface, not used
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras) {
      // required for interface, not used
    }
  };

  public class Locater {
    public double getLatitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
```

```
        return(0);
      }

      return(loc.getLatitude());
    }

    public double getLongitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(0);
      }

      return(loc.getLongitude());
    }
  }
}
```

This looks a bit like some of the `WebView` examples in the *BCG to Android*'s chapter on integrating WebKit. However, it adds three key bits of code:

1. It sets up the `LocationManager` to provide updates when the device position changes, routing those updates to a do-nothing `LocationListener` callback object
2. It has a `Locater` inner class that provides a convenient API for accessing the current location, in the form of latitude and longitude values
3. It uses `addJavascriptInterface()` to expose a `Locater` instance under the name `locater` to the Web content loaded in the `WebView`

The Web page itself is referenced in the source code as `file:///android_asset/geoweb1.html`, so the `GeoWeb1` project has a corresponding `assets/` directory containing `geoweb1.html`:

```
<html>
<head>
<title>Android GeoWebOne Demo</title>
<script language="javascript">
  function whereami() {
    document.getElementById("lat").innerHTML=locater.getLatitude();
    document.getElementById("lon").innerHTML=locater.getLongitude();
  }
</script>
</head>
<body>
<p>
```

```
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="whereami()">Update Location</a></p>
</body>
</html>
```

When you click the "Update Location" link, the page calls a `whereami()` Javascript function, which in turn uses the `locater` object to update the latitude and longitude, initially shown as "(unknown)" on the page.

If you run the application, initially, the page is pretty boring:

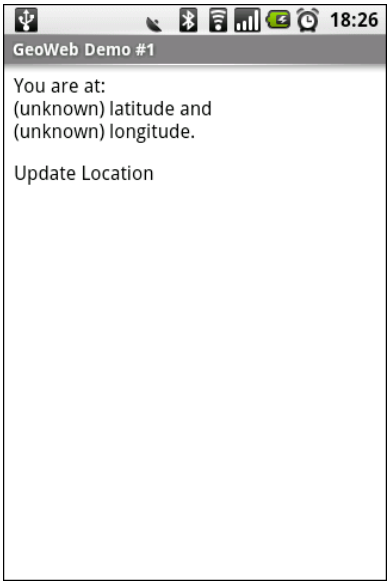


Figure 1. The GeoWebOne sample application, as initially launched

However, if you wait a bit for a GPS fix, and click the "Update Location" link...the page is still pretty boring, but it at least knows where you are:

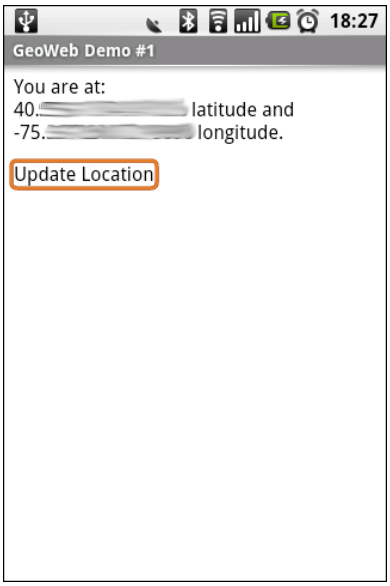


Figure 2. The GeoWebOne sample application, after clicking the Update Location link

Turnabout is Fair Play

Now that we have seen how Javascript can call into Java, it would be nice if Java could somehow call out to Javascript. In our example, it would be helpful if we could expose automatic location updates to the Web page, so it could proactively update the position as the user moves, rather than wait for a click on the "Update Location" link.

Well, as luck would have it, we can do that too. This is a good thing, otherwise, this would be a really weak section of the book.

What is unusual is how you call out to Javascript. One might imagine there would be an `executeJavascript()` counterpart to `addJavascriptInterface()`, where you could supply some Javascript source and have it executed within the context of the currently-loaded Web page.

Oddly enough, that is not how this is accomplished.

Instead, given your snippet of Javascript source to execute, you call `loadUrl()` on your `WebView`, as if you were going to load a Web page, but you put `javascript:` in front of your code and use that as the "address" to load.

If you have ever created a "bookmarklet" for a desktop Web browser, you will recognize this technique as being the Android analogue – the `javascript:` prefix tells the browser to treat the rest of the address as Javascript source, injected into the currently-viewed Web page.

So, armed with this capability, let us modify the previous example to continuously update our position on the Web page.

The layout for this new project (`WebView/GeoWeb2`) is the same as before. The Java source for our activity changes a bit:

```
public class GeoWebTwo extends Activity {
  private static String PROVIDER="gps";
  private WebView browser;
  private LocationManager myLocationManager=null;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    myLocationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE
);

    browser.getSettings().setJavaScriptEnabled(true);
    browser.addJavascriptInterface(new Locater(), "locater");
    browser.loadUrl("file:///android_asset/geoweb2.html");
  }

  @Override
  public void onResume() {
    super.onResume();
    myLocationManager.requestLocationUpdates(PROVIDER, 10000,
                                             100.0f,
                                             onLocationChange);
  }

  @Override
  public void onPause() {
    super.onPause();
    myLocationManager.removeUpdates(onLocationChange);
  }
```

```
  LocationListener onLocationChange=new LocationListener() {
    public void onLocationChanged(Location location) {
      StringBuilder buf=new StringBuilder("javascript:whereami(");

      buf.append(String.valueOf(location.getLatitude()));
      buf.append(",");
      buf.append(String.valueOf(location.getLongitude()));
      buf.append(")");

      browser.loadUrl(buf.toString());
    }

    public void onProviderDisabled(String provider) {
      // required for interface, not used
    }

    public void onProviderEnabled(String provider) {
      // required for interface, not used
    }

    public void onStatusChanged(String provider, int status,
                                Bundle extras) {
      // required for interface, not used
    }
  };

  public class Locater {
    public double getLatitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(0);
      }

      return(loc.getLatitude());
    }

    public double getLongitude() {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

      if (loc==null) {
        return(0);
      }

      return(loc.getLongitude());
    }
  }
}
```

Before, the `onLocationChanged()` method of our `LocationListener` callback did nothing. Now, it builds up a call to a `whereami()` Javascript function, providing the latitude and longitude as parameters to that call. So, for

example, if our location were 40 degrees latitude and -75 degrees longitude, the call would be `whereami(40,-75)`. Then, it puts `javascript:` in front of it and calls `loadUrl()` on the `WebView`. The result is that a `whereami()` function in the Web page gets called with the new location.

That Web page, of course, also needed a slight revision, to accommodate the option of having the position be passed in:

```
<html>
<head>
<title>Android GeoWebTwo Demo</title>
<script language="javascript">
  function whereami(lat, lon) {
    document.getElementById("lat").innerHTML=lat;
    document.getElementById("lon").innerHTML=lon;
  }
</script>
</head>
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="whereami(locater.getLatitude(), locater.getLongitude())">
Update Location</a></p>
</body>
</html>
```

The basics are the same, and we can even keep our "Update Location" link, albeit with a slightly different `onClick` attribute.

If you build, install, and run this revised sample on a GPS-equipped Android device, the page will initially display with "(unknown)" for the current position. After a fix is ready, though, the page will automatically update to reflect your actual position. And, as before, you can always click "Update Location" if you wish.

Gearing Up

In these examples, we demonstrate how WebView can interact with Java code, code that provides a service a little like one of those from Gears.

Of course, what would be really slick is if we could use Gears itself.

The good news is that Android is close on that front. Gears is actually baked into Android. However, it is only exposed by the Browser application, not via `WebView`. So, an end user of an Android device can leverage Gears-enabled Web pages.

For example, you could load the Geolocation sample application in your Android device's Browser application. Initially, you will get the standard "can we please use Gears?" security prompt:

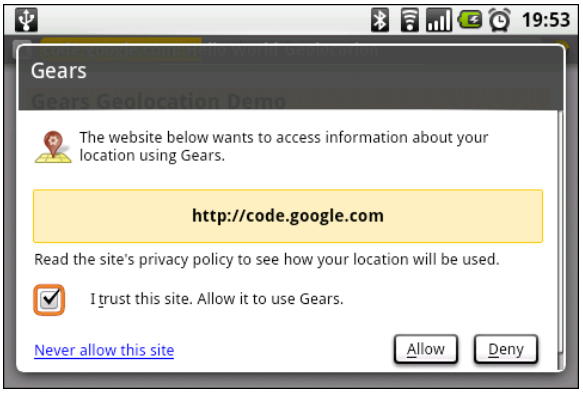


Figure 3. The Gears security prompt

Then, Gears will fire up the GPS interface (if enabled) and will fetch your location:

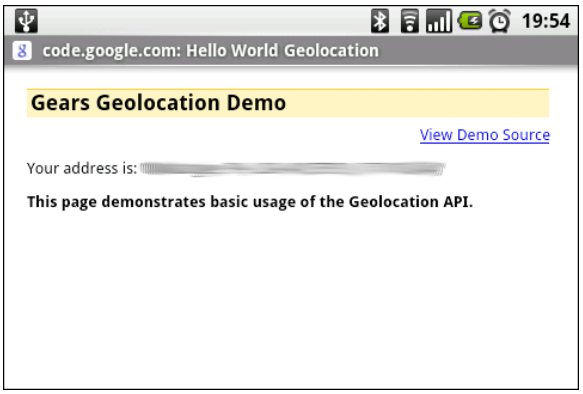


Figure 4. The Gears Geolocation sample application

Back To The Future

The core Android team has indicated that these sorts of capabilities will increase in future editions of the Android operating system. This could include support for more types of plugins, a richer Java-Javascript bridge, and so on.

You can also expect some improvements coming from the overall Android ecosystem. For example, the PhoneGap project is attempting to build a framework that supports creating Android applications solely out of Web content, using WebView as the front-end, supporting a range of Gears-like capabilities and more, such as accelerometer awareness.

CHAPTER 2

Crafting Your Own Views

One of the classic forms of code reuse is the GUI widget. Since the advent of Microsoft Windows – and, to some extent, even earlier – developers have been creating their own widgets to extend an existing widget set. These range from 16-bit Windows "custom controls" to 32-bit Windows OCX components to the innumerable widgets available for Java Swing and SWT, and beyond.

Android has facilities for creating packaged widgets as well, both for reuse within your own applications and for wider distribution. Later in this book, we will discuss how to distribute such reusable code. Here, though, we focus on the widgets themselves, covering some of the techniques for extending existing widgets in new and exciting ways, yet leaving them packaged much like an existing widget.

Providing Attribution

Tailor Your Buttons

The Department of State

Spin It Your Way

CHAPTER 3

More Fun With ListViews

One of the most important widgets in your toolbelt is the `ListView`. Some activities are purely a `ListView`, to allow the user to sift through a few choices...or perhaps a few thousand. We already saw in *The Busy Coder's Guide to Android Development* how to create "fancy `ListViews`", where you have complete control over the list rows themselves. In this chapter, we will cover some additional techniques you can use to make your `ListView` widgets be pleasant for your users to work with.

Giant Economy-Size Dividers

You may have noticed that the preference UI has what behaves a lot like a `ListView`, but with a curious characteristic: not everything is selectable:

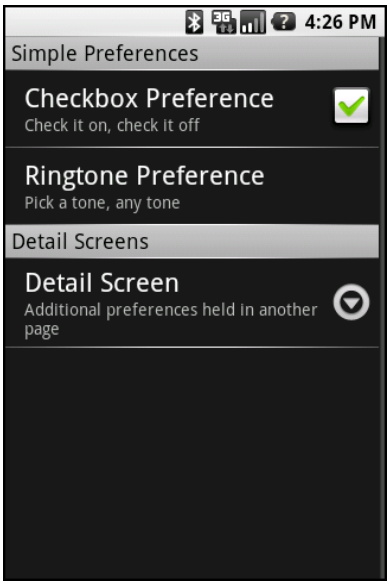


Figure 5. A PreferenceScreen UI

You may have thought that this required some custom widget, or some fancy on-the-fly `View` handling, to achieve this effect.

If so, you would have been wrong.

It turns out that any `ListView` can exhibit this behavior. In this section, we will see how this is achieved and a reusable framework for creating such a `ListView`.

Choosing What Is Selectable

There are two methods in the `Adapter` hierarchy that let you control what is and is not selectable in a `ListView`:

- `areAllItemsSelectable()` should return `true` for ordinary `ListView` widgets and `false` for `ListView` widgets where some items in the Adapter are selectable and others are not
- `isEnabled()`, given a position, should return `true` if the item at that position should be selectable and `false` otherwise

Given these two, it is "merely" a matter of overriding your chosen `Adapter` class and implementing these two methods as appropriate to get the visual effect you desire.

As one might expect, this is not quite as easy as it may sound.

For example, suppose you have a database of books, and you want to present a list of book titles for the user to choose from. Furthermore, suppose you have arranged for the books to be in alphabetical order within each major book style (Fiction, Non-Fiction, etc.), courtesy of a well-crafted `ORDER BY` clause on your query. And suppose you want to have headings, like on the preferences screen, for those book styles.

If you simply take the `Cursor` from that query and hand it to a `SimpleCursorAdapter`, the two methods cited above will be implemented as the default, saying every row is selectable. And, since every row is a book, that is what you want...for the books.

To get the headings in place, your `Adapter` needs to mix the headings in with the books (so they all appear in the proper sequence), return a custom `View` for each (so headings look different than the books), and implement the two methods that control whether the headings or books are selectable. There is no easy way to do this from a simple query.

Instead, you need to be a bit more creative, and wrap your `SimpleCursorAdapter` in something that can intelligently inject the section headings.

Composition for Sections

Jeff Sharkey, author of CompareEverywhere and all-around Android guru, demonstrated a way of using composition to create a `ListView` with section headings. The code presented here is based on his implementation, with a few alterations. As his original code was released under the GPLv3, bear in mind that the code presented here is also released under the GPLv3, as

opposed to the Apache License 2.0 that most of the book's code uses as a license.

The pattern is fairly simple:

- Create one `Adapter` for each section. For example, in the book scenario described above, you might have one `SimpleCursorAdapter` for each book style (one for Fiction, one for Non-Fiction, etc.).
- Put each of those `Adapter` objects into a container `Adapter`, associating each with a heading name.
- Implement, on your container `Adapter` subclass, a method to return the `View` for a heading, much like you might implement `getView()` to return a `View` for a row
- Put the container `Adapter` in the `ListView`, and everything flows from there

You will see this implemented in the `ListView/Sections` sample project, which is another riff on the "list of *lorem ipsum* words" sample you see scattered throughout the *Busy Coder* books.

The layout for the screen is just a `ListView`, because the activity – `SectionedDemo` – is just a `ListActivity`:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:drawSelectorOnTop="true"
/>
```

Most of the smarts can be found in `SectionedAdapter`. This class extends `Adapter` and delegates all of the `Adapter` methods to a list of child `Adapter` objects:

```
package com.commonsware.android.listview;

import android.view.View;
```

```
import android.view.ViewGroup;
import android.widget.Adapter;
import android.widget.BaseAdapter;
import java.util.ArrayList;
import java.util.List;

abstract public class SectionedAdapter extends BaseAdapter {
  abstract protected View getHeaderView(String caption,
                                        int index,
                                        View convertView,
                                        ViewGroup parent);

  private List<Section> sections=new ArrayList<Section>();
  private static int TYPE_SECTION_HEADER=0;

  public SectionedAdapter() {
    super();
  }

  public void addSection(String caption, Adapter adapter) {
    sections.add(new Section(caption, adapter));
  }

  public Object getItem(int position) {
    for (Section section : this.sections) {
      if (position==0) {
        return(section);
      }

      int size=section.adapter.getCount()+1;

      if (position<size) {
        return(section.adapter.getItem(position-1));
      }

      position-=size;
    }

    return(null);
  }

  public int getCount() {
    int total=0;

    for (Section section : this.sections) {
      total+=section.adapter.getCount()+1; // add one for header
    }

    return(total);
  }

  public int getViewTypeCount() {
    int total=1;  // one for the header, plus those from sections
```

```
    for (Section section : this.sections) {
      total+=section.adapter.getViewTypeCount();
    }

    return(total);
  }

  public int getItemViewType(int position) {
    int typeOffset=TYPE_SECTION_HEADER+1;  // start counting from here

    for (Section section : this.sections) {
      if (position==0) {
        return(TYPE_SECTION_HEADER);
      }

      int size=section.adapter.getCount()+1;

      if (position<size) {
        return(typeOffset+section.adapter.getItemViewType(position-1));
      }

      position-=size;
      typeOffset+=section.adapter.getViewTypeCount();
    }

    return(-1);
  }

  public boolean areAllItemsSelectable() {
    return(false);
  }

  public boolean isEnabled(int position) {
    return(getItemViewType(position)!=TYPE_SECTION_HEADER);
  }

  @Override
  public View getView(int position, View convertView,
                      ViewGroup parent) {
    int sectionIndex=0;

    for (Section section : this.sections) {
      if (position==0) {
        return(getHeaderView(section.caption, sectionIndex,
                             convertView, parent));
      }

      int size=section.adapter.getCount()+1;

      if (position<size) {
        return(section.adapter.getView(position-1,
                                       convertView,
                                       parent));
      }
```

```
      position-=size;
      sectionIndex++;
    }

    return(null);
  }

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

  class Section {
    String caption;
    Adapter adapter;

    Section(String caption, Adapter adapter) {
      this.caption=caption;
      this.adapter=adapter;
    }
  }
}
```

`SectionedAdapter` holds a `List` of `Section` objects, where a `Section` is simply a name and an `Adapter` holding the contents of that section of the list. You can give `SectionAdapter` the details of a `Section` via `addSection()` – the sections will appear in the order in which they were added.

`SectionedAdapter` synthesizes the overall list of objects from each of the adapters, plus the section headings:

TBD – diagram

So, for example, the implementation of `getView()` walks each section and returns either a `View` for the section header (if the requested item is the first one for that section) or the `View` from the section's adapter (if the requested item is any other one in this section). The same holds true for `getCount()` and `getItem()`.

One thing that `SectionedAdapter` needs to do, though, is ensure that the pool of section header `View` objects is recycled separately from each section's own pool of `View` objects. To do this, `SectionedAdapter` takes advantage of `getViewTypeCount()`, by returning the total number of distinct types of `View`

objects from all section `Adapters` plus one for its own header `View` pool. Similarly, `getItemViewType()` considers the `0`th `View` type to be the header `View` pool, with the pools for each `Adapter` in sequence starting from 1. This pattern requires that each section `Adapter` have its `View` type numbers starting from 0 and incrementing by 1, but most `Adapter` classes only use one `View` type and do not even implement their own `getViewTypeCount()` or `getItemViewType()`, so this will work most of the time.

To use a `SectionedAdapter`, `SectionedDemo` simply creates one, adds in three sections (with three sets of the *lorem ipsum* words), and attaches the `SectionedAdapter` to the `ListView` for the `ListActivity`:

```
package com.commonsware.android.listview;

import android.app.ListActivity;
import android.content.Context;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class SectionedDemo extends ListActivity {
  private static String[] items={"lorem", "ipsum", "dolor",
                                 "sit", "amet", "consectetuer",
                                 "adipiscing", "elit", "morbi",
                                 "vel", "ligula", "vitae",
                                 "arcu", "aliquet", "mollis",
                                 "etiam", "vel", "erat",
                                 "placerat", "ante",
                                 "porttitor", "sodales",
                                 "pellentesque", "augue",
                                 "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    adapter.addSection("Original",
                       new ArrayAdapter<String>(this,
                         android.R.layout.simple_list_item_1,
                         items));
```

```
    List<String> list=Arrays.asList(items);

    Collections.shuffle(list);

    adapter.addSection("Shuffled",
                       new ArrayAdapter<String>(this,
                         android.R.layout.simple_list_item_1,
                         list));

    list=Arrays.asList(items);

    Collections.shuffle(list);

    adapter.addSection("Re-shuffled",
                       new ArrayAdapter<String>(this,
                         android.R.layout.simple_list_item_1,
                         list));

    setListAdapter(adapter);
  }

  SectionedAdapter adapter=new SectionedAdapter() {
    protected View getHeaderView(String caption, int index,
                                 View convertView,
                                 ViewGroup parent) {
      TextView result=(TextView)convertView;

      if (convertView==null) {
        result=(TextView)getLayoutInflater()
                                 .inflate(R.layout.header,
                                        null);
      }

      result.setText(caption);

      return(result);
    }
  };
}
```

The result is much as you might expect:

TBD – screenshots

Here, the headers are simple bits of text with an appropriate style applied. Your section headers, of course, can be as complex as you like.

From Head To Toe

Perhaps you do not need section headers scattered throughout your list. If you only need extra "fake rows" at the beginning or end of your list, you can use header and footer views.

`ListView` supports `addHeaderView()` and `addFooterView()` methods that allow you to add `View` objects to the beginning and end of the list, respectively. These `View` objects otherwise behave like regular rows, in that they are part of the scrolled area and will scroll off the screen if the list is long enough. If you want fixed headers or footers, rather than put them in the `ListView` itself, put them outside the `ListView`, perhaps using a `LinearLayout`.

To demonstrate header and footer views, take a peek at `ListView/HeaderFooter`, particularly the `HeaderFooterDemo` class:

```
package com.commonsware.android.listview;

import android.app.ListActivity;
import android.content.Context;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListView;
import android.widget.TextView;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class HeaderFooterDemo extends ListActivity {
  private static String[] items={"lorem", "ipsum", "dolor",
                                 "sit", "amet", "consectetuer",
                                 "adipiscing", "elit", "morbi",
                                 "vel", "ligula", "vitae",
                                 "arcu", "aliquet", "mollis",
                                 "etiam", "vel", "erat",
                                 "placerat", "ante",
                                 "porttitor", "sodales",
                                 "pellentesque", "augue",
                                 "purus"};
  long startTime=SystemClock.uptimeMillis();
```

```
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    getListView().addHeaderView(buildHeader());
    getListView().addFooterView(buildFooter());
    setListAdapter(new ArrayAdapter<String>(this,
                       android.R.layout.simple_list_item_1,
                       items));
  }

  private View buildHeader() {
    Button btn=new Button(this);

    btn.setText("Randomize!");
    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        List<String> list=Arrays.asList(items);

        Collections.shuffle(list);

        setListAdapter(new ArrayAdapter<String>(HeaderFooterDemo.this,
                           android.R.layout.simple_list_item_1,
                           list));
      }
    });

    return(btn);
  }

  private View buildFooter() {
    TextView txt=new TextView(this);

    updateFooter(txt);

    return(txt);
  }

  private void updateFooter(final TextView txt) {
    long runtime=(SystemClock.uptimeMillis()-startTime)/1000;

    txt.setText(String.valueOf(runtime)+" seconds since activity launched");

    txt.postDelayed(new Runnable() {
      public void run() {
        updateFooter(txt);
      }
    }, 1000);
  }
}
```

Here, we add a header `View` built via `buildHeader()`, returning a `Button` that, when clicked, will shuffle the contents of the list. We also add a footer `View`

built via `buildFooter()`, returning a `TextView` that shows how long the activity has been running, updated every second. The list itself is the ever-popular list of *lorem ipsum* words.

When initially displayed, the header is visible but the footer is not, because the list is too long:

TBD – screenshot

If you scroll downward, the header will slide off the top, and eventually the footer will scroll into view:

TBD – screenshot

Control Your Selection

The stock Android UI for a selected `ListView` row is fairly simplistic: it highlights the row in orange...and nothing more. You can control the `Drawable` used for selection via the `android:listSelector` and `android:drawSelectorOnTop` attributes on the `ListView` element in your layout. However, even those simply apply some generic look to the selected row.

It may be you want to do something more elaborate for a selected row, such as changing the row around to expose more information. Maybe you have thumbnail photos but only display the photo on the selected row. Or perhaps you want to show some sort of secondary line of text, like a person's instant messenger status, only on the selected row. Or, there may be times you want a more subtle indication of the selected item than having the whole row show up in some neon color. The stock Android UI for highlighting a selection will not do any of this for you.

That just means you have to do it yourself. The good news is, it is not very difficult.

Create a Unified Row View

The simplest way to accomplish this is for each row `View` to have all of the widgets you want for the selected-row perspective, but with the "extra stuff" flagged as invisible at the outset. That way, rows initially look "normal" when put into the list – all you need to do is toggle the invisible widgets to visible when a row gets selected and toggle them back to invisible when a row is de-selected.

For example, in the `ListView/Selector` project, you will find a row.xml layout representing a row in a list:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent" >
  <View
    android:id="@+id/bar"
    android:background="#FFFF0000"
    android:layout_width="5px"
    android:layout_height="fill_parent"
    android:visibility="invisible"
  />
  <TextView
    android:id="@+id/label"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textSize="10pt"
    android:paddingTop="2px"
    android:paddingBottom="2px"
    android:paddingLeft="5px"
  />
</LinearLayout>
```

There is a `TextView` representing the bulk of the row. Before it, though, on the left, is a plain `View` named `bar`. The background of the `View` is set to red (`android:background = "#FFFF0000"`) and the width to `5px`. More importantly, it is set to be invisible (`android:visibility = "invisible"`). Hence, when the row is put into a `ListView`, the red bar is not seen...until we make the bar visible.

Configure the List, Get Control on Selection

Next, we need to set up a `ListView` and arrange to be notified when rows are selected and de-selected. That is merely a matter of calling `setOnItemSelectedListener()` for the `ListView`, providing a listener to be notified on a selection change. You can see that in the context of a `ListActivity` in our `SelectorDemo` class:

```
package com.commonsware.android.listview;

import android.app.ListActivity;
import android.content.Context;
import android.os.Bundle;
import android.content.res.ColorStateList;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;

public class SelectorDemo extends ListActivity {
  private static ColorStateList allWhite=ColorStateList.valueOf(0xFFFFFFFF);
  private static String[] items={"lorem", "ipsum", "dolor",
                                 "sit", "amet", "consectetuer",
                                 "adipiscing", "elit", "morbi",
                                 "vel", "ligula", "vitae",
                                 "arcu", "aliquet", "mollis",
                                 "etiam", "vel", "erat",
                                 "placerat", "ante",
                                 "porttitor", "sodales",
                                 "pellentesque", "augue",
                                 "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new SelectorAdapter(this));
    getListView().setOnItemSelectedListener(listener);
  }

  class SelectorAdapter extends ArrayAdapter {
    SelectorAdapter(Context ctxt) {
      super(ctxt, R.layout.row, items);
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      SelectorWrapper wrapper=null;
```

```
      if (convertView==null) {
        convertView=getLayoutInflater().inflate(R.layout.row,
                                                null);
        wrapper=new SelectorWrapper(convertView);
        wrapper.getLabel().setTextColor(allWhite);
        convertView.setTag(wrapper);
      }
      else {
        wrapper=(SelectorWrapper)convertView.getTag();
      }

      wrapper.getLabel().setText(items[position]);

      return(convertView);
    }
  }

  class SelectorWrapper {
    View row=null;
    TextView label=null;
    View bar=null;

    SelectorWrapper(View row) {
      this.row=row;
    }

    TextView getLabel() {
      if (label==null) {
        label=(TextView)row.findViewById(R.id.label);
      }

      return(label);
    }

    View getBar() {
      if (bar==null) {
        bar=row.findViewById(R.id.bar);
      }

      return(bar);
    }
  }

  AdapterView.OnItemSelectedListener listener=new
AdapterView.OnItemSelectedListener() {
    View lastRow=null;

    public void onItemSelected(AdapterView<?> parent,
                               View view, int position,
                               long id) {
      if (lastRow!=null) {
        SelectorWrapper wrapper=(SelectorWrapper)lastRow.getTag();
```

```
        wrapper.getBar().setVisibility(View.INVISIBLE);
      }

      SelectorWrapper wrapper=(SelectorWrapper)view.getTag();

      wrapper.getBar().setVisibility(View.VISIBLE);
      lastRow=view;
    }

    public void onNothingSelected(AdapterView<?> parent) {
      if (lastRow!=null) {
        SelectorWrapper wrapper=(SelectorWrapper)lastRow.getTag();

        wrapper.getBar().setVisibility(View.INVISIBLE);
        lastRow=null;
      }
    }
  };
}
```

`SelectorDemo` sets up a `SelectorAdapter`, which follow the view-wrapper pattern established in *The Busy Coder's Guide to Android Development*. Each row is created from the layout shown earlier, with a `SelectorWrapper` providing access to both the `TextView` (for setting the text in a row) and the bar `View`.

Change the Row

Our `AdapterView.OnItemSelectedListener` instance keeps track of the last selected row (`lastRow`). When the selection changes to another row in `onItemSelected()`, we make the bar from the last selected row invisible, before we make the bar visible on the newly-selected row. In `onNothingSelected()`, we make the bar invisible and make our last selected row be null.

The net effect is that as the selection changes, we toggle the bar off and on as needed to indicate which is the selected row.

In the layout for the activity's `ListView`, we turn off the regular highlighting:

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

```
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:listSelector="#00000000"
/>
```

The result is we are controlling the highlight, in the form of the red bar:

TBD – screenshot

Obviously, what we do to highlight a row could be much more elaborate than what is demonstrated here. At the same time, it needs to be fairly quick to execute, lest the list appear to be too sluggish.

PART II – Advanced Media

CHAPTER 4

Creating Drawables

`Drawable` resources come in all shapes and sizes, and not just in terms of pixel dimensions. While many `Drawable` resources will be PNG or JPEG files, you can easily create other resources that supply other sorts of `Drawable` objects to your application. In this chapter, we will examine a few of these that may prove useful as you try to make your application look its best.

Traversing Along a Gradient

Gradients have long been used to add "something a little extra" to a user interface, whether it is Microsoft adding them to Office's title bars in the late 1990's or the seemingly endless number of gradient buttons adorning "Web 2.0" sites.

And now, you can have gradients in your Android applications as well.

The easiest way to create a gradient is to use an XML file to describe the gradient. By placing the file in `res/drawable/`, it can be referenced as a `Drawable` resource, no different than any other such resource, like a PNG file.

For example, here is a gradient `Drawable` resource, `active_row.xml`, from the `Drawable/Gradient` sample project:

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

```
  <gradient
    android:startColor="#44FF0000"
    android:endColor="#FFFF0000"
    android:angle="270"
  />
  <padding
    android:top="2px"
    android:bottom="2px"
  />
  <corners android:radius="6px" />
</shape>
```

A gradient is applied to the more general-purpose `<shape>` element, in this case, a rectangle. The gradient is defined as having a start and end color – in this case, the gradient is an increasing amount of red, with only the alpha channel varying to control how much the background blends in. The color is applied in a direction determined by the number of degrees specified by the `android:angle` attribute, with `270` representing "down" (start color at the top, end color at the bottom).

As with any other XML-defined shape, you can control various aspects of the way the shape is drawn. In this case, we put some padding around the drawable and round off the corners of the rectangle.

To use this `Drawable` in Java code, you can reference it as `R.drawable.active_row`. One possible use of a gradient is in custom `ListView` row selection, as shown in `GradientDemo`:

```
package com.commonsware.android.drawable;

import android.app.ListActivity;
import android.content.Context;
import android.os.Bundle;
import android.content.res.ColorStateList;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;

public class GradientDemo extends ListActivity {
  private static ColorStateList allWhite=ColorStateList.valueOf(0xFFFFFFFF);
  private static String[] items={"lorem", "ipsum", "dolor",
                                 "sit", "amet", "consectetuer",
                                 "adipiscing", "elit", "morbi",
```

```
                                "vel", "ligula", "vitae",
                                "arcu", "aliquet", "mollis",
                                "etiam", "vel", "erat",
                                "placerat", "ante",
                                "porttitor", "sodales",
                                "pellentesque", "augue",
                                "purus"};

@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);
  setListAdapter(new GradientAdapter(this));
  getListView().setOnItemSelectedListener(listener);
}

class GradientAdapter extends ArrayAdapter {
  GradientAdapter(Context ctxt) {
    super(ctxt, R.layout.row, items);
  }

  @Override
  public View getView(int position, View convertView,
                      ViewGroup parent) {
    GradientWrapper wrapper=null;

    if (convertView==null) {
      convertView=getLayoutInflater().inflate(R.layout.row,
                                              null);
      wrapper=new GradientWrapper(convertView);
      convertView.setTag(wrapper);
    }
    else {
      wrapper=(GradientWrapper)convertView.getTag();
    }

    wrapper.getLabel().setText(items[position]);

    return(convertView);
  }
}

class GradientWrapper {
  View row=null;
  TextView label=null;

  GradientWrapper(View row) {
    this.row=row;
  }

  TextView getLabel() {
    if (label==null) {
      label=(TextView)row.findViewById(R.id.label);
    }
```

```
      return(label);
    }
  }

  AdapterView.OnItemSelectedListener listener=new
AdapterView.OnItemSelectedListener() {
    View lastRow=null;

    public void onItemSelected(AdapterView<?> parent,
                               View view, int position,
                               long id) {
      if (lastRow!=null) {
        lastRow.setBackgroundColor(0x00000000);
      }

      view.setBackgroundResource(R.drawable.active_row);
      lastRow=view;
    }

    public void onNothingSelected(AdapterView<?> parent) {
      if (lastRow!=null) {
        lastRow.setBackgroundColor(0x00000000);
        lastRow=null;
      }
    }
  };
}
```

In an earlier chapter, we showed how you can get control and customize how a selected row appears in a `ListView`. This time, we apply the gradient rounded rectangle as the background of the row. We could have accomplished this via appropriate choices for `android:listSelector` and `android:drawSelectorOnTop` as well.

The result is a selection bar implementing the gradient:

TBD – screenshot

Note that because the list background is black, the red is mixed with black on the top end of the gradient. If the list background were white, the top end of the gradient would be red mixed with white, as determined by the alpha channel specified on the gradient's top color.

A Stitch In Time Saves Nine

As you read through the Android documentation, you no doubt ran into references to "nine-patch" or "9-patch" and wondered what Android had to do with quilting. Rest assured, you will not need to take up needlework to be an effective Android developer.

If, however, you are looking to create backgrounds for resizable widgets, like a `Button`, you will probably need to work with nine-patch images.

As the Android documentation states, a nine-patch is "a PNG image in which you define stretchable sections that Android will resize to fit the object at display time to accommodate variable sized sections, such as text strings". By using a specially-created PNG file, Android can avoid trying to use vector-based formats (e.g., SVG) and their associated overhead when trying to create a background at runtime. Yet, at the same time, Android can still resize the background to handle whatever you want to put inside of it, such as the text of a `Button`.

In this section, we will cover some of the basics of nine-patch graphics, including how to customize and apply them to your own Android layouts.

The Name and the Border

Nine-patch graphics are PNG files whose names end in `.9.png`. This means they can be edited using normal graphics tools, but Android knows to apply nine-patch rules to their use.

What makes a nine-patch graphic different than an ordinary PNG is a one-pixel-wide border surrounding the image. When drawn, Android will remove that border, showing only the stretched rendition of what lies inside the border. The border is used as a control channel, providing instructions to Android for how to deal with stretching the image to fit its contents.

TBD – diagram showing image and control area

Padding and the Box

Along the right and bottom sides, you can draw one-pixel-wide black lines to indicate the "padding box". Android will stretch the image such that the contents of the widget will fit inside that padding box.

For example, suppose we are using a nine-patch as the background of a `Button`. When you set the text to appear in the button (e.g., "Hello, world!"), Android will compute the size of that text, in terms of width and height in pixels. Then, it will stretch the nine-patch image such that the text will reside inside the padding box. What lies outside the padding box forms the border of the button, typically a rounded rectangle of some form.

TBD – diagram showing image and padding box

Stretchable Zones

To tell Android where on the image to actually do the stretching, draw one-pixel-wide black lines on the top and left sides of the image. Android will scale the graphic only in those areas – areas outside the stretchable zones are not stretched.

Perhaps the most common pattern is the center-stretch, where the middle portions of the image on both axes are considered stretchable, but the edges are not:

TBD – diagram showing image and stretchable zones

Here, the stretchable zones will be stretched just enough for the contents to fit in the padding box. The edges of the graphic are left unstretched.

TBD – diagram showing before and after

Some additional rules to bear in mind:

- If you have multiple discrete stretchable zones along an axis (e.g., two zones separated by whitespace), Android will stretch both of them but keep them in their current proportions. So, if the first zone is twice as wide as the second zone in the original graphic, the first zone will be twice as wide as the second zone in the stretched graphic.
- If you leave out the control lines for the padding box, it is assumed that the padding box and the stretchable zones are one and the same.

Tooling

To experiment with nine-patch images, you may wish to use the `draw9patch` program, found in the `tools/` directory of your SDK installation:

TBD – screenshot

While a regular graphics editor would allow you to draw any color on any pixel, `draw9patch` limits you to drawing or erasing pixels in the control area. If you attempt to draw inside the main image area itself, you will be blocked:

TBD – screenshot

On the right, you will see samples of the image in various stretched sizes, so you can see the impact as you change the stretchable zones and padding box.

While this is convenient for working with the nine-patch nature of the image, you will still need some other graphics editor to create or modify the body of the image itself. For example, the image shown above, from the `Drawable/NinePatch` project, is a modified version of a nine-patch graphic from the SDK's `ApiDemos`, where the GIMP was used to add the neon green stripe across the bottom portion of the image.

Using Nine-Patch Images

Nine-patch images are most commonly used as backgrounds, as illustrated by the following layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TableLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
  >
    <TableRow
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Horizontal:"
      />
      <SeekBar android:id="@+id/horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Vertical:"
      />
      <SeekBar android:id="@+id/vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
  </TableLayout>
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
```

```
    <Button android:id="@+id/resize"
      android:layout_width="48px"
      android:layout_height="48px"
      android:text="Hi!"
      android:background="@drawable/button"
    />
  </LinearLayout>
</LinearLayout>
```

Here, we have two `SeekBar` widgets, labeled for the horizontal and vertical axes, plus a `Button` set up with our nine-patch graphic as its background (`android:background = "@drawable/button"`).

The `NinePatchDemo` activity then uses the two `SeekBar` widgets to let the user control how large the button should be drawn on-screen, starting from an initial size of `48px` square:

```
package com.commonsware.android.drawable;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.widget.SeekBar;

public class NinePatchDemo extends Activity {
  SeekBar horizontal=null;
  SeekBar vertical=null;
  View thingToResize=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    thingToResize=findViewById(R.id.resize);

    horizontal=(SeekBar)findViewById(R.id.horizontal);
    vertical=(SeekBar)findViewById(R.id.vertical);

    horizontal.setMax(272);  // 320 less 48 starting size
    vertical.setMax(272);    // keep it square @ max

    horizontal.setOnSeekBarChangeListener(h);
    vertical.setOnSeekBarChangeListener(v);
  }

  SeekBar.OnSeekBarChangeListener h=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
```

```
                                int progress,
                                boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(48+progress,
                                                                   old.height);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };

  SeekBar.OnSeekBarChangeListener v=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                int progress,
                                boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(old.width,
                                                                   48+progress);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };
}
```

The result is an application that can be used much like the right pane of `draw9patch`, to see how the nine-patch graphic looks on an actual device or emulator in various sizes:

TBD – screenshots

CHAPTER 5

Animating Widgets

Android is full of things that move. You can swipe left and right on the home screen to view other panels of the desktop. You can drag icons around on the home screen. You can drag down the notifications area or drag up the applications drawer. And that is just on one screen!

Of course, it would be nice to employ such animations in your own application. While this chapter will not cover full-fledged drag-and-drop, we will cover some of the basic animations and how to apply them to your existing widgets. Along the way, we will implement a similar sort of "drawer" container to the application drawer found on the home screen.

It's Not Just For Toons Anymore

Android has a package of classes (`android.view.animation`) dedicated to animating the movement and behavior of widgets.

They center around an `Animation` base class that describes what is to be done. Built-in animations exist to move a widget (`TranslateAnimation`), change the transparency of a widget (`AlphaAnimation`), revolving a widget (`RotateAnimation`), and resizing a widget (`ScaleAnimation`). There is even a way to aggregate animations together into a composite `Animation` called an `AnimationSet`. Later sections in this chapter will examine the use of several of these animations.

Given that you have an animation, to apply it, you have two main options:

- You may be using a container that supports animating its contents, such as a `ViewFlipper` or `TextSwitcher`. These are typically subclasses of `ViewAnimator` and let you define the "in" and "out" animations to apply. For example, with a `ViewFlipper`, you can specify how it flips between `View`s in terms of what animation is used to animate "out" the currently-visible `View` and what animation is used to animate "in" the replacement `View`. Examples of this sort of animation can be found in *The Busy Coder's Guide to Android Development*.
- You can simply tell any `View` to `startAnimation()`, given the `Animation` to apply to itself. This is the technique we will be seeing used in the examples in this chapter.

A Quirky Translation

Animation takes some getting used to. Frequently, it takes a fair bit of experimentation to get it all working as you wish. This is particularly true of `TranslateAnimation`, as not everything about it is intuitive, even to authors of Android books.

Mechanics of Translation

The simple constructor for `TranslateAnimation` takes four parameters describing how the widget should move: the before and after X offsets from the current position, and the before and after Y offsets from the current position. The Android documentation refers to these as `fromXDelta`, `toXDelta`, `fromYDelta`, and `toYDelta`.

In Android's pixel-space, an `(X,Y)` coordinate of `(0,0)` represents the upper-left corner of the screen. Hence, if `toXDelta` is greater than `fromXDelta`, the widget will move to the right, if `toYDelta` is greater than `fromYDelta`, the widget will move down, and so on.

Imagining a Drawer

Think now about the application drawer on the Android home screen. It is anchored at the bottom of the screen. If we are to make a similar sort of drawer, we will want to animate the drawer up from the bottom when it is opened and back down to the bottom when it is closed. As important, we need to know how far we want to "open" the drawer, in terms of a target height for the open drawer.

One way to implement such a drawer is to have a container (e.g., a `LinearLayout`) whose contents are absent (`GONE`) when the drawer is closed and is present (`VISIBLE`) when the drawer is open. If we simply toggled `setVisibility()` using the aforementioned values, though, the drawer would wink open and closed immediately, without any sort of animation. So, instead, we want to:

- Make the drawer visible and animate it up from the bottom of the screen when we open the drawer
- Animate it down to the bottom of the screen and make the drawer gone when we close the drawer

Our drawer, in addition to the contents, will also need an `ImageButton` to toggle the drawer open and closed. This `ImageButton` is always visible; the question is simply whether it is sitting at the bottom of the screen or if it is sitting atop the open drawer.

The Aftermath

This brings up a key point with respect to `TranslateAnimation`: the animation temporarily moves the widget, but if you want the widget to stay where it is when the animation is over, you have to handle that yourself. Otherwise, the widget will snap back to its original position when the animation completes.

In the case of the drawer opening, we handle that via the transition from `GONE` to `VISIBLE`. Technically speaking, the drawer is always "open", in that we are not, in the end, changing its position. But when the body of the drawer

is GONE, it takes up no space on the screen; when we make it VISIBLE, it takes up whatever space it is supposed to.

Later in this chapter, we will cover how to use animation listeners to accomplish this end for closing the drawer.

Introducing DrawerLayout

With all that said, turn your attention to the Animation/Drawer project and, in particular, the DrawerLayout class.

This class implements a layout that works as a drawer, anchored to the bottom of the screen. Clicking a button opens or closes the drawer. The drawer itself is a LinearLayout, so you can put whatever contents you want in there. However, this simple implementation assumes there is only one widget in the LinearLayout besides the tab (ImageButton and a View serving as the top edge of the drawer).

We use two flavors of TranslateAnimation, one for opening the drawer and one for closing it.

Here is the opening animation:

```
anim=new TranslateAnimation(0.0f, 0.0f, targetHeight,
                            0.0f);
```

Our fromXDelta and toXDelta are both 0, since we are not shifting the drawer's position along the horizontal axis. Our fromYDelta is targetHeight (representing how big we want the drawer to be), because we want the drawer to start the animation at the bottom of the screen; our toYDelta is 0 because we want the drawer to be at its "natural" open position at the end of the animation.

Conversely, here is the closing animation:

```
anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                            targetHeight);
```

It has the same basic structure, except the Y values are reversed, since we want the drawer to start open and animate to a closed position.

The result is a container that can be closed:

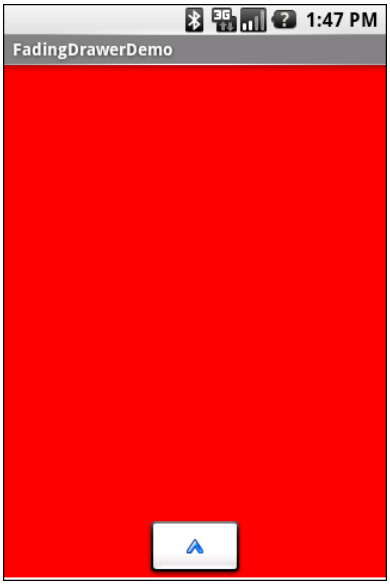


Figure 6. The Drawer sample application, with the drawer closed

...or open:

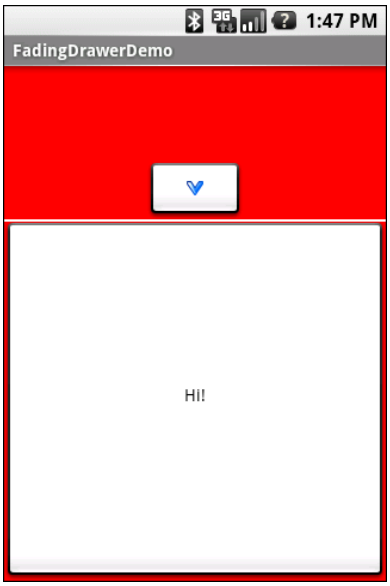


Figure 7. The Drawer sample application, with the drawer open

Using the Animation

When setting up an animation, you also need to indicate how long the animation should take. This is done by calling `setDuration()` on the animation, providing the desired length of time in milliseconds.

When we are ready with the animation, we simply call `startAnimation()` on the `DrawerLayout` itself, causing it to move as specified by the `TranslateAnimation` instance.

Fading To Black. Or Some Other Color.

`AlphaAnimation` allows you to fade a widget in or out by making it less or more transparent. The greater the transparency, the more the widget appears to be "fading".

Alpha Numbers

You may be used to alpha channels, when used in `#AARRGGBB` color notation, or perhaps when working with alpha-capable image formats like PNG.

Similarly, `AlphaAnimation` allows you to change the alpha channel for an entire widget, from fully-solid to fully-transparent.

In Android, a float value of `1.0` indicates a fully-solid widget, while a value of `0.0` indicates a fully-transparent widget. Values in between, of course, represent various amounts of transparency.

Hence, it is common for an `AlphaAnimation` to either start at `1.0` and smoothly change the alpha to `0.0` (a fade) or vice versa.

Animations in XML

With `TranslateAnimation`, we showed how to construct the animation in Java source code. One can also create animation resources, which define the animations using XML. This is similar to the process for defining layouts, albeit much simpler.

For example, there is a second animation project, `Animation/FadingDrawer`, which demonstrates a drawer that fades out as it is closed. In there, you will find a `res/anim/` directory, which is where animation resources should reside. In there, you will find `fade.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromAlpha="1.0"
  android:toAlpha="0.0" />
```

The name of the root element indicates the type of animation (in this case, alpha for an `AlphaAnimation`). The attributes specify the characteristics of the animation, in this case a fade from `1.0` to `0.0` on the alpha channel.

This XML is the same as calling `new AlphaAnimation(1.0f,0.0f)` in Java.

Using XML Animations

To make use of XML-defined animations, you need to inflate them, much as you might inflate a `View` or `Menu` resource. This is accomplished by using the `loadAnimation()` static method on the `AnimationUtils` class:

```
fadeOut=AnimationUtils.loadAnimation(ctxt, R.anim.fade);
```

Here, we are loading our fade animation, given a `Context`. This is being put into an `Animation` variable, so we neither know nor care that this particular XML that we are loading defines an `AlphaAnimation` instead of, say, a `RotateAnimation`.

When It's All Said And Done

Sometimes, you need to take action when an animation completes.

For example, when we close the drawer, we want to use a `TranslationAnimation` to slide it down from the open position to closed...then *keep* it closed. With the system used in `DrawerLayout`, keeping the drawer closed is a matter of calling `setVisibility()` on the contents with `GONE`.

However, you cannot do that when the animation begins; otherwise, the drawer is gone by the time you try to animate its motion.

Instead, you need to arrange to have it be gone when the animation ends. To do that, you use a animation listener.

An animation listener is simply an instance of the `AnimationListener` interface, provided to an animation via `setAnimationListener()`. The listener will be invoked when the starts, ends, or repeats (the latter courtesy of `CycleInterpolator`, discussed later in this chapter). You can put logic in the `onAnimationEnd()` callback in the listener to take action when the animation finishes.

For example, here is the `AnimationListener` for `DrawerLayout`:

```
Animation.AnimationListener collapseListener=new Animation.AnimationListener() {
  public void onAnimationEnd(Animation animation) {
    tab.setImageResource(R.drawable.up);
    contents.setVisibility(View.GONE);
  }

  public void onAnimationRepeat(Animation animation) {
    // not needed
  }

  public void onAnimationStart(Animation animation) {
    // not needed
  }
};
```

All we do is set the `ImageButton`'s image to be the upward-pointing arrow and setting our content's visibility to be `GONE`, thereby closing the drawer.

Hit The Accelerator

In addition to the `Animation` classes themselves, Android also provides a set of `Interpolator` classes. These provide instructions for how an animation is supposed to behave during its operating period.

For example, the `AccelerateInterpolator` indicates that, during the duration of an animation, the rate of change of the animation should begin slowly and accelerate until the end. When applied to a `TranslateAnimation`, for example, the sliding movement will start out slowly and pick up speed until the movement is complete.

There are several implementations of the Interpolator interface besides `AccelerateInterpolator`, including:

- `AccelerateDecelerateInterpolator`, which starts slowly, picks up speed in the middle, and slows down again at the end
- `DecelerateInterpolator`, which starts quickly and slows down towards the end
- `LinearInterpolator`, the default, which indicates the animation should proceed smoothly from start to finish

- `CycleInterpolator`, which repeats an animation for a number of cycles, following the AccelerateDecelerateInterpolator pattern (slow, then fast, then slow)

To apply an interpolator to an animation, simply call `setInterpolator()` on the animation with the `Interpolator` instance, such as the following line from `DrawerLayout`:

```
anim.setInterpolator(new AccelerateInterpolator(1.0f));
```

You can also specify one of the stock interpolators via the `android:interpolator` attribute in your animation XML file.

Animate. Set. Match.

For the `Animation/FadingDrawer` project, though, we want the drawer to slide open, but also fade when it slides closed. This implies two animations working at the same time (a fade and a slide). Android supports this via the `AnimationSet` class.

An `AnimationSet` is itself an `Animation` implementation. Following the composite design pattern, it simply cascades the major `Animation` events to each of the animations in the set.

To create a set, just create an `AnimationSet` instance, add the animations, and configure the set. For example, here is the logic from the `DrawerLayout` implementation in `Animation/FadingDrawer`:

```
public void onClick(View view) {
  TranslateAnimation anim=null;
  AnimationSet set=new AnimationSet(true);

  isOpen=!isOpen;

  if (isOpen) {
    tab.setImageResource(R.drawable.down);
    contents.setVisibility(View.VISIBLE);

    anim=new TranslateAnimation(0.0f, 0.0f, targetHeight,
                                0.0f);
```

```
  }
  else {
    anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                targetHeight);
    anim.setAnimationListener(collapseListener);
    set.addAnimation(fadeOut);
  }

  set.addAnimation(anim);
  set.setDuration(speed);
  set.setInterpolator(new AccelerateInterpolator(1.0f));
  startAnimation(set);
}
```

If the drawer is to be opened, we set the `ImageButton`'s icon to point downward, make the contents visible (so we can animate the motion upwards), and create a `TranslateAnimation` for the upward movement. If the drawer is to be closed, we create a `TranslateAnimation` for the downward movement, but also add a pre-defined `AlphaAnimation` (`fadeOut`) to an `AnimationSet`. In either case, we add the `TranslateAnimation` to the set, give the set a duration and interpolator, and run the animation.

A Chest of Drawers

Of course, the drawer implementations demonstrated here are simple book examples. You are welcome to use them, but there are others out there that you may consider.

The canonical drawer is the SlidingDrawer in Android itself. Alas, as of the 1.0r2 SDK, this class is internal to Android and should not be used directly by Android applications. Reportedly, it will be made public as part of a future SDK release. Plus, the source code is open, so there is nothing preventing you from cloning it in your own namespace, to give you something to use before the official one is available.

Also, the community crafts and publishes its own widgets from time to time. One such drawer, `Panel`, was implemented by pskink and can be found his his android-misc-widgets project on Google Code.

CHAPTER 6

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 has multimedia support for you, as a developer, to build your own games, media players, and so on.

This chapter is focused on audio and video playback; other chapters will tackle media input, including the camera and audio recording.

Get Your Media On

In Android, you have five different places you can pull media clips from – one of these will hopefully fit your needs:

1. You can package media clips as raw resources (`res/raw` in your project), so they are bundled with your application. The benefit is that you're guaranteed the clips will be there; the downside is that they cannot be replaced without upgrading the application.
2. You can package media clips as assets (`assets/` in your project) and reference them via `file:///android_asset/` URLs in a `Uri`. The benefit over raw resources is that this location works with APIs that expect `Uri` parameters instead of resource IDs. The downside – assets are only replaceable when the application is upgraded – remains.

3. You can store media in an application-local directory, such as content you download off the Internet. Your media may or may not be there, and your storage space isn't infinite, but you can replace the media as needed.
4. You can store media – or reference media that the user has stored herself – that is on an SD card. There is likely more storage space on the card than there is on the device, and you can replace the media as needed, but other applications have access to the SD card as well.
5. You can, in some cases, stream media off the Internet, bypassing any local storage, as with the StreamFurious application

Internet streaming is tricky, particularly for video, and is well beyond the scope of this book. For the T-Mobile G1, the recommended approach for anything of significant size is to put it on the SD card, as there is very little on-board flash memory for file storage.

Making Noise

The crux of playing back audio comes in the form of the `MediaPlayer` class. With it, you can feed it an audio clip, start/stop/pause playback, and get notified on key events, such as when the clip is ready to be played or is done playing.

You have three ways to set up a `MediaPlayer` and tell it what audio clip to play:

1. If the clip is a raw resource, use `MediaPlayer.create()` and provide the resource ID of the clip
2. If you have a `Uri` to the clip, use the `Uri`-flavored version of `MediaPlayer.create()`
3. If you have a string path to the clip, just create a `MediaPlayer` using the default constructor, then call `setDataSource()` with the path to the clip

Next, you need to call `prepare()` or `prepareAsync()`. Both will set up the clip to be ready to play, such as fetching the first few seconds off the file or

stream. The `prepare()` method is synchronous; as soon as it returns, the clip is ready to play. The `prepareAsync()` method is asynchronous – more on how to use this version later.

Once the clip is prepared, `start()` begins playback, `pause()` pauses playback (with `start()` picking up playback where `pause()` paused), and `stop()` ends playback. One caveat: you cannot simply call `start()` again on the `MediaPlayer` once you have called `stop()` – we'll cover a workaround a bit later in this section.

To see this in action, take a look at the `Media/Audio` sample project. The layout is pretty trivial, with three buttons and labels for play, pause, and stop:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="4px"
  >
    <ImageButton android:id="@+id/play"
      android:src="@drawable/play"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:paddingRight="4px"
      android:enabled="false"
    />
    <TextView
      android:text="Play"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="4px"
  >
```

```
    <ImageButton android:id="@+id/pause"
      android:src="@drawable/pause"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:paddingRight="4px"
    />
    <TextView
      android:text="Pause"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="4px"
  >
    <ImageButton android:id="@+id/stop"
      android:src="@drawable/stop"
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:paddingRight="4px"
    />
    <TextView
      android:text="Stop"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
</LinearLayout>
```

The Java, of course, is where things get interesting:

```
public class AudioDemo extends Activity
  implements MediaPlayer.OnCompletionListener {

  private ImageButton play;
  private ImageButton pause;
  private ImageButton stop;
  private MediaPlayer mp;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
```

```
    play=(ImageButton)findViewById(R.id.play);
    pause=(ImageButton)findViewById(R.id.pause);
    stop=(ImageButton)findViewById(R.id.stop);

    play.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        play();
      }
    });

    pause.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        pause();
      }
    });

    stop.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        stop();
      }
    });

    setup();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    if (stop.isEnabled()) {
      stop();
    }
  }

  public void onCompletion(MediaPlayer mp) {
    stop();
  }

  private void play() {
    mp.start();

    play.setEnabled(false);
    pause.setEnabled(true);
    stop.setEnabled(true);
  }

  private void stop() {
    mp.stop();
    mp.release();
    setup();
  }

  private void pause() {
    mp.pause();
```

```
    play.setEnabled(true);
    pause.setEnabled(false);
    stop.setEnabled(true);
  }

  private void loadClip() {
    try {
      mp=MediaPlayer.create(this, R.raw.clip);
      mp.setOnCompletionListener(this);
    }
    catch (Throwable t) {
      goBlooey(t);
    }
  }

  private void setup() {
    loadClip();
    play.setEnabled(true);
    pause.setEnabled(false);
    stop.setEnabled(false);
  }

  private void goBlooey(Throwable t) {
    AlertDialog.Builder builder=new AlertDialog.Builder(this);

    builder
      .setTitle("Exception!")
      .setMessage(t.toString())
      .setPositiveButton("OK", null)
      .show();
  }
}
```

In `onCreate()`, we wire up the three buttons to appropriate callbacks, then call `setup()`. In `setup()`, we create our `MediaPlayer`, set to play a clip we package in the project as a raw resource. We also configure the activity itself as the completion listener, so we find out when the clip is over. Note that, since we use the static `create()` method on `MediaPlayer`, we have already implicitly called `prepare()`, so we do not need to call that separately ourselves.

The buttons simply work the `MediaPlayer` and toggle each others' states, via appropriately-named callbacks. So, `play()` starts MediaPlayer playback, `pause()` pauses playback, and `stop()` stops playback and resets our `MediaPlayer` to play again. The `stop()` callback is also used for when the audio clip completes of its own accord.

To reset the MediaPlayer, the `stop()` callback calls `release()` on the existing `MediaPlayer` (to release its resources), then calls `setup()` again, discarding the used `MediaPlayer` and starting a fresh one.

The UI is nothing special, but we are more interested in the audio in this sample, anyway:

Figure 8. The AudioDemo sample application

Moving Pictures

Video clips get their own widget, the `VideoView`. Put it in a layout, feed it an MP4 video clip, and you get playback!

For example, take a look at this layout, from the `Media/Video` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <VideoView
```

```
    android:id="@+id/video"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
    />
</LinearLayout>
```

The layout is simply a full-screen video player. Whether it will use the full screen will be dependent on the video clip, its aspect ratio, and whether you have the device (or emulator) in portrait or landscape mode.

Wiring up the Java is almost as simple:

```
public class VideoDemo extends Activity {
  private VideoView video;
  private MediaController ctlr;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    getWindow().setFormat(PixelFormat.TRANSLUCENT);
    setContentView(R.layout.main);

    File clip=new File("/sdcard/test.mp4");

    if (clip.exists()) {
      video=(VideoView)findViewById(R.id.video);
      video.setVideoPath(clip.getAbsolutePath());

      ctlr=new MediaController(this);
      ctlr.setMediaPlayer(video);
      video.setMediaController(ctlr);
      video.requestFocus();
    }
  }
}
```

The biggest trick with `VideoView` is getting a video clip onto the device. While `VideoView` does support some streaming video, the requirements on the MP4 file are fairly stringent. If you want to be able to play a wider array of video clips, you need to have them on the device, preferably on an SD card.

The crude `VideoDemo` class assumes there is an MP4 file in `/sdcard/test.mp4` on your emulator. To make this a reality:

1. Find a clip, such as Aaron Rosenberg's *Documentaries and You* from Duke University's Center for the Study of the Public Domain's Moving Image Contest, which was used in the creation of this book
2. Use `mksdcard` (in the Android SDK's tools directory) to create a suitably-sized SD card image (e.g., `mksdcard 128M sd.img`)
3. Use the `-sdcard` switch when launching the emulator, providing the path to your SD card image, so the SD card is "mounted" when the emulator starts
4. Use the `adb push` command (or DDMS or the equivalent in your IDE) to copy the MP4 file into `/sdcard/test.mp4`

Once there, the Java code shown above will give you a working video player:

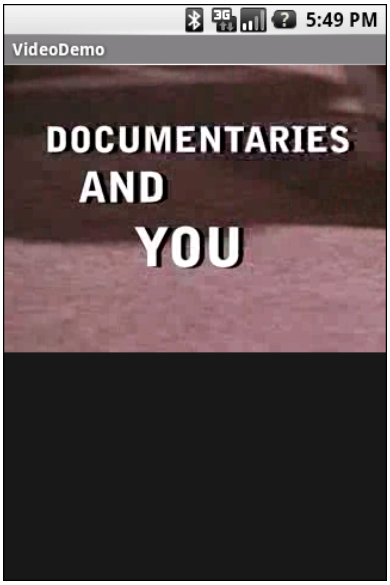


Figure 9. The VideoDemo sample application, showing a Creative Commons-licensed video clip

Tapping on the video will pop up the playback controls:

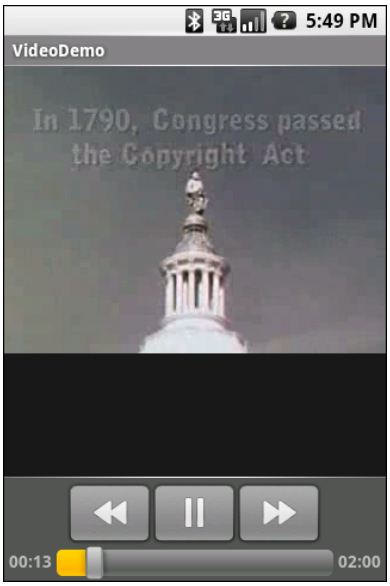


Figure 10. The VideoDemo sample application, with the media controls displayed

The video will scale based on space, as shown in this rotated view of the emulator (`<Ctrl>-<F12>`):

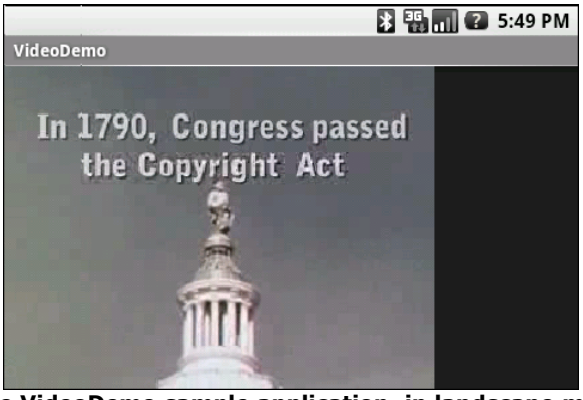


Figure 11. The VideoDemo sample application, in landscape mode, with the video clip scaled to fit

Note that playback may be rather jerky in the emulator, depending on the power of the PC that is hosting the emulator. For example, on a Pentium-M

1.6GHz PC, playback in the emulator is extremely jerky, while playback on the T-Mobile G1 is very smooth.

CHAPTER 7

Using the Camera

Most Android devices will have a camera, since they are fairly commonplace on mobile devices these days. You, as an Android developer, can take advantage of the camera, for everything from snapping tourist photos to scanning barcodes. For simple operations, the APIs needed to use the camera are fairly straight-forward, requiring a bit of boilerplate code plus your own unique application logic.

What is a problem is using the camera with the emulator. The emulator does not emulate a camera, nor is there a convenient way to pretend there are pictures via DDMS or similar tools. For the purposes of this chapter, it is assumed you have access to an actual Android-powered hardware device and can use it for development purposes.

Sneaking a Peek

First, it is fairly common for a camera-using application to support a preview mode, to show the user what the camera sees. This will help make sure the camera is lined up on the subject properly, whether there is sufficient lighting, etc.

So, let us take a look at how to create an application that shows such a live preview. The code snippets shown in this section are pulled from the `Camera/Preview` sample project.

The Permission

First, you need permission to use the camera. That way, when end users install your application off of the Internet, they will be notified that you intend to use the camera, so they can determine if they deem that appropriate for your application.

You simply need the `CAMERA` permission in your `AndroidManifest.xml` file, along with whatever other permissions your application logic might require. Here is the manifest from the `Camera/Preview` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.camera"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.CAMERA" />
    <application android:label="@string/app_name">
        <activity android:name=".PreviewDemo"
                  android:label="@string/app_name"
                  android:configChanges="keyboardHidden|orientation"
                  android:screenOrientation="landscape"
                  android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Also note a few other things about our `PreviewDemo` activity as registered in this manifest:

- We use `android:configChanges = "keyboardHidden|orientation"` to ensure we control what happens when the keyboard is hidden or exposed, rather than have Android rotate the screen for us
- We use `android:screenOrientation = "landscape"` to tell Android we are always in landscape mode. This is necessary because of a bit of a bug in the camera preview logic, such that it works best in landscape mode.

- We use `android:theme = "@android:style/Fullscreen"` to get rid of the title bar and status bar, so the preview is truly full-screen (e.g., 480x320 on a T-Mobile G1).

The SurfaceView

Next, you need a layout supporting a `SurfaceView`. `SurfaceView` is used as a raw canvas for displaying all sorts of graphics outside of the realm of your ordinary widgets. In this case, Android knows how to display a live look at what the camera sees on a `SurfaceView`, to serve as a preview pane.

For example, here is a full-screen `SurfaceView` layout as used by the `PreviewDemo` activity:

```
<?xml version="1.0" encoding="utf-8"?>
<android.view.SurfaceView
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/preview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
</android.view.SurfaceView>
```

The Camera

The biggest step, of course, is telling Android to use the camera service and tie a camera to the `SurfaceView` to show the actual preview. We will also eventually need the camera service to take real pictures, as will be described in the next section.

There are three major components to getting picture preview working:

1. The `SurfaceView`, as defined in our layout
2. A `SurfaceHolder`, which is a means of controlling behavior of the `SurfaceView`, such as its size, or being notified when the surface changes, such as when the preview is started
3. A `Camera`, obtained from the `open()` static method on the `Camera` class

To wire these together, we first need to:

- Get the `SurfaceHolder` for our `SurfaceView` via `getHolder()`
- Register a `SurfaceHolder.Callback` with the `SurfaceHolder`, so we are notified when the `SurfaceView` is ready or changes
- Tell the `SurfaceView` (via the `SurfaceHolder`) that it has the `SURFACE_TYPE_PUSH_BUFFERS` type (`setType()`) – this indicates something in the system will be updating the `SurfaceView` and providing the bitmap data to display

This gives us a configured `SurfaceView` (shown below), but we still need to tie in the `Camera`.

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

  preview=(SurfaceView)findViewById(R.id.preview);
  previewHolder=preview.getHolder();
  previewHolder.addCallback(surfaceCallback);
  previewHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
}
```

A `Camera` object has a `setPreviewDisplay()` method that takes a `SurfaceHolder` and, as you might expect, arranges for the camera preview to be displayed on the associated `SurfaceView`. However, the `SurfaceView` may not be ready immediately after being changed into `SURFACE_TYPE_PUSH_BUFFERS` mode. So, while the previous setup work could be done in `onCreate()`, you should wait until the `SurfaceHolder.Callback` has its `surfaceCreated()` method called, then register the `Camera`:

```
public void surfaceCreated(SurfaceHolder holder) {
  camera=Camera.open();
  camera.setPreviewDisplay(previewHolder);
}
```

Next, once the `SurfaceView` is set up and sized by Android, we need to pass configuration data to the `Camera`, so it knows how big to draw the preview. Since the preview pane is not a fixed size – it might vary based on hardware – we cannot safely pre-determine the size. It is simplest to wait for our

`SurfaceHolder.Callback` to have its `surfaceChanged()` method called, when we are told the size of the surface. Then, we can pour that information into a `Camera.Parameters` object, update the `Camera` with those parameters, and have the `Camera` show the preview images via `startPreview()`:

```
public void surfaceChanged(SurfaceHolder holder,
                           int format, int width,
                           int height) {
  Camera.Parameters parameters=camera.getParameters();

  parameters.setPreviewSize(width, height);
  camera.setParameters(parameters);
  camera.startPreview();
}
```

Eventually, the preview needs to stop. In this particular case, that will be as the activity is being destroyed. It is important to release the `Camera` at this time – for many devices, there is only one physical camera, so only one activity can be using it at a time. Our `SurfaceHolder.Callback` will be told, via `surfaceDestroyed()`, when it is being closed up, and we can stop the preview (`stopPreview()`), release the camera (`release()`), and let go of it (`camera = null`) at that point:

```
public void surfaceDestroyed(SurfaceHolder holder) {
  camera.stopPreview();
  camera.release();
  camera=null;
}
```

If you compile and run the `Camera/Preview` sample application, you will see, on-screen, what the camera sees.

Image Is Everything

Showing the preview imagery is nice and all, but it is probably more important to actually take a picture now and again. The previews show the user what the camera sees, but we still need to let our application know what the camera sees at particular points in time.

In principle, this is easy. Where things get a bit complicated comes with ensuring the application (and device as a whole) has decent performance, not slowing down to process the pictures.

The code snippets shown in this section are pulled from the `Camera/Picture` sample project, which builds upon the `Camera/Preview` sample shown in the previous section.

Asking for a Format

We need to tell the `Camera` what sort of picture to take when we decide to take a picture. The two options are raw and JPEG.

At least, that is the theory.

In practice, the T-Mobile G1 does not support raw output, only JPEG. So, we need to tell the `Camera` that we want JPEG output.

That is merely a matter of calling `setPictureFormat()` on the `Camera.Parameters` object when we configure our `Camera`, using the value `JPEG` to indicate that we, indeed, want JPEG:

```
public void surfaceChanged(SurfaceHolder holder,
                           int format, int width,
                           int height) {
  Camera.Parameters parameters=camera.getParameters();

  parameters.setPreviewSize(width, height);
  parameters.setPictureFormat(PixelFormat.JPEG);
  camera.setParameters(parameters);
  camera.startPreview();
}
```

Connecting the Camera Button

Somehow, your application will need to indicate when a picture should be taken. That could be via widgets on the UI, though in our samples here, the preview is full-screen.

An alternative is to use the camera hardware button. Like every hardware button other than the Home button, we can find out when the camera button is clicked via `onKeyDown()`:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode==KeyEvent.KEYCODE_CAMERA) {
    takePicture();

    return(true);
  }

  return(super.onKeyDown(keyCode, event));
}
```

Taking a Picture

Once it is time to take a picture, all you need to do is:

- Stop the preview
- Tell the `Camera` to `takePicture()`

The `takePicture()` method takes three parameters, all callback-style objects:

1. A "shutter" callback (`Camera.ShutterCallback`), which is notified when the picture has been captured by the hardware but the data is not yet available – you might use this to play a "camera click" sound
2. Callbacks to receive the image data, either in raw format or JPEG format

Since the T-Mobile G1 only supports JPEG output, and because we do not want to fuss with a shutter click, `PictureDemo` only passes in the third parameter to `takePicture()`:

```
private void takePicture() {
  camera.stopPreview();
  camera.takePicture(null, null, photoCallback);
}
```

The `Camera.PictureCallback` (`photoCallback`) needs to implement `onPictureTaken()`, which provides the picture data as a `byte[]`, plus the

`Camera` object that took the picture. At this point, it is safe to start up the preview again.

Plus, of course, it would be nice to do something with that byte array.

The catch is that the byte array is going to be large – the T-Mobile G1 has a 3-megapixel camera, and future hardware is more likely to have richer hardware than that. Writing that to flash, or sending it over the network, or doing just about anything with the data, will be slow. Slow is fine...so long as it is not on the UI thread.

That means we need to do a little more work.

The Job Queue Pattern

In theory, we could just fork a background thread to save off the image data or do whatever it is we wanted done with it. However, we could wind up with several such threads, particularly if we are sending the image over the Internet and do not have a fast connection to our destination server.

Another pattern is the work queue. We set up one background thread that simply monitors a job queue for work. When a job appears on the queue, the thread pops the job off the queue, does whatever the job needs to have done, then waits for another job. This means work can "stack up" if needed without having multiple background threads – the thread will catch up when activity quiets down.

In `PictureDemo`, we have such a work queue set up. The queue itself is a `LinkedBlockingQueue<Job>`, where `Job` is a local base class for all jobs that should go on the queue:

```
class Job {
  boolean stopThread() {
    return(false);
  }

  void process() {
    // no-op
```

```
  }
}
```

When the activity starts up, we can fire off a `Thread` on a `Runnable` that monitors the job queue for work:

```
Runnable qProcessor=new Runnable() {
  public void run() {
    while (true) {
      try {
        Job j=q.take();

        if (j.stopThread()) {
          break;
        }
        else {
          j.process();
        }
      }
      catch (InterruptedException e) {
        break;
      }
    }
  }
};
```

When the activity is shutting down, via `onDestroy()`, we can pop a `KillJob` on the queue, which causes the background thread to fall out of its watch-the-job-queue loop:

```
class KillJob extends Job {
  @Override
  boolean stopThread() {
    return(true);
  }
}
```

And, when we actually take a picture, we can bundle our image data into a `SavePhotoJob` and have our background queue deal with doing something useful with the picture. More on that below.

Saving the Picture

The `SavePhotoJob`, in principle, could do almost anything. The byte array is simply the JPEG itself, so the data could be written to a file, transformed,

sent to a Web service, converted into a `BitmapDrawable` for display on the screen or whatever.

In the case of `PictureDemo`, we take the simple approach of writing the JPEG file as `photo.jpg` in the root of the SD card:

```
class SavePhotoJob extends Job {
  byte[] jpeg=null;

  SavePhotoJob(byte[] jpeg) {
    this.jpeg=jpeg;
  }

  @Override
  void process() {
    File photo=new File(Environment.getExternalStorageDirectory(),
                        "photo.jpg");

    if (photo.exists()) {
      photo.delete();
    }

    try {
      FileOutputStream fos=new FileOutputStream(photo.getPath());

      fos.write(jpeg);
      fos.close();
    }
    catch (java.io.IOException e) {
      Log.e("PictureDemo", "Exception in photoCallback", e);
    }
  }
}
```

The byte array itself will be garbage collected once we are done saving it, so there is no explicit "free" operation we need to do to release that memory.

PART III – Advanced System

CHAPTER 8

Sensors

"Sensors" is Android's overall term for ways that Android can detect elements of the physical world around it, from magnetic flux to the movement of the device. Not all devices will have all possible sensors, and other sensors are likely to be added over time. In this chapter, we will explore what sensors are theoretically available and how to use a few of them that work on early Android devices like the T-Mobile G1.

The samples in this chapter assume that you have access to a piece of sensor-equipped Android hardware, such as a T-Mobile G1. The OpenIntents.org project has a sensor simulator which you can also use, though the use of this tool is not covered here.

The author would like to thank Sean Catlin for code samples that helped clear up confusion surrounding the use of sensors.

The Sixth Sense. Or Possibly the Seventh.

In theory, Android supports the following sensor types:

- An accelerometer, that tells you the motion of the device in space through all three dimensions
- An ambient light sensor, telling you how bright or dark the surroundings are

- A magnetic field sensor, to tell you where magnetic north is (unless some other magnetic field is nearby, such as from an electrical motor)
- An orientation sensor, to tell you how the device is positioned in all three dimensions
- A proximity sensor, to tell you how far the device is from some other specific object
- A temperature sensor, to tell you the temperature of the surrounding environment
- A tricorder sensor, to turn the device into "a fully functional Tricorder"

Clearly, not all of these possible sensors are available today, such as the last one. What definitely are available today on the T-Mobile G1 are the accelerometer, the magnetic field sensor, and the orientation sensor.

To access any of these sensors, you need a `SensorManager`, found in the `android.hardware` package. Like other aspects of Android, the `SensorManager` is a system service, and as such is obtained via the `getSystemService()` method on your `Activity` or other `Context`:

```
sensor=(SensorManager)getSystemService(Context.SENSOR_SERVICE);
```

Orienting Yourself

In principle, to find out which direction is north, you would use the magnetic flux sensor and go through a lovely set of calculations to figure out the appropriate direction.

Fortunately for us, Android did all that as part of the orientation sensor...so long as the device is held flat in the horizontal plane (e.g., on a level tabletop).

Akin to the location services, there is no way to ask the `SensorManager` what the current value of a sensor is. Instead, you need to hook up a

SensorListener and respond to changes in the sensor values. To do this, simply call registerListener() with your SensorListener and a bitmask of which sensors you want to hear from. For example, from the Sensor/Compass sample project, here is where we register our listener:

```
sensor.registerListener(listener,
                        SensorManager.SENSOR_ORIENTATION);
```

It is important to unregister the listener when the activity closes down; otherwise, the application will never really terminate and the listener will get updates indefinitely. To do this, just call unregisterListener() from a likely location, such as onDestroy():

```
@Override
public void onDestroy() {
  super.onDestroy();
  sensor.unregisterListener(listener);
}
```

Your SensorListener implementation will need two methods. The one you probably will not use that often is onAccuracyChanged(), when you will be notified as a given sensor's accuracy changes from SENSOR_STATUS_ACCURACY_HIGH to SENSOR_STATUS_ACCURACY_MEDIUM to SENSOR_STATUS_ACCURACY_LOW to SENSOR_STATUS_UNRELIABLE.

The one you will use more commonly is onSensorChanged(), where you are provided a float[] of values for the sensor. The tricky part is determining what these sensor values mean.

In the case of SENSOR_ORIENTATION, the first of the supplied values represents the orientation of the device in degrees off of magnetic north. 90 degrees means east, 180 means south, and 270 means west, just like on a regular compass.

In Sensor/Compass, we toss out 9 out of every 10 readings and update a TextView with the 10th reading, so the TextView doesn't get too "twitchy", changing all the time:

```
private SensorListener listener=new SensorListener() {
  public void onSensorChanged(int sensor, float[] values) {
    if (sensor==SensorManager.SENSOR_ORIENTATION) {
      if (++count==10) {
          degrees.setText(String.valueOf(values[0]));
          count=0;
      }
    }
  }

  public void onAccuracyChanged(int sensor, int accuracy) {
    // unused
  }
};
```

What you get is a trivial application showing where the top of the phone is pointing. Note that the sensor seems to take a bit to get initially stabilized, then will tend to lag actual motion a bit.

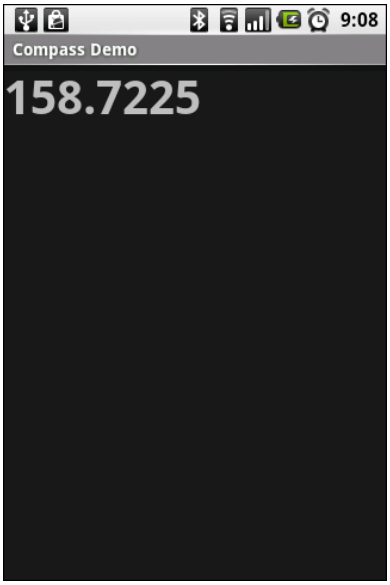


Figure 12. The CompassDemo application, showing a T-Mobile G1 pointing south-by-southeast

Steering Your Phone

In television commercials for other mobile devices, you may see them being used like a steering wheel, often times for playing a driving simulation game.

Android can do this too. You can see it in the `Sensor/Steering` sample application.

In the preceding section, we noted that `SENSOR_ORIENTATION` returns in the first value of the `float[]` the orientation of the phone, compared to magnetic north, if the device is horizontal. When the device is held like a steering wheel, the second value of the `float[]` will change as the device is "steered".

This sample application is very similar to the `Sensor/Compass` one shown in the previous section. The biggest change comes in the `SensorListener` implementation:

```
private SensorListener listener=new SensorListener() {
  public void onSensorChanged(int sensor, float[] values) {
    if (sensor==SensorManager.SENSOR_ORIENTATION) {
      float orientation=values[1];

      if (prevOrientation!=orientation) {
        if (prevOrientation<orientation) {
          steerLeft(orientation,
                    orientation-prevOrientation);
        }
        else {
          steerRight(orientation,
                     prevOrientation-orientation);
        }

        prevOrientation=values[1];
      }
    }
  }

  public void onAccuracyChanged(int sensor, int accuracy) {
    // unused
  }
};
```

Here, we track the previous orientation (`prevOrientation`) and call a `steerLeft()` or `steerRight()` method based on which direction the "wheel" is turned. For each, we provide the new current position of the wheel and the amount the wheel turned, measured in degrees.

The `steerLeft()` and `steerRight()` methods, in turn, simply dump their results to a "transcript": a `TextView` inside a `ScrollView`, set up to automatically keep scrolling to the bottom:

```
private void steerLeft(float position, float delta) {
  StringBuffer line=new StringBuffer("Steered left by ");

  line.append(String.valueOf(delta));
  line.append(" to ");
  line.append(String.valueOf(position));
  line.append("\n");
  transcript.setText(transcript.getText().toString()+line.toString());
  scroll.fullScroll(View.FOCUS_DOWN);
}

private void steerRight(float position, float delta) {
  StringBuffer line=new StringBuffer("Steered right by ");

  line.append(String.valueOf(delta));
  line.append(" to ");
  line.append(String.valueOf(position));
  line.append("\n");
  transcript.setText(transcript.getText().toString()+line.toString());
  scroll.fullScroll(View.FOCUS_DOWN);
}
```

The result is a log of the steering "events" as the device is turned like a steering wheel. Obviously, a real game would translate these events into game actions, such as changing your perspective of the driving course.

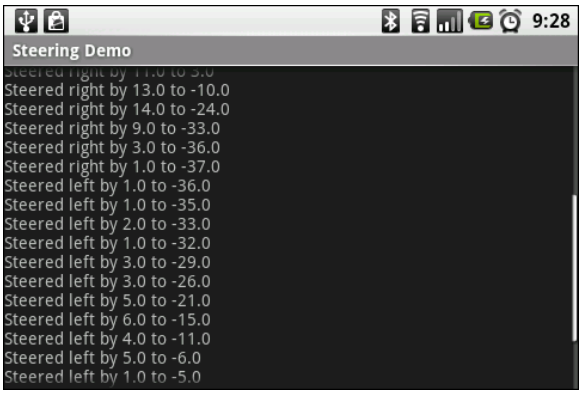


Figure 13. The SteeringDemo application

Do "The Shake"

Another demo you often see with certain other mobile devices is shaking the device to cause some on-screen effect, such as rolling dice or scrambling puzzle pieces.

Android can do this as well, as you can see in the `Sensor/Shaker` sample application, with our data provided by the accelerometer sensor (`SENSOR_ACCELEROMETER`).

What the accelerometer sensor provides is the accleration in each of three dimensions. At rest, the acceleration is equal to Earth's gravity (or the gravity of wherever you are, if you are not on Earth). When shaken, the acceleration should be higher than Earth's gravity – how much higher is dependent on how hard the device is being shaken. While the individual axes of acceleration might tell you, at any point in time, what direction the device is being shaken in, since a shaking action involves frequent constant changes in direction, what we really want to know is how fast the device is moving overall – a slow steady movement is not a shake, but something more aggressive is.

Once again, our UI output is simply a "transcript" `TextView` as before. This time, though, we separate out the actual shake-detection logic into a `Shaker` class which our `ShakerDemo` activity references, as shown below:

```
package com.commonsware.android.sensor;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ScrollView;
import android.widget.TextView;

public class ShakerDemo extends Activity
  implements Shaker.Callback {
  private Shaker shaker=null;
  private TextView transcript=null;
  private ScrollView scroll=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    transcript=(TextView)findViewById(R.id.transcript);
    scroll=(ScrollView)findViewById(R.id.scroll);

    shaker=new Shaker(this, 1.25d, 500, this);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    shaker.close();
  }

  public void shakingStarted() {
    Log.d("ShakerDemo", "Shaking started!");
    transcript.setText(transcript.getText().toString()+"Shaking started\n");
    scroll.fullScroll(View.FOCUS_DOWN);
  }

  public void shakingStopped() {
    Log.d("ShakerDemo", "Shaking stopped!");
    transcript.setText(transcript.getText().toString()+"Shaking stopped\n");
    scroll.fullScroll(View.FOCUS_DOWN);
  }
}
```

The `Shaker` takes four parameters:

- A `Context`, so we can get access to the `SensorManager` service
- An indication of how hard a shake should qualify as a shake, expressed as a ratio applied to Earth's gravity, so a value of `1.25`

means the shake has to be 25% stronger than gravity to be considered a shake

- An amount of time with below-threshold acceleration, after which the shake is considered "done"
- A `Shaker.Callback` object that will be notified when a shake starts and stops

While in this case, the callback methods (implemented on the `ShakerDemo` activity itself) simply log shake events to the transcript, a "real" application would, say, start an animation of dice rolling when the shake starts and end the animation shortly after the shake ends.

The `Shaker` simply converts the three individual acceleration components into a combined acceleration value (square root of the sum of the squares), then compares that value to Earth's gravity. If the ratio is higher than the supplied threshold, then we consider the device to be presently shaking, and we call the `shakingStarted()` callback method if the device was not shaking before. Once shaking ends, and time elapses, we call `shakingStopped()` on the callback object and assume that the shake has ended. A more robust implementation of `Shaker` would take into account the possibility that the sensor will not be updated for a while after the shake ends, though in reality, normal human movement will ensure that there are some sensor updates, so we can find out when the shaking ends.

```
package com.commonsware.android.sensor;

import android.content.Context;
import android.hardware.SensorListener;
import android.hardware.SensorManager;
import android.os.SystemClock;
import java.util.ArrayList;
import java.util.List;

public class Shaker {
  private SensorManager sensor=null;
  private long lastShakeTimestamp=0;
  private double threshold=1.0d;
  private long gap=0;
  private Shaker.Callback cb=null;

  public Shaker(Context ctxt, double threshold, long gap,
                Shaker.Callback cb) {
```

```
    this.threshold=threshold;
    this.gap=gap;
    this.cb=cb;

    sensor=(SensorManager)ctxt.getSystemService(Context.SENSOR_SERVICE);
    sensor.registerListener(listener,
                            SensorManager.SENSOR_ACCELEROMETER);
  }

  public void close() {
    sensor.unregisterListener(listener);
  }

  private void isShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp==0) {
      lastShakeTimestamp=now;

      if (cb!=null) {
        cb.shakingStarted();
      }
    }
    else {
      lastShakeTimestamp=now;
    }
  }

  private void isNotShaking() {
    long now=SystemClock.uptimeMillis();

    if (lastShakeTimestamp>0) {
      if (now-lastShakeTimestamp>gap) {
        lastShakeTimestamp=0;

        if (cb!=null) {
          cb.shakingStopped();
        }
      }
    }
  }

  public interface Callback {
    void shakingStarted();
    void shakingStopped();
  }

  private SensorListener listener=new SensorListener() {
    public void onSensorChanged(int sensor, float[] values) {
      if (sensor==SensorManager.SENSOR_ACCELEROMETER) {
        double netForce=Math.pow(values[SensorManager.DATA_X], 2.0);

        netForce+=Math.pow(values[SensorManager.DATA_Y], 2.0);
        netForce+=Math.pow(values[SensorManager.DATA_Z], 2.0);
```

```
        if (threshold<(Math.sqrt(netForce)/SensorManager.GRAVITY_EARTH)) {
          isShaking();
        }
        else {
          isNotShaking();
        }
      }
    }

    public void onAccuracyChanged(int sensor, int accuracy) {
      // unused
    }
  };
}
```

All the transcript shows, of course, is when shaking starts and stops:

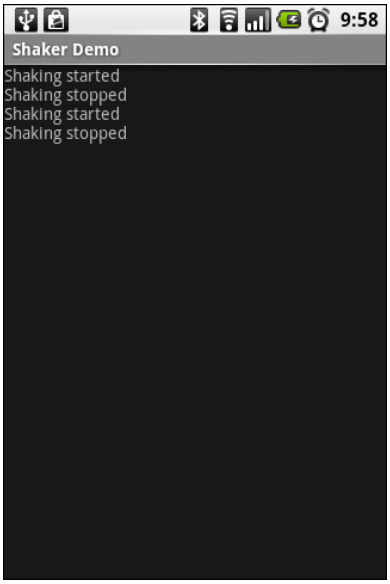


Figure 14. The ShakerDemo application, showing a pair of shakes

CHAPTER 9

Databases and Content Providers

In the abstract, working with SQLite databases and Android-style content providers is fairly straight-forward. Each supports a CRUD-style interface (`query()`, `insert()`, `update()`, `delete()`) using `Cursor` objects for query results. While implementing a `ContentProvider` is no picnic for non-SQLite data stores, everything else is fairly rote.

In reality, though, databases and content providers cause more than their fair share of hassles. Mostly, this comes from everything *outside* of simple CRUD operations, such as:

- How do we get a database into our application?
- How do we get data into our application on initial install? On an update?
- Where is the documentation for the built-in Android content providers?
- How do we deal with joins between data stores, such as merging contacts with our own database data?

In this chapter, we explore these issues, to show how you can better work with databases and content providers in the real world.

Distributed Data

Some databases used by Android applications naturally start empty. For example, a "password safe" probably has no passwords when initially launched by the user, and an expense-tracking application probably does not have any expenses recorded at the outset.

However, sometimes, there are databases that need to ship with an application that must be pre-populated with data. For example, you might be implementing an online catalog, with a database of items for sale installed with the application and updated as needed via calls to some Web service. The same structure would hold true for any sort of reference, from chemicals to word translations to historical sports records.

Unfortunately, there is no way to ship a database with data in it via the Android APK packaging mechanism. An APK is an executable blob, from the standpoint of Android and Dalvik. More importantly, it is stored read-only in a ZIP file, which makes updates to that data doubly impossible.

The next-best option is to ship your data with the application by some other means and load it into a newly-created database when the application is first run. This does involve two copies of the data: the original in your application and the working copy in the database. That may seem wasteful in terms of space. However, courtesy of ZIP compression, the original copy may not take up all that much space. Also, you can turn this into a feature, offering some sort of "reset" mechanism to reload the working database from the original if needed.

The challenge then becomes how to package the database contents into the APK and load it into the working database. Ideally, this involves as little work as possible from the developer, can fit into the existing build system, and can take advantage of existing database manipulation tools (versus, say, hand-writing hundreds of `SQL INSERT` statements).

Note that another possibility exists: package the binary SQLite database file in the APK (e.g., in `res/raw/`) and copy it into position using binary streams.

This assumes the SQLite database file your development environment would create is the same as what is expected by the SQLite engine baked into Android. This can work, but is likely to be more prone to versioning issues – for example, if your development environment is upgraded to a newer SQLite that has a slightly different file format.

SQLite: On-Device, On-Desktop

This becomes much simpler when you realize that Android uses SQLite for the database, and SQLite works on just about every platform you might need. It is trivial to work with SQLite databases on your development workstation, even easier than working with databases inside an Android emulator or device.

The plan, therefore, is to allow developers to create the database to be "shipped" as a SQLite database, then build tools that package the SQLite contents into the Android APK and turn it back into a database when the application needs it.

This allows developers to use whatever tools they want to manipulate the SQLite database, ranging from typical database management UIs to specialized conversion scripts to whatever.

To make this plan work, though, we need two bits of code:

1. We need something that extracts the data out of the SQLite database the developer has prepared and puts it someplace inside the Android APK
2. We need something that ties in with `SQLiteOpenHelper` that takes the APK-packaged data and turns it into an on-device database when the database is first accessed.

Exporting a Database

Fortunately, the `sqlite3` command-line executable that comes standard with SQLite offers a `.dump` command to dump the contents of a table as a series of

SQL statements: one to create the table, plus the necessary `SQL INSERT` statements to populate it. All we need to do is tie this into the build system, so the act of compiling the APK also deals with the database.

You can find some sample code that handles this in the `Database/Packager` sample application. Specifically:

- There is a SQLite database containing data in the `db/` project directory – in this case, it is the database from the `ContentProvider/Constants` project from *The Busy Coder's Guide to Android Development*
- There is a `package_db.rb` Ruby script that wraps around the `.dump` command to export the data
- There is a change to the `build.xml` Ant script to use this Ruby script

The Ruby Script

You may or may not be a fan of Ruby. While this sample code shows this utility as a Ruby script, rest assured that SQLite has interfaces to most programming languages (though its Java support is not the strongest), so you can create your own edition of this script in whatever language suits you.

The script is fairly short:

```
require 'rubygems'
require 'sqlite3'

Dir['db/*'].each do |path|
  db=SQLite3::Database.new(path)

  begin
    db.execute("SELECT name FROM sqlite_master WHERE type='table'") do |row|
      if ARGV.include?(row[0])
        puts `sqlite3 #{path} ".dump #{row[0]}"`
      end
    end
  ensure
    db.close
  end
end
```

It iterates over every file in the `db/` directory and opens each as a SQLite database. It then queries the database for the list of tables (`SELECT name FROM sqlite_master WHERE type = 'table'`). Any table matching a table name passed in on the command line is assumed to be one needing to be exported, so it prints to `stdout` the results of the `sqlite3 .dump` command, run on that database and table. We use `sqlite3` because there does not appear to be an API call that implements the `.dump` functionality.

To run this script, you need SQLite3 installed, with `sqlite3` in your PATH, and you need the Ruby interpreter. You also need to run it from the project directory, with a `db/` directory containing one or more database files.

The Ant Script

To take advantage of this Ruby script, we need to inject it into the build process.

Specifically, the `build.xml` for `Database/Packager` contains the following new Ant target:

```
<target name="package-db">
    <exec executable="ruby" output="res/raw/packaged_db.txt">
        <arg line="package_db.rb"/>
        <arg line="constants"/>
    </exec>
</target>
```

We invoke the `ruby` interpreter, providing it the path to the `package_db.rb` Ruby script and the name of the one table needing to be exported (`constants`). The results are placed in `res/raw/packaged_db.txt`.

This Ant task is then tied into the build chain by making the `package-res` task depend upon it:

```
<target name="package-res" depends="package-db">
    <available file="${asset-dir}" type="dir"
            property="res-target" value="and-assets" />
    <property name="res-target" value="no-assets" />
    <antcall target="package-res-${res-target}" />
</target>
```

The net result is that running Ant on this build script will run the Ruby script, which will dump the specified tables to a set of SQL statements:

```
BEGIN TRANSACTION;
CREATE TABLE constants (_id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, value
REAL);
INSERT INTO "constants" VALUES(1,'Gravity, Death Star I',3.53036142541896e-07);
INSERT INTO "constants" VALUES(2,'Gravity, Earth',9.80665016174316);
INSERT INTO "constants" VALUES(3,'Gravity, Jupiter',23.1200008392334);
INSERT INTO "constants" VALUES(4,'Gravity, Mars',3.71000003814697);
INSERT INTO "constants" VALUES(5,'Gravity, Mercury',3.70000004768372);
INSERT INTO "constants" VALUES(6,'Gravity, Moon',1.60000002384186);
INSERT INTO "constants" VALUES(7,'Gravity, Neptune',11.0);
INSERT INTO "constants" VALUES(8,'Gravity, Pluto',0.600000023841858);
INSERT INTO "constants" VALUES(9,'Gravity, Saturn',8.96000003814697);
INSERT INTO "constants" VALUES(10,'Gravity, Sun',275.0);
INSERT INTO "constants" VALUES(11,'Gravity, The Island',4.81516218185425);
INSERT INTO "constants" VALUES(12,'Gravity, Uranus',8.6899995803833);
INSERT INTO "constants" VALUES(13,'Gravity, Venus',8.86999988555908);
COMMIT;
```

In this case, the constants table is empty, so there are no `SQL INSERT` statements. However, you could easily add some rows to the `constants` table – perhaps constants not available in Android itself – and ship those along with the table schema.

Note that `build.xml` is generated by the Android `activitycreator`, so on SDK updates you will probably need to re-establish these changes to the file. Also, if you use Eclipse, you will need to find the appropriate hooks to integrate the packaging step into the build process. Of course, there is nothing to prevent you from manually running `.dump` on the appropriate tables, but anything that is not automated can be forgotten, leading to errors and confusion.

Loading the Exported Database

The other end of his process is to take the raw SQL stores in `res/raw/packaged_db.txt` and "inflate" it at runtime into a database. Since `SQLiteOpenHelper` is designed to handle such operations, it seems to make sense to implement this logic as a subclass. You can find such a class – `DatabaseInstaller` – in the `Database/Packager` sample project:

```
import android.content.Context;
import android.database.SQLException;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import java.io.*;


abstract class DatabaseInstaller extends SQLiteOpenHelper {
  abstract void handleInstallError(Throwable t);

  private Context ctxt=null;

  public DatabaseInstaller(Context context, String name,
                           SQLiteDatabase.CursorFactory factory,
                           int version)  {
    super(context, name, factory, version);

    this.ctxt=context;
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    try {
      InputStream stream=ctxt
                         .getResources()
                         .openRawResource(R.raw.packaged_db);
      InputStreamReader is=new InputStreamReader(stream);
      BufferedReader in=new BufferedReader(is);
      String str;

      while ((str = in.readLine()) != null) {
        if (!str.equals("BEGIN TRANSACTION;") && !str.equals("COMMIT;")) {
          db.execSQL(str);
        }
      }

      in.close();
    }
    catch (IOException e) {
      handleInstallError(e);
    }
  }
}
```

This class is abstract, expecting subclasses to implement both the `onUpgrade()` path from `SQLiteOpenHelper` and a `handleInstallError()` callback in case something fails during `onCreate()`.

Most of the smarts are found in `DatabaseInstaller`'s `onCreate()` implementation. Since `SQLiteDatabase` has no means to execute SQL

statements contained in an `InputStream`, we are stuck opening the `R.raw.packaged_db` resource and reading the statements out ourselves, one at a time.

However, the exported SQL will likely contain `BEGIN TRANSACTION;` and `COMMIT;` statements, since `sqlite3` expects that `sqlite3` itself would be used to re-executed the dumped SQL script. Since transactions are handled via API calls with `SQLiteDatabase`, we cannot execute `BEGIN TRANSACTION;` and `COMMIT;` statements via `execSQL()` without getting a "nested transaction" error. So, we skip those two statements and execute everything else, one line at a time.

The net result: `onCreate()` takes our raw SQL and turns it into a table in our on-device database.

Of course, to really use this, you will need to create a `DatabaseInstaller` subclass, such as `ConstantsInstaller`:

```
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.util.Log;


class ConstantsInstaller  extends DatabaseInstaller {
  public ConstantsInstaller(Context context, String name,
                            SQLiteDatabase.CursorFactory factory,
                            int version)  {
    super(context, name, factory, version);
  }

  void handleInstallError(Throwable t) {
    Log.e("Constants", "Exception installing database", t);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS constants");
    onCreate(db);
  }
}
```

The rest of this project is largely identical to the `ContentProvider/Constants` sample from *The Busy Coder's Guide to Android Development*.

One possible enhancement to `DatabaseInstaller` is to create our own transaction around the loop of `execSQL()` calls. This would improve performance dramatically, as otherwise, each `execSQL()` call is its own transaction. The proof of this is left to the reader as an exercise.

Examining Your Relationships

Android has a built-in contact manager, integrated with the phone dialer. You can work with the contacts via the `Contacts` content provider.

However, compared to content providers found in, say, simplified book examples, the `Contacts` content provider is rather intimidating. After all, there are 16 classes and 9 interfaces all involved in accessing this content provider. This section will attempt to illustrate some of the patterns for making use of `Contacts`.

Contact Permissions

Since contacts are privileged data, you need certain permissions to work with them. Specifically, you need the `READ_CONTACTS` permission to query and examine the `Contacts` content and `WRITE_CONTACTS` to add, modify, or remove contacts from the system.

For example, here is the manifest for the `Database/Contacts` sample application:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.database"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.READ_CONTACTS" />
    <application android:label="@string/app_name">
        <activity android:name=".ContactsDemo"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

```
    </application>
</manifest>
```

Pre-Joined Data

While the database underlying the `Contacts` content provider is private, one can imagine that it has several tables: one for people, one for their phone numbers, one for their email addresses, etc. These are tied together by typical database relations, most likely 1:N, so the phone number and email address tables would have a foreign key pointing back to the table containing information about people.

To simplify accessing all of this through the content provider interface, Android pre-joins queries against some of the tables. For example, one can query for phone numbers and get the contact name and other data along with the number, without having to somehow do a join operation yourself.

The Sample Activity

The `ContactsDemo` activity is simply a `ListActivity`, though it sports a `Spinner` to go along with the obligatory `ListView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <Spinner android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

The activity itself sets up a listener on the `Spinner` and toggles the list of information shown in the `ListView` when the `Spinner` value changes:

```
package com.commonsware.android.database;

import android.app.ListActivity;
import android.database.Cursor;
import android.os.Bundle;
import android.provider.Contacts;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;
import android.widget.Spinner;

public class ContactsDemo extends ListActivity
  implements AdapterView.OnItemSelectedListener {
  private static String[] options={"Contact Names",
                                   "Contact Names & Numbers",
                                   "Contact Names & Email Addresses"};
  private ListAdapter[] listAdapters=new ListAdapter[3];

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    initListAdapters();

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                            android.R.layout.simple_spinner_item,
                            options);

    aa.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }

  public void onItemSelected(AdapterView<?> parent,
                             View v, int position, long id) {
    setListAdapter(listAdapters[position]);
  }

  public void onNothingSelected(AdapterView<?> parent) {
    // ignore
  }

  private void initListAdapters() {
    listAdapters[0]=buildNameAdapter();
```

```
    listAdapters[1]=buildPhonesAdapter();
    listAdapters[2]=buildEmailAdapter();
  }

  private ListAdapter buildNameAdapter() {
    String[] PROJECTION=new String[] {  Contacts.People._ID,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=managedQuery(Contacts.People.CONTENT_URI,
                          PROJECTION, null, null,
                          Contacts.People.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_1,
                                   c,
                                   new String[] {
                                     Contacts.PeopleColumns.NAME
                                   },
                                   new int[] {
                                     android.R.id.text1
                                   }));
  }

  private ListAdapter buildPhonesAdapter() {
    String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                        Contacts.Phones.NAME,
                                        Contacts.Phones.NUMBER
                                      };
    Cursor c=managedQuery(Contacts.Phones.CONTENT_URI,
                          PROJECTION, null, null,
                          Contacts.Phones.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_2,
                                   c,
                                   new String[] {
                                     Contacts.Phones.NAME,
                                     Contacts.Phones.NUMBER
                                   },
                                   new int[] {
                                     android.R.id.text1,
                                     android.R.id.text2
                                   }));
  }

  private ListAdapter buildEmailAdapter() {
    String[] PROJECTION=new String[] {  Contacts.ContactMethods._ID,
                                        Contacts.ContactMethods.DATA,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=managedQuery(Contacts.ContactMethods.CONTENT_EMAIL_URI,
                          PROJECTION, null, null,
                          Contacts.ContactMethods.DEFAULT_SORT_ORDER);
```

```
    return(new SimpleCursorAdapter(  this,
                                     android.R.layout.simple_list_item_2,
                                     c,
                                     new String[] {
                                       Contacts.PeopleColumns.NAME,
                                       Contacts.ContactMethods.DATA
                                     },
                                     new int[] {
                                       android.R.id.text1,
                                       android.R.id.text2
                                     }));
  }
}
```

When the activity is first opened, it sets up three `Adapter` objects, one for each of three perspectives on the contacts data. The `Spinner` simply resets the list to use the `Adapter` associated with the `Spinner` value selected.

Accessing People

The first `Adapter` shows the names of all of the contacts. Since all the information we seek is in the contact itself, we can use the `CONTENT_URI` provider, retrieve all of the contacts in the default sort order, and pour them into a `SimpleCursorAdapter` set up to show each person on its own row:

```
private ListAdapter buildNameAdapter() {
  String[] PROJECTION=new String[] {  Contacts.People._ID,
                                      Contacts.PeopleColumns.NAME
                                    };
  Cursor c=managedQuery(Contacts.People.CONTENT_URI,
                        PROJECTION, null, null,
                        Contacts.People.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_1,
                                   c,
                                   new String[] {
                                     Contacts.PeopleColumns.NAME
                                   },
                                   new int[] {
                                     android.R.id.text1
                                   }));
}
```

Assuming you have some contacts in the database, they will appear when you first open the `ContactsDemo` activity, since that is the default perspective:

TBD – screenshot

Accessing Phone Numbers

Retrieving a list of contacts by their phone number can be done by querying the `CONTENT_URI` content provider:

```
private ListAdapter buildPhonesAdapter() {
  String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                      Contacts.Phones.NAME,
                                      Contacts.Phones.NUMBER
                                    };
  Cursor c=managedQuery(Contacts.Phones.CONTENT_URI,
                        PROJECTION, null, null,
                        Contacts.Phones.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_2,
                                   c,
                                   new String[] {
                                     Contacts.Phones.NAME,
                                     Contacts.Phones.NUMBER
                                   },
                                   new int[] {
                                     android.R.id.text1,
                                     android.R.id.text2
                                   }));
}
```

Since the documentation for `Contacts.Phones` shows that it incorporates `Contacts.PeopleColumns` and `Contacts.PhonesColumns`, we know we can get the phone number and the contact's name in one query, which is why both are included in our projection of columns to retrieve.

In this case, we still pour the results into a `SimpleCursorAdapter`, but we use a two-line layout (`android.R.layout.simple_list_item_2`) to show both the name and the phone number:

TBD – screenshot

Accessing Email Addresses

Similarly, to get a list of all the email addresses, we can use the `CONTENT_EMAIL_URI` content provider, which incorporates the `Contacts.ContactMethodsColumns` and `Contacts.PeopleColumns`, so we can get access to the contact name as well as the email address itself (`DATA`):

```
private ListAdapter buildEmailAdapter() {
  String[] PROJECTION=new String[] {  Contacts.ContactMethods._ID,
                                      Contacts.ContactMethods.DATA,
                                      Contacts.PeopleColumns.NAME
                                    };
  Cursor c=managedQuery(Contacts.ContactMethods.CONTENT_EMAIL_URI,
                        PROJECTION, null, null,
                        Contacts.ContactMethods.DEFAULT_SORT_ORDER);

  return(new SimpleCursorAdapter(  this,
                                   android.R.layout.simple_list_item_2,
                                   c,
                                   new String[] {
                                     Contacts.PeopleColumns.NAME,
                                     Contacts.ContactMethods.DATA
                                   },
                                   new int[] {
                                     android.R.id.text1,
                                     android.R.id.text2
                                   }));
}
```

Again, the results are displayed via a two-line `SimpleCursorAdapter`:

TBD – screenshot

Rummaging Through Your Phone Records

The `CallLog` content provider in Android gives you access to the calls associated with your phone: the calls you placed, the calls you received, and the calls that you missed. This is a much simpler structure than the `Contacts` content provider described in the previous section.

The columns available to you can be found in the `CallLog.Calls` class. The commonly-used ones include:

- NUMBER: the phone number associated with the call
- DATE: when the call was placed, in milliseconds-since-the-epoch format
- DURATION: how long the call lasted, in seconds
- TYPE: indicating if the call was incoming, outgoing, or missed

These, of course, are augmented by the stock BaseColumns, which CallLog.Calls inherits from.

So, for example, here is a projection used against the call log, from the JoinDemo activity in the Database/JoinCursor project:

```
private static String[] PROJECTION=new String[] { CallLog.Calls._ID,
                                                  CallLog.Calls.NUMBER,
                                                  CallLog.Calls.DATE,
                                                  CallLog.Calls.DURATION
                                                };
```

Here is where we get a Cursor on that projection, with the most-recent calls first in the list:

```
Cursor c=managedQuery(android.provider.CallLog.Calls.CONTENT_URI,
                      PROJECTION, null, null,
                      CallLog.Calls.DATE+" DESC");
```

Unlike contacts, the call log appears unmodifiable by Android applications. So while you can query the log, you cannot add your own calls, delete calls, etc.

Also note that, to access the call log, you need the READ_CONTACTS permission.

Come Together, Right Now

If you have multiple tables within a database, and you want a Cursor that represents a join of those tables, you can accomplish that simply through a well-constructed query. However, if you have multiple databases, or you wish to join data in your database with data from a third-party

ContentProvider, the join becomes significantly more difficult. You cannot simply construct a query, since SQLite has no facility (today) to query a ContentProvider, let alone join a ContentProvider's contents with those from native tables.

One solution is to do the join at the Cursor itself. Android's Cursors offer a fairly vanilla interface, and Android even supplies a CursorWrapper class that can handle much of the effort for us. In this section, we will examine the use of CursorWrapper to create a JoinCursor, blending data from a SQLite table with that from the CallLog.

Note that the implementation shown here is for illustrative purposes only. It may suffer from significant performance issues, particularly memory consumption, that would need to be addressed in a serious production application. If you are interested in perhaps pursuing an open source project to implement a better version of JoinCursor, contact the author.

Also note that there is a CursorJoiner class in the android.database package in the SDK. A CursorJoiner takes two Cursor objects plus a list of key columns, using the key columns to join the Cursor values together. This is more efficient but somewhat less flexible that the implemenation shown here.

CursorWrapper

As the name suggests, CursorWrapper wraps a Cursor object. Specifically, CursorWrapper implements the Cursor interface itself and delegates all of the interface's calls to the wrapped Cursor.

On the surface, this seems pointless. After all, if CursorWrapper simply serves as a pass-through to the Cursor, why not use the underlying Cursor directly?

The key is not CursorWrapper itself, but rather custom subclasses of CursorWrapper. You can then override certain Cursor methods, to perform work in addition to, or perhaps instead of, passing the call to the wrapped Cursor.

In this case, we want to create a `CursorWrapper` subclass that allows us to inject additional columns into the results. These columns will be the result of a join operation between a SQLite table and the `CallLog`.

Specifically, the Database/JoinCursor project adds "call notes" – a block of text about a specific call one made. You could use this concept in a contact management system, for example, to annotate what all was discussed in a call or otherwise document the call itself. Since `CallLog` is not modifiable and has no field for "call notes" anyway, we cannot store such notes in the `CallLog`. Instead, we store those notes in a call_notes SQLite table, mapping the `CallLog` row `_id` to the note.

For simplicity, this example will assume that there are 0 or 1 notes per call, not several. That allows the `JoinCursor` to simply inject the call note into the `CallLog Cursor` results, without having to worry about dealing with several possible notes. We do, however, need to deal with the case where the call does not yet have a note.

Implementing a JoinCursor

A `JoinCursor` is a relatively complex class. Some of that complexity is due to repeated boilerplate code, and some is due to the problem being solved.

What we need the `JoinCursor` to do is:

- Override `Cursor`-related methods that involve the columns
- Check to see if there is a note for the current row
- Adjust the results of the method to accomodate the possibility (or reality) of a note

You can see an implementation of this in the `JoinCursor` class in the `Database/JoinCursor` project:

```
import android.content.ContentValues;
import android.database.Cursor;
import android.database.CursorWrapper;
import java.util.LinkedHashMap;
```

```
import java.util.Map;

class JoinCursor extends CursorWrapper {
  private I_JoinHandler join=null;
  private JoinCache cache=new JoinCache(100);

  JoinCursor(Cursor main, I_JoinHandler join) {
    super(main);

    this.join=join;
  }

  public int getColumnCount() {
    return(super.getColumnCount()+join.getColumnNames().length);
  }

  public int getColumnIndex(String columnName) {
    for (int i=0;i<join.getColumnNames().length;i++) {
      if (columnName.equals(join.getColumnNames()[i])) {
        return(super.getColumnCount()+i);
      }
    }

    return(super.getColumnIndex(columnName));
  }

  public int getColumnIndexOrThrow(String columnName) {
    for (int i=0;i<join.getColumnNames().length;i++) {
      if (columnName.equals(join.getColumnNames()[i])) {
        return(super.getColumnCount()+i);
      }
    }

    return(super.getColumnIndexOrThrow(columnName));
  }

  public String getColumnName(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      return(join.getColumnNames()[columnIndex-super.getColumnCount()]);
    }

    return(super.getColumnName(columnIndex));
  }

  public byte[] getBlob(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsByteArray(join.getColumnNames()[offset]));
    }

    return(super.getBlob(columnIndex));
  }
```

```
public double getDouble(int columnIndex) {
  if (columnIndex>=super.getColumnCount()) {
    ContentValues extras=cache.get(join.getCacheKey(this));
    int offset=columnIndex-super.getColumnCount();

    return(extras.getAsDouble(join.getColumnNames()[offset]));
  }

  return(super.getDouble(columnIndex));
}

public float getFloat(int columnIndex) {
  if (columnIndex>=super.getColumnCount()) {
    ContentValues extras=cache.get(join.getCacheKey(this));
    int offset=columnIndex-super.getColumnCount();

    return(extras.getAsFloat(join.getColumnNames()[offset]));
  }

  return(super.getFloat(columnIndex));
}

public int getInt(int columnIndex) {
  if (columnIndex>=super.getColumnCount()) {
    ContentValues extras=cache.get(join.getCacheKey(this));
    int offset=columnIndex-super.getColumnCount();

    return(extras.getAsInteger(join.getColumnNames()[offset]));
  }

  return(super.getInt(columnIndex));
}

public long getLong(int columnIndex) {
  if (columnIndex>=super.getColumnCount()) {
    ContentValues extras=cache.get(join.getCacheKey(this));
    int offset=columnIndex-super.getColumnCount();

    return(extras.getAsLong(join.getColumnNames()[offset]));
  }

  return(super.getLong(columnIndex));
}

public short getShort(int columnIndex) {
  if (columnIndex>=super.getColumnCount()) {
    ContentValues extras=cache.get(join.getCacheKey(this));
    int offset=columnIndex-super.getColumnCount();

    return(extras.getAsShort(join.getColumnNames()[offset]));
  }

  return(super.getShort(columnIndex));
```

```
  }

  public String getString(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.getAsString(join.getColumnNames()[offset]));
    }

    return(super.getString(columnIndex));
  }

  public boolean isNull(int columnIndex) {
    if (columnIndex>=super.getColumnCount()) {
      ContentValues extras=cache.get(join.getCacheKey(this));
      int offset=columnIndex-super.getColumnCount();

      return(extras.get(join.getColumnNames()[offset])==null);
    }

    return(super.isNull(columnIndex));
  }

  public boolean requery() {
    cache.clear();

    return(super.requery());
  }

  class JoinCache extends LinkedHashMap<String, ContentValues> {
    private int capacity=100;

    JoinCache(int capacity) {
      super(capacity+1, 1.1f, true);
      this.capacity=capacity;
    }

    protected boolean removeEldestEntry(Entry<String, ContentValues> eldest) {
      return(size()>capacity);
    }

    ContentValues get(String key) {
      ContentValues result=super.get(key);

      if (result==null) {
        result=join.getJoin(JoinCursor.this);
        put(key, result);
      }

      return(result);
    }
  }
}
```

JoinCursor, when instantiated, gets both the Cursor to wrap and an I_JoinHandler instance. The join handler is responsible for getting the extra columns for a given row:

```
import android.content.ContentValues;
import android.database.Cursor;
import java.util.Map;

public interface I_JoinHandler {
  String[] getColumnNames();
  String getCacheKey(Cursor c);
  ContentValues getJoin(Cursor c);
}
```

Most of JoinCursor is then using the I_JoinHandler information to adjust the results of various Cursor methods. For example:

- getColumnCount() returns the sum of the Cursor's column count and the number of extra columns returned by the join handler
- getColumnIndex() and kin need to search through the join handler's columns as well as the Cursor's to find the match, if any
- getInt(), isNull(), and kin need to support retrieving values from both the Cursor and the join handler

To improve performance, JoinCursor keeps a cache of the extra values for requested rows, using an "LRU cache"-style LinkedHashMap and an inner JoinCache class. The JoinCache keeps the ContentValues returned by I_JoinHandler on a getJoin() call, representing the extra columns (if any) for that particular Cursor row. Since we are caching data, however, we need to flush that cache sometimes; in this case, we override requery() to flush the cache if the Cursor itself is being proactively updated.

Using a JoinCursor

To use a JoinCursor, of course, you need an implementation of I_JoinHandler, such as this one from the JoinDemo activity:

```
I_JoinHandler join=new I_JoinHandler() {
  String[] columns={NOTE_ID, NOTE};
```

```
  public String[] getColumnNames() {
      return(columns);
  }

  public String getCacheKey(Cursor c) {
      return(String.valueOf(c.getInt(c.getColumnIndex(CallLog.Calls._ID))));
  }

  public ContentValues getJoin(Cursor c) {
    String[] args={getCacheKey(c)};
    Cursor j=getDb().rawQuery("SELECT _ID, note FROM call_notes WHERE
call_id=?", args);
    ContentValues result=new ContentValues();

    j.moveToFirst();

    if (j.isAfterLast()) {
      result.put(columns[0], -1);
      result.put(columns[1], (String)null);
    }
    else {
      result.put(columns[0], j.getInt(0));
      result.put(columns[1], j.getString(1));
    }

    j.close();

    return(result);
  }
};
```

The columns are a fixed pair (the note's ID and the note itself). These are retrieved via `getJoin()` from the `call_notes` SQLite table. The call notes themselves are keyed by the call's own `_id`, which is also used as the key for the `JoinCursor`'s cache of results. The net effect is that we only ever retrieve a note once for a given call, at least until a `requery()`. And, if there is no note for the call, we use a `null` note to indicate that we are, indeed, note-free for this call.

The note information is then used by our `CursorAdapter` subclass (`CallPlusAdapter`) and its associated `ViewWrapper`, also found in the `JoinDemo` activity:

```
class CallPlusAdapter extends CursorAdapter {
  CallPlusAdapter(Cursor c) {
    super(JoinDemo.this, c);
  }
```

```
  @Override
  public void bindView(View row, Context ctxt,
                       Cursor c) {
    ViewWrapper wrapper=(ViewWrapper)row.getTag();

    wrapper.update(c);
  }

  @Override
  public View newView(Context ctxt, Cursor c,
                      ViewGroup parent) {
    LayoutInflater inflater=getLayoutInflater();

    View row=inflater.inflate(R.layout.row, null);
    ViewWrapper wrapper=new ViewWrapper(row);

    row.setTag(wrapper);
    wrapper.update(c);

    return(row);
  }
}

class ViewWrapper {
  View base;
  TextView number=null;
  TextView duration=null;
  TextView time=null;
  ImageView icon=null;

  ViewWrapper(View base) {
    this.base=base;
  }

  TextView getNumber() {
    if (number==null) {
      number=(TextView)base.findViewById(R.id.number);
    }

    return(number);
  }

  TextView getDuration() {
    if (duration==null) {
      duration=(TextView)base.findViewById(R.id.duration);
    }

    return(duration);
  }

  TextView getTime() {
    if (time==null) {
      time=(TextView)base.findViewById(R.id.time);
    }
```

```
    return(time);
  }

  ImageView getIcon() {
    if (icon==null) {
      icon=(ImageView)base.findViewById(R.id.note);
    }

    return(icon);
  }

  void update(Cursor c) {
    getNumber().setText(c.getString(c.getColumnIndex(CallLog.Calls.NUMBER)));
    getTime().setText(FORMAT.format(c.getInt(c.getColumnIndex(CallLog.Calls.DATE
))));
    getDuration().setText(c.getString(c.getColumnIndex(CallLog.Calls.DURATION))
+" seconds");

    String note=c.getString(c.getColumnIndex(NOTE));

    if (note!=null && note.length()>0) {
      getIcon().setVisibility(View.VISIBLE);
    }
    else {
      getIcon().setVisibility(View.GONE);
    }
  }
}
```

Mostly, we are populating a row to go in a `ListView` based off of the call data (e.g., duration). However, if there is a non-`null` note, we also display an icon in the row, indicating that a note is available.

The `JoinDemo` activity itself is just a `ListActivity`, using the `CallPlusAdapter` and the `CallLog` `Cursor` we saw in the previous section:

```
import android.app.ListActivity;
import android.content.ContentValues;
import android.content.Context;
import android.content.Intent;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.provider.CallLog;
import android.view.View;
import android.view.ViewGroup;
import android.view.LayoutInflater;
import android.widget.CursorAdapter;
import android.widget.ImageView;
```

```
import android.widget.ListView;
import android.widget.TextView;
import java.text.SimpleDateFormat;

public class JoinDemo extends ListActivity {
  public static String NOTE="_NOTE";
  private static String NOTE_ID="NOTE_ID";
  private static String[] PROJECTION=new String[] { CallLog.Calls._ID,
                                                    CallLog.Calls.NUMBER,
                                                    CallLog.Calls.DATE,
                                                    CallLog.Calls.DURATION
                                                  };
  private static SimpleDateFormat FORMAT=new SimpleDateFormat("MM/d h:mm a");
  private Cursor cursor=null;
  private int noteColumn=-1;
  private int idColumn=-1;
  private int noteIdColumn=-1;
  private SQLiteDatabase db=null;

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

    Cursor c=managedQuery(android.provider.CallLog.Calls.CONTENT_URI,
                          PROJECTION, null, null,
                          CallLog.Calls.DATE+" DESC");

    cursor=new JoinCursor(c, join);
    noteColumn=cursor.getColumnIndex(NOTE);
    idColumn=cursor.getColumnIndex(CallLog.Calls._ID);
    noteIdColumn=cursor.getColumnIndex(NOTE_ID);
    setListAdapter(new CallPlusAdapter(cursor));
  }

  @Override
  public void onResume() {
    super.onResume();

    cursor.requery();
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    if (db!=null) {
      db.close();
    }
  }

  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    cursor.moveToPosition(position);
```

```
    String note=cursor.getString(noteColumn);

    if (note==null || note.length()==0) {
      Intent i=new Intent(this, NoteEditor.class);

      i.putExtra(NOTE, note);
      i.putExtra("call_id", cursor.getInt(idColumn));
      i.putExtra("note_id", cursor.getInt(noteIdColumn));
      startActivityForResult(i, 1);
    }
    else {
      Intent i=new Intent(this, NoteActivity.class);

      i.putExtra(NOTE, note);
      startActivity(i);
    }
  }
```

When the user clicks on a row, depending on whether there is a note, we either spawn a `NoteEditor` (to create a new note) or a `NoteActivity` (to view an existing note). In a real implementation of this functionality, of course, we would allow users to edit existing notes, delete notes, and the like, all of which is skipped in this simplified sample application.

Visually, the activity does not look like much, but you will see the note icon on calls containing notes (with some phone numbers smudged for privacy):

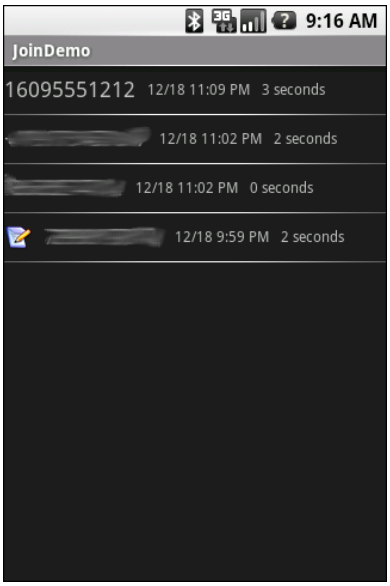


Figure 15. The JoinCursor sample application, showing one call with a note

CHAPTER 10

Using System Services

Get Alarmed

Meeting the User's Preference

Get Set

CHAPTER 11

Handling System Events

Get Moving, First Thing

I Sense a Connection Between Us...

Feeling Drained

CHAPTER 12

Your Own (Advanced) Services

In *The Busy Coder's Guide to Android Development*, we covered how to create and consume services. Now, we can get into some more interesting facets of service implementations, notably remote services, so your service can serve activities outside of your application.

Service From Afar

Remote services are nothing particularly special.

No, really.

Services, in general, expose some sort of API, perhaps through AIDL, to consuming clients (activities, other services, etc.). A remote service just means the service in question is running in some other process than the consumer of that service.

AIDL is designed to marshal its parameters and transport them across process boundaries, which is why there are so many quirky rules about what you can and cannot pass as parameters to your AIDL-defined APIs. With AIDL handling the cross-process smarts for you, implementing a remote service is not significantly different than implementing a local service.

The trick – and, of course, there is always a trick – is in determining how to connect to the remote service from the client.

Our sample applications – shown in the `AdvServices/RemoteService` and `AdvServices/RemoteClient` sample projects – convert our Beanshell demo from *The Busy Coder's Guide to Android Development* into a remote service. If you actually wanted to use scripting in an Android application, with scripts loaded off of the Internet, isolating their execution into a service might not be a bad idea. In the service, those scripts are sandboxed, only able to access files and APIs available to that service. The scripts cannot access your own application's databases, for example. If the script-executing service is kept tightly controlled, it minimizes the mischief a rogue script could possibly do.

Service Names

To bind to a service's AIDL-defined API, you need to craft an Intent that can identify the service in question. In the case of a local service, that Intent can use the local approach of directly referencing the service class.

Obviously, that is not possible in a remote service case, where the service class is not in the same process, and may not even be known by name to the client.

When you define a service to be used by remote, you need to add an intent-filter element to your service declaration in the manifest, indicating how you want that service to be referred to by clients. The manifest for `RemoteService` is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.advservice"
      android:versionCode="1"
      android:versionName="1.0">
  <application android:label="@string/app_name">
    <service android:name=".BshService">
      <intent-filter>
        <action android:name="com.commonsware.android.advservice.IScript" />
      </intent-filter>
```

```
    </service>
  </application>
</manifest>
```

Here, we say that the service can be identified by the name `com.commonsware.android.advservice.IScript`. So long as the client uses this name to identify the service, it can bind to that service's API.

In this case, the name is not an implementation, but the AIDL API, as you will see below. In effect, this means that so long as some service exists on the device that implements this API, the client will be able to bind to something.

The Service

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, `IScript`:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
  void executeScript(String script);
}
```

And there is the actual service class itself, `BshService`:

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script) {
      executeScriptImpl(script);
    }
  };
  private Interpreter i=new Interpreter();

  @Override
```

```
  public void onCreate() {
    super.onCreate();

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(binder);
  }

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

  private void executeScriptImpl(String script) {
    try {
      i.eval(script);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }
}
```

If you have seen the service and Beanshell samples in then this implementation will seem familiar. The biggest thing to note is that the service returns no result and handles any errors locally. Hence, the client will not get any response back from the script – the script will just run. In a real implementation, this would be silly, and we will work to rectify this later in this chapter.

Also note that, in this implementation, the script is executed directly by the service on the calling thread. One might think this is not a problem, since the service is in its own process and, therefore, cannot possibly be using the client's UI thread. However, AIDL IPC calls are synchronous, so the client will still block waiting for the script to be executed. This too will be corrected later in this chapter.

The Client

The client – `BshServiceDemo` out of `AdvServices/RemoteClient` – is a fairly straight-forward mashup of the service and Beanshell clients, with two twists:

```
package com.commonsware.android.advservice.client;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import com.commonsware.android.advservice.IScript;

public class BshServiceDemo extends Activity {
  private IScript service=null;
  private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
      service=IScript.Stub.asInterface(binder);
    }

    public void onServiceDisconnected(ComponentName className) {
      service=null;
    }
  };

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);

    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          service.executeScript(src);
        }
        catch (android.os.RemoteException e) {
          AlertDialog.Builder builder=
                    new AlertDialog.Builder(BshServiceDemo.this);
```

```
          builder
            .setTitle("Exception!")
            .setMessage(e.toString())
            .setPositiveButton("OK", null)
            .show();
        }
      }
    });

    bindService(new Intent(IScript.class.getName()),
                svcConn, Context.BIND_AUTO_CREATE);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    unbindService(svcConn);
  }
}
```

One twist is that the client needs its own copy of `IScript.aidl`. After all, it is a totally separate application, and therefore does not share source code with the service. In a production environment, we might craft and distribute a JAR file that contains the `IScript` classes, so both client and service can work off the same definition (see the upcoming chapter on reusable components). For now, we will just have a copy of the AIDL.

Then, the `bindService()` call uses a slightly different `Intent`, one that references the name of the AIDL interface's class implementation. That happens to be the name the service is registered under, and that is the glue that allows the client to find the matching service.

If you compile both applications and upload them to the device, then start up the client, you can enter in Beanshell code and have it be executed by the service. Note, though, that you cannot perform UI operations (e.g., raise a `Toast`) from the service. If you choose some script that is long-running, you will see that the Go! button is blocked until the script is complete:

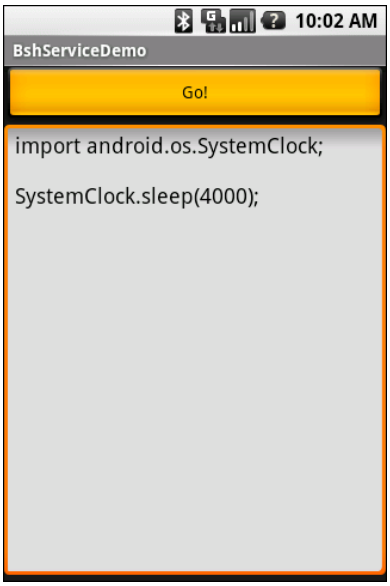


Figure 16. The BshServiceDemo application, running a long script

Servicing the Service

The preceding section outlined two flaws in the implementation of the Beanshell remote service:

1. The client received no results from the script execution
2. The client blocked waiting for the script to complete

If we were not worried about the blocking-call issue, we could simply have the `executeScript()` exported API return some sort of result (e.g., `toString()` on the result of the Beanshell `eval()` call). However, that would not solve the fact that calls to service APIs are synchronous even for remote services.

Another approach would be to pass some sort of callback object with `executeScript()`, such that the server could run the script asynchronously and invoke the callback on success or failure. This, though, implies that there is some way to have the activity export an API to the service.

Fortunately, this is eminently doable, as you will see in this section, and the accompanying samples (`AdvServices/RemoteServiceEx` and `AdvServices/RemoteClientEx`).

Callbacks via AIDL

AIDL does not have any concept of direction. It just knows interfaces and stub implementations. In the preceding example, we used AIDL to have the service flesh out the stub implementation and have the client access the service via the AIDL-defined interface. However, there is nothing magic about services implementing and clients accessing – it is equally possible to reverse matters and have the client implement something the service uses via an interface.

So, for example, we could create an `IScriptResult.aidl` file:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScriptResult {
  void success(String result);
  void failure(String error);
}
```

Then, we can augment `IScript` itself, to pass an `IScriptResult` with `executeScript()`:

```
package com.commonsware.android.advservice;

import com.commonsware.android.advservice.IScriptResult;

// Declare the interface.
interface IScript {
  void executeScript(String script, IScriptResult cb);
}
```

Notice that we need to specifically import `IScriptResult`, just like we might import some "regular" Java interface. And, as before, we need to make sure the client and the server are working off of the same AIDL definitions, so these two AIDL files need to be replicated across each project.

But other than that one little twist, this is all that is required, at the AIDL level, to have the client pass a callback object to the service: define the AIDL for the callback and add it as a parameter to some service API call.

Of course, there is a little more work to do on the client and server side to make use of this callback object.

Revising the Client

On the client, we need to implement an `IScriptResult`. On `success()`, we can do something like raise a `Toast`; on `failure()`, we can perhaps show an `AlertDialog`.

The catch is that we cannot be certain we are being called on the UI thread in our callback object.

So, the safest way to do that is to make the callback object use something like `runOnUiThread()` to ensure the results are displayed on the UI thread:

```
 builder
    .setTitle("Exception!")
    .setMessage(error)
    .setPositiveButton("OK", null)
    .show();
}

private final IScriptResult.Stub callback=new IScriptResult.Stub() {
  public void success(final String result) {
    runOnUiThread(new Runnable() {
      public void run() {
        successImpl(result);
      }
    });
  }

  public void failure(final String error) {
```

The work of actually showing the `Toast` or `AlertDialog` are delegated to `successImpl()` and `failureImpl()` methods:

```
@Override
public void onDestroy() {
```

```
  super.onDestroy();

  unbindService(svcConn);
}

private void successImpl(String result) {
  Toast
    .makeText(BshServiceDemo.this, result, Toast.LENGTH_LONG)
    .show();
}

private void failureImpl(String error) {
  AlertDialog.Builder builder=
            new AlertDialog.Builder(BshServiceDemo.this);
```

And, of course, we need to update our call to `executeScript()` to pass the callback object to the remote service:

```
@Override
public void onCreate(Bundle icicle) {
  super.onCreate(icicle);
  setContentView(R.layout.main);

  Button btn=(Button)findViewById(R.id.eval);
  final EditText script=(EditText)findViewById(R.id.script);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
      String src=script.getText().toString();

      try {
        service.executeScript(src, callback);
      }
      catch (android.os.RemoteException e) {
        failureImpl(e.toString());
      }
    }
  });

  bindService(new Intent(IScript.class.getName()),
              svcConn, Context.BIND_AUTO_CREATE);
}
```

Revising the Service

The service also needs changing, to both execute the scripts asynchronously and use the supplied callback object for the end results of the script's execution.

As was demonstrated in the chapter on `Camera`, `BshService` from `AdvServices/RemoteServiceEx` uses the `LinkedBlockingQueue` pattern to manage a background thread. An `ExecuteScriptJob` wraps up the script and callback; when the job is eventually processed, it uses the callback to supply the results of the `eval()` (on success) or the message of the `Exception` (on failure):

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import java.util.concurrent.LinkedBlockingQueue;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script, IScriptResult cb) {
      executeScriptImpl(script, cb);
    }
  };
  private Interpreter i=new Interpreter();
  private LinkedBlockingQueue<Job> q=new LinkedBlockingQueue<Job>();

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

    new Thread(qProcessor).start();

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(binder);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    q.add(new KillJob());
  }
```

```
  private void executeScriptImpl(String script,
                                 IScriptResult cb) {
    q.add(new ExecuteScriptJob(script, cb));
  }

  Runnable qProcessor=new Runnable() {
    public void run() {
      while (true) {
        try {
          Job j=q.take();

          if (j.stopThread()) {
            break;
          }
          else {
            j.process();
          }
        }
        catch (InterruptedException e) {
          break;
        }
      }
    }
  };

  class Job {
    boolean stopThread() {
      return(false);
    }

    void process() {
      // no-op
    }
  }

  class KillJob extends Job {
    @Override
    boolean stopThread() {
      return(true);
    }
  }

  class ExecuteScriptJob extends Job {
    IScriptResult cb;
    String script;

    ExecuteScriptJob(String script, IScriptResult cb) {
      this.script=script;
      this.cb=cb;
    }

    void process() {
      try {
        cb.success(i.eval(script).toString());
```

```
      }
      catch (Throwable e) {
        Log.e("BshService", "Error executing script", e);

        try {
          cb.failure(e.getMessage());
        }
        catch (Throwable t) {
          Log.e("BshService",
                "Error returning exception to client",
                t);
        }
      }
    }
  }
}
```

Notice that the service's own API just needs the `IScriptResult` parameter, which can be passed around and used like any other Java object. The fact that it happens to cause calls to be made synchronously back to the remote client is invisible to the service.

The net result is that the client can call the service and get its results without tying up the client's UI thread.

PART IV – Advanced Development

CHAPTER 13

Reusable Components

In the world of Java outside of Android, reusable components rule the roost. Whether they are simple JARs, are tied in via inversion-of-control (IoC) containers like Spring, or rely on enterprise service buses like Mule, reusable Java components are a huge portion of the overall Java ecosystem. Even full-fledged applications, like Eclipse or NetBeans, are frequently made up of a number of inter-locking components, many of which are available for others to use in their own applications.

In an ideal world, Android will evolve similarly, particularly given its reliance upon the Java programming language. This begs the question: what are the best ways to package code into a reusable component? Or, perhaps more basic: what are the possibilities for making reusable components? In this chapter, we will review two techniques introduced in *BCG to Android* – `Intents` and services – plus describe how you can create a good old-fashioned JAR containing Android-specific code.

Pick Up a JAR

A Java JAR is simplicity incarnate: a ZIP archive of classes compiled to bytecode, plus an optional manifest or other resources. While the JAR as a packaging method is imperfect – dealing with dependencies can be no fun – it is still a very easy way to bundle Java logic into a discrete item that can be uploaded, downloaded, installed, integrated, and used.

The challenge with Android, though, is that an Android application does not use traditional JVM bytecodes, but rather bytecodes for the Dalvik VM. There is a whole step in the build process that deals with converting JVM bytecodes to Dalvik ones. At first glance, this might seem to make creating an Android JAR more difficult. In reality, it is not significantly more different than creating a JAR designed to plug into some other large framework, such as an IoC container or a rich client engine like Eclipse RCP.

But first, it helps to understand how an Android application is built, before we translate that process to build a JAR of Android code.

The Android Build Process

Roughly speaking, these are the steps that take a set of Java source files and related resources and create an APK:

1. Run `aapt` to create `R.java` and related files, so you can access your resources by identifier from within Java source code
2. Run `aidl` to create stubs and interfaces for any AIDL you have created for use by your services
3. Run the `javac` compiler, for whatever version of Java you have installed on your development machine, to create JVM bytecode for your classes
4. Run `dx` to convert the JVM bytecode into Dalvik VM bytecode, creating `bin/classes.dex` as output
5. Run `aapt` again, in a different mode, to package the resources and assets
6. Run `apkbuilder` to bundle everything up into an APK file

You can see these steps in action by examining the Ant `build.xml` file for an Android project, as generated by `activitycreator` or other sources.

Integrating JARs into Android

Let's suppose you have a JAR containing Java code, whether that code is specifically for Android or not. As described in *BCG for Android*, all you need to do is drop that JAR in the `libs/` folder of your project (or import it into your Eclipse project, etc.), and it will magically be available to your code.

In terms of the build process, the `javac` step will add the JARs in `libs/` to the `classpath`, `dx` will convert the JARs' bytecodes and pour the results into the overall `classes.dex` file for the application, and `apkbuilder` will package it all up.

Conversely, the other steps in the build process largely ignore the existence of JARs in `libs/`:

- The `aapt` command does not inspect JARs for resources, let alone package any such resources
- The `aidl` command does not inspect JARs for any AIDL files that require code generation

Putting Limits on the JAR

How the Android build process works with third-party JARs suggests some limitations on how we construct a JAR specifically for use with Android:

- We cannot package any resources, such as layouts or images, inside the JAR, at least not while expecting the Android build process to do something with them automatically.
- We must generate our AIDL interfaces and stubs ourselves while building our JAR. This should not come as a surprise, as we will need those classes anyway as part of building our AIDL implementations and test code that uses them.

The limit on resources does not mean we cannot put resources in our JARs, but merely that we need to provide instructions to those using those JARs to

manually extract those resources and put them in their own projects' `res/` directory trees. Of course, you could then package those resources separately, rather than pile them all into a single JAR file.

Crafting an Android-Aware JAR

With all that in mind, here is how to create a Java project that builds a JAR that uses Android and serves as a reusable component, but does not represent an Android application:

1. Create a plain old ordinary Java project, not one created by `activitycreator` or the equivalent Android operation in Eclipse. In other words, initially, pretend that your project has nothing at all to do with Android.
2. If you are using AIDL, arrange for your project's build process to run the `aidl` command. For example, you might paste in the necessary portions of an Android project's `build.xml` file into your own project's `build.xml` file – just enough to run AIDL and generate your interfaces and stubs. If you are not using AIDL, you can safely skip this step.
3. Arrange to have `android.jar` from your Android SDK installation be part of the `bootclasspath` during your `javac` compile step. As one might expect, `android.jar` represents the public API of Android. Including it in the compile step is what allows your code to reference Android classes, constants, and whatnot. The reason for putting it in `bootclasspath` is unclear, but that is how Android application projects do it, so it is probably safest if you follow their lead.

And that is pretty much it. Everything else should be normal for building a JAR for any Java project (e.g., running the `jar` command to package the classes into the JAR itself). You do not need to worry about any of the other steps in the typical Android build process.

[[TBD: example]]

An API with Intent

Sending Data in the Intent

Callbacks As Intents

Serving Your Fellow Bits

(mostly covered in to-be-written remote services chapter)

Pros, Cons, and Other Forms of Navel-Gazing

So, which approach should you take? Just a JAR? A `BrodcastReceiver`? A service? Or maybe some hybrid of these approaches? Which of these will a "reuser" (developer reusing your component) find best?

Well, that depends.

There are any number of criteria upon which you can judge those three core techniques. Below, we examine a few such criteria, in hopes of illustrating

the benefits and the detriments of each approach, so you can apply the same sort of analysis for the criteria that are important to you.

Richness of API

One criterion is the richness of the API. In other words, how "natural" is it for somebody to reuse your reusable component? Does it feel like it is simply part of the Android API or other Java development? Or does the architecture of the potential component system leave reusers feeling constrained?

For fine-grained interactions, the JAR is tough to beat. You can publish whole class libraries this way, without being limited to certain data types or having to jump through hoops for each method you want to expose. Your component is just another set of Java classes a reuser can code against.

The service model does let you expose a Java API, but connecting to the service via AIDL is asynchronous, meaning the API might not yet be ready for use when you want to use it. The `Intent` "extras" API actually allows a somewhat richer set of data to be passed along with the request, but it is somewhat more awkward if you need to get responses back.

You might also consider some form of hybrid, putting your own rich Java API wrapper around the service AIDL or `Intent`-based IPC scheme. This gives your reusers the best of both words

Code Duplication

With space at a premium on some devices, minimizing code duplication may be worth considering. An ordinary JAR, used by several applications, must be bundled with each of those applications – there is no shared `classpath` for common JARs. As a result, one JAR can wind up consuming several times its "natural" size in actual footprint, if several copies are baked into several applications.

Conversely, a service – whether accessed via AIDL or by a set of `Intents` – can support several applications while only being installed once.

Ease of Initial Deployment

Unfortunately, Android's packaging mechanism runs a bit counter to the benefits of a single service described in the previous section.

Applications are installed on a per-APK basis. There is no "package manager" in the sense you see in Linux, or a `.msi` file like you might see on Windows, that let's you bundle up several components to be installed at once.

A remote service intended for use among several applications must be packaged and deployed as its own application. End users have to know that they need to not only install the main application but also install any support services that are not already installed. This can cause a fair amount of confusion, because end users are used to installing and running applications, not installing applications and ignoring them (since they are not meant for direct use). Also, end users are used to installing applications and having an associated icon appear in their launcher, yet there may not be a point for a remote service to offer any sort of UI, let alone appear in the launcher.

Until this issue is rectified in one form or fashion, it will generally be simpler to deploy a JAR baked into the application reusing it, whether that JAR exposes a class library or a local service.

Intended Form of Integration

Most of the time, reusable components are meant to be specifically reused by other developers, who code to an API, whether that API is expressed as a Java class or an IPC method or an Intent to be raised.

However, Android does offer an introspection engine, allowing one activity to find other activities that can perform useful operations upon a piece of content. For example, you might create a PDF file viewer, since none are built into Android; Android email clients might then be able to use an `Intent` to trigger your activity to view a PDF attachment. To make this work, though, you need to implement an `Intent` receiver, either as an activity or as a `BroadcastReceiver`, so you can provide your functionality to other applications this way. The benefit is that you can add value to existing applications without those applications specifically integrating your code.

CHAPTER 14

Testing Your Code

Testing Your Instrument

Something Incompletely Different

CHAPTER 15

Production Applications

Making Your Mark

(code signing)

To Market, To Market

Wide Distribution

Click Here To Download

Let Your Fingers Do the Distributing

Late-Breaking Updates

Keyword Index

Class.....

AccelerateDecelerateInterpolator.....53

AccelerateInterpolator.....53

Activity.....82

Adapter.....16-18, 21, 22, 105

AdapterView.OnItemSelectedListener.....30

AlertDialog.....133

AlphaAnimation.....45, 50-52, 55

Animation.....45, 46, 52-54

AnimationListener.....52

AnimationSet.....45, 54, 55

AnimationUtils.....52

BaseColumns.....108

BitmapDrawable.....78

BroadcastReceiver.....148

BrodcastReceiver.....145

BshService.....127, 135

BshServiceDemo.....129

Button.....25, 39, 40, 43

CallLog.....107, 109, 110, 117

CallLog.Calls.....107, 108

CallPlusAdapter.....115, 117

Camera.....71-76, 135

Camera.Parameters.....73, 74

Camera.PictureCallback.....75

Camera.ShutterCallback.....75

ConstantsInstaller.....100

Contacts.....101, 102, 107

Contacts.ContactMethodsColumns.....107

Contacts.PeopleColumns.....106, 107

Contacts.Phones.....106

Contacts.PhonesColumns.....106

ContactsDemo.....102, 105

ContentProvider.....109

ContentValues.....114

Context.....52, 82, 88

Cursor.....17, 108-110, 114, 117

CursorAdapter.....115

CursorJoiner.....109

CursorWrapper.....109, 110

CycleInterpolator....52, 54
DatabaseInstaller....98-101
DecelerateInterpolator....53
Drawable....26, 35, 36
DrawerLayout....48, 50, 52, 54
Exception....135
ExecuteScriptJob....135
GeoWebOne....2
GradientDemo....36
HeaderFooterDemo....24
I_JoinHandler....114
ImageButton....47, 48, 53, 55
InputStream....100
Intent....xii, 130, 141, 146-148
Interpolator....53, 54
IScript....127, 130, 132
IScriptResult....132, 133, 137
Job....76
JoinCache....114
JoinCursor....109, 110, 114, 115
JoinDemo....108, 114, 115, 117
KillJob....77
LinearInterpolator....53
LinearLayout....24, 47, 48
LinkedBlockingQueue....76, 135
LinkedHashMap....114
List....21
ListActivity....18, 22, 28, 102, 117
ListView. 15-18, 22, 24, 26-28, 30, 36, 38, 102, 103, 117
Locater....4
LocationListener....4, 8
LocationManager....4
Media/Audio....59
MediaPlayer....58, 59, 62, 63
Menu....52
NinePatchDemo....43
NoteActivity....119
NoteEditor....119
PictureDemo....75, 76, 78
PreviewDemo....70, 71
RemoteService....126
RotateAnimation....45, 52
Runnable....77
SavePhotoJob....77
ScaleAnimation....45
ScrollView....86
Section....21
SectionAdapter....21
SectionedAdapter....18, 21, 22
SectionedDemo....18, 22
SeekBar....43
SelectorAdapter....30
SelectorDemo....28, 30
SelectorWrapper....30
SensorListener....83, 85
SensorManager....82, 88
Shaker....87-89
Shaker.Callback....89
ShakerDemo....87, 89
SimpleCursorAdapter....17, 18, 105-107

Spinner....102, 103, 105
SQLiteDatabase....99, 100
SQLiteOpenHelper....95, 98, 99
SurfaceHolder....71, 72
SurfaceHolder.Callback....72, 73
SurfaceView....71, 72
TextSwitcher....46
TextView....26, 27, 30, 83, 86, 87
Thread....77
Toast....130, 133
TranslateAnimation....45-48, 50, 51, 53, 55
TranslationAnimation....52
Uri....57, 58
VideoDemo....64
VideoView....63, 64
View....16-18, 21, 22, 24, 25, 27, 30, 46, 48, 52
ViewAnimator....46
ViewFlipper....46
ViewWrapper....115
WebSettings....1
WebView....1, 2, 4, 7, 9, 10
WebViewClient....1

Command

aapt....142, 143
activitycreator....98, 142, 144
adb push....65
aidl....142-144
apkbuilder....142, 143
draw9patch....41, 44
dx....142, 143
jar....144
javac....142-144
mksdcard....65
ruby....97
sqlite3....95, 97, 100

Method

addFooterView()....24
addHeaderView()....24
addJavascriptInterface()....2, 4, 6
addSection()....21
areAllItemsSelectable()....16
bindService()....130
buildFooter()....26
buildHeader()....25
create()....62
delete()....93
eval()....131, 135
execSQL()....100, 101
executeScript()....131, 132, 134
failure()....133
failureImpl()....133
getColumnCount()....114
getColumnIndex()....114
getCount()....21
getHolder()....72
getInt()....114
getItem()....21
getItemViewType()....22

getJoin()....114, 115
getSystemService()....82
getView()....18, 21
getViewTypeCount()....21, 22
handleInstallError()....99
insert()....93
isEnabled()....16
isNull()....114
loadAnimation()....52
loadUrl()....7, 9
onAccuracyChanged()....83
onAnimationEnd()....52
onCreate()....62, 72, 99, 100
onDestroy()....77, 83
onItemSelected()....30
onKeyDown()....75
onLocationChanged()....8
onNothingSelected()....30
onPictureTaken()....75
onSensorChanged()....83
onUpgrade()....99
open()....71
pause()....59, 62
play()....62
prepare()....58, 59, 62
prepareAsync()....58, 59
query()....93
registerListener()....83
release()....63, 73
requery()....114, 115
runOnUiThread()....133
setAnimationListener()....52
setDataSource()....58
setDuration()....50
setInterpolator()....54
setOnItemSelectedListener()....28
setPictureFormat()....74
setPreviewDisplay()....72
setType()....72
setup()....62, 63
setVisibility()....47, 52
shakingStarted()....89
shakingStopped()....89
start()....59
startAnimation()....46, 50
startPreview()....73
steerLeft()....86
steerRight()....86
stop()....59, 62, 63
stopPreview()....73
success()....133
successImpl()....133
surfaceChanged()....73
surfaceCreated()....72
surfaceDestroyed()....73
takePicture()....75
toString()....131
unregisterListener()....83
update()....93