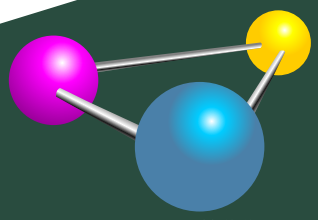
Version 1.4

Updated for Android 2.1!

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

Printed in the United States of America.

CommonsWare books may be purchased in printed (bulk) or digital form for educational or business use. For more information, contact *direct@commonsware.com*.

Printing History:
Mar 2010: Version 1.4 ISBN: 978-0-9816780-1-6

Table of Contents

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.

You can search through the PDF using most PDF readers (e.g., Adobe Reader). If you wish to search all of the CommonsWare books at once, and your operating system does not support that directly, you can always combine the PDFs into one, using tools like PDF Split-And-Merge or the Linux command `pdftk *.pdf cat output combined.pdf`.

Some notes for first-generation 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*, sometimes via the shorthand *BCG to Android* moniker.

In order to make effective use of this book, you will want to download the source code for it off of the book's page on the CommonsWare site.

You can find out when new releases of this book are available via:

- The cw-android Google Group, which is also a great place to ask questions about the book and its examples
- The commonsguy Twitter feed
- The Warescription newsletter, which you can subscribe to off of your Warescription page

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

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.

If you wish to use the source code from the CommonsWare Web site, bear in mind a few things:

1. The projects are set up to be built by Ant, not by Eclipse. If you wish to use the code with Eclipse, you will need to create a suitable Android Eclipse project and import the code and other assets.
2. You should delete build.xml, then run `android update project -p ...` (where ... is the path to a project of interest) on those projects you wish to use, so the build files are updated for your Android SDK version.

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 March 1, **2014**. 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 UI

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:

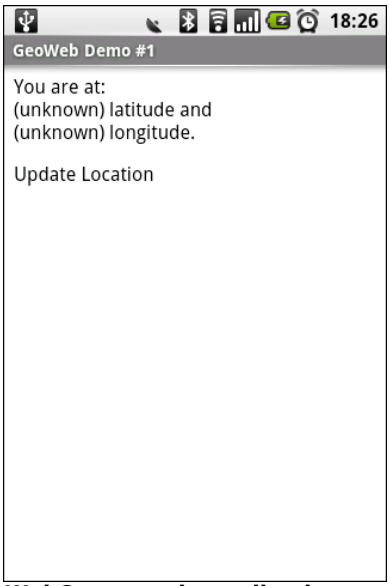


Figure 1. The GeoWebOne sample application, as initially launched

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:

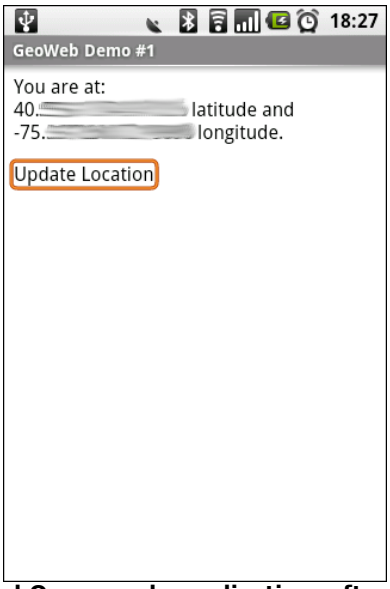


Figure 2. The GeoWebOne sample application, after clicking the Update Location link

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, 0,
                                             0,
                                             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:

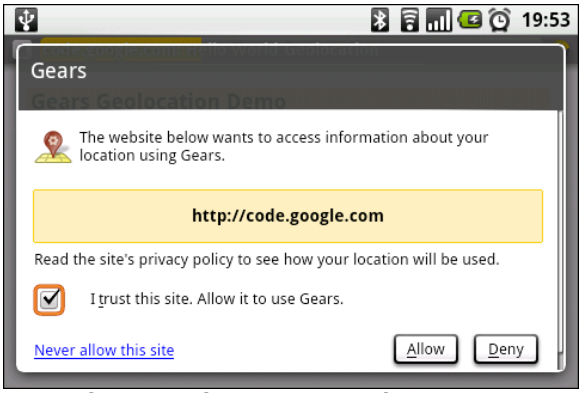


Figure 3. The Gears security prompt

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

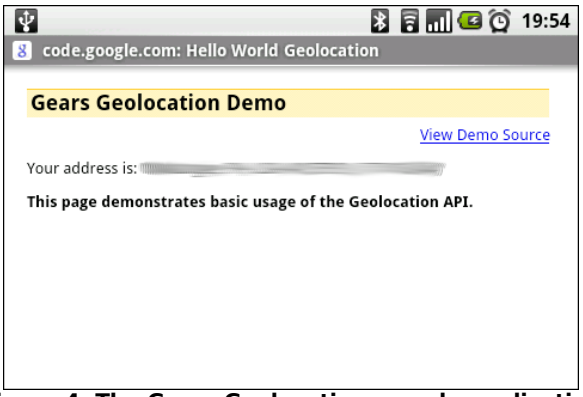


Figure 4. The Gears Geolocation sample application

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 lets you craft your own widgets as well, such as extending an existing widget with a new UI or new behaviors.

Getting Meta

One common way of creating your own widgets is to aggregate other widgets together into a reusable "meta" widget. Rather than worry about all the details of measuring the widget sizes and drawing its contents, you simply worry about creating the public interface for how one interacts with the widget.

In this section, we will look at the `Views/Meter` sample project. Here, we bundle a `ProgressBar` and two `ImageButton` widgets into a reusable `Meter` widget, allowing one to change a value by clicking "increment" and "decrement" buttons. In most cases, one would probably be better served using the built-in `SeekBar` widget. However, there are times when we only want people to change the value a certain amount at a time, for which the `Meter` is ideally suited. In fact, we will reuse the `Meter` in a later chapter when we show how to manipulate the various volume levels in Android.

The Widget Layout

The first step towards creating a reusable widget is to lay it out. In some cases, you may prefer to create your widget contents purely in Java, particularly if many copies of the widget will be created and you do not want to inflate them every time. However, otherwise, layout XML is simpler in many cases than the in-Java alternative.

Here is one such `Meter` layout (`res/layout/meter.xml` in `Views/Meter`):

```
<?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="wrap_content"
>
  <ImageButton android:id="@+id/decr"
    android:layout_height="30px"
    android:layout_width="30px"
    android:src="@drawable/decr"
  />
  <ProgressBar android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="0px"
    android:layout_weight="1"
    android:layout_height="wrap_content"
  />
  <ImageButton android:id="@+id/incr"
    android:layout_height="30px"
    android:layout_width="30px"
    android:src="@drawable/incr"
  />
</LinearLayout>
```

All we do is line them up in a row, giving the `ProgressBar` any excess space (via `android:layout_width = "0px"` and `android:layout_weight = "1"`). We are using a pair of 16x16 pixel images from the Nuvola icon set for the increment and decrement button faces.

The Attribute Declarations

Widgets usually have attributes that you can set in the XML file, such as the `android:src` attribute we specified on the `ImageButton` widgets in the layout

above. You can create your own custom attributes that can be used in your custom widget, by creating a `res/values/attrs.xml` file to specify them.

For example, here is the attributes file for `Meter`:

```
<resources>
  <declare-styleable name="Meter">
    <attr name="max" format="integer" />
    <attr name="incr" format="integer" />
    <attr name="decr" format="integer" />
  </declare-styleable>
</resources>
```

The `declare-styleable` element describes what attributes are available on the widget class specified in the name attribute – in our case, we will call the widget `Meter`. Inside `declare-styleable` you can have one or more `attr` elements, each indicating the name of an attribute (e.g., `incr`) and what data type the attribute has (e.g., `integer`). The data type will help with compile-time validation and in getting any supplied values for this attribute parsed into the appropriate type at runtime.

Here, we indicate there are three attributes: `max` (indicating the highest value the `Meter` will go to), `incr` (indicating how much the value should increase when the increment button is clicked), and `decr` (indicating how much the value should decrease when the decrement button is clicked).

The Widget Implementation

There are many ways to go about actually implementing a widget. This section outlines one option: using a container class (specifically `LinearLayout`) and inflating the contents of your widget layout into the container.

The Constructor

To be usable inside of layout XML, you need to implement a constructor that takes two parameters:

1. A `Context` object, typically representing the `Activity` that is inflating your widget
2. An `AttributeSet`, representing the bundle of attributes included in the element in the layout being inflated that references your widget

In this constructor, after chaining to your superclass, you can do some basic configuration of your widget. Bear in mind, though, that you are not in position to configure the widgets that make up your aggregate widget – you need to wait until `onFinishInflate()` before you can do anything with those.

One thing you definitely want to do in the constructor, though, is use that `AttributeSet` to get the values of whatever attributes you defined in your `attrs.xml` file. For example, here is the constructor for `Meter`:

```
public Meter(final Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  this.setOrientation(HORIZONTAL);

  TypedArray a=ctxt.obtainStyledAttributes(attrs,
                                           R.styleable.Meter,
                                           0, 0);

  max=a.getInt(R.styleable.Meter_max, 100);
  incrAmount=a.getInt(R.styleable.Meter_incr, 1);
  decrAmount=-1*a.getInt(R.styleable.Meter_decr, 1);

  a.recycle();
}
```

The `obtainStyledAttributes()` on `Context` allows us to convert the `AttributeSet` into useful values:

- It resolves references to other resources, such as strings
- It handles any styles that might be declared via the style attribute in the layout element that references your widget
- It finds the resources you declared via attrs.xml and makes them available to you

In the code shown above, we get our `TypedArray` via `obtainStyledAttributes()`, then call `getInt()` three times to get our values

out of the `TypedArray`. The `TypedArray` is keyed by `R.styleable` identifiers, so we use the three generated for us by the build tools for `max`, `incr`, and `decr`.

Note that you should call `recycle()` on the `TypedArray` when done – this makes this `TypedArray` available for immediate reuse, rather than forcing it to be garbage-collected.

Finishing Inflation

Your widget will also typically override `onFinishInflate()`. At this point, you can turn around and add your own contents, via Java code or, as shown below, by inflating a layout XML resource into yourself as a container:

```
@Override
protected void onFinishInflate() {
  super.onFinishInflate();

  ((Activity)getContext()).getLayoutInflater().inflate(R.layout.meter, this);

  bar=(ProgressBar)findViewById(R.id.bar);
  bar.setMax(max);

  ImageButton btn=(ImageButton)findViewById(R.id.incr);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      bar.incrementProgressBy(incrAmount);

      if (onIncr!=null) {
        onIncr.onClick(Meter.this);
      }
    }
  });

  btn=(ImageButton)findViewById(R.id.decr);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      bar.incrementProgressBy(decrAmount);

      if (onDecr!=null) {
        onDecr.onClick(Meter.this);
      }
    }
  });
}
```

Of course, once you have constructed or inflated your contents, you can configure them, particularly using the attributes you declared in attrs.xml and retrieved in your constructor.

Event Handlers and Other Methods

If you wish to expose events to the outside world – such as `Meter` exposing when the increment or decrement buttons are clicked – you need to do a few things:

- Choose or create an appropriate listener class or classes (e.g., `View.OnClickListener`)
- Hold onto instances of those classes as data members of the widget class
- Offer setters (and, optionally, getters) to define those listener objects
- Call those listeners when appropriate

For example, `Meter` holds onto a pair of `View.OnClickListener` instances:

```
private View.OnClickListener onIncr=null;
private View.OnClickListener onDecr=null;
```

It lets users of `Meter` define those listeners via getters:

```
public void setOnIncrListener(View.OnClickListener onIncr) {
  this.onIncr=onIncr;
}

public void setOnDecrListener(View.OnClickListener onDecr) {
  this.onDecr=onDecr;
}
```

And, as shown in the previous section, it passes along the button clicks to the listeners:

```
ImageButton btn=(ImageButton)findViewById(R.id.incr);

btn.setOnClickListener(new View.OnClickListener() {
```

```
  public void onClick(View v) {
    bar.incrementProgressBy(incrAmount);

    if (onIncr!=null) {
      onIncr.onClick(Meter.this);
    }
  }
});

btn=(ImageButton)findViewById(R.id.decr);

btn.setOnClickListener(new View.OnClickListener() {
  public void onClick(View v) {
    bar.incrementProgressBy(decrAmount);

    if (onDecr!=null) {
      onDecr.onClick(Meter.this);
    }
  }
});
```

Note that we change the value passed in the `onClick()` method – our listener receives the `ImageButton`, but we pass the `Meter` widget on the outbound `onClick()` call. This is so we do not leak internal implementation of our widget. The users of `Meter` should neither know nor care that we have `ImageButton` widgets as part of the `Meter` internals.

Your widget may well require other methods as well, for widget-specific configuration or functionality, though `Meter` does not.

Using the Widget

Given all of that, using the `Meter` widget is not significantly different than using any other widget provided in the system...with a few minor exceptions.

In the layout, since your custom widget is not in the `android.widget` Java package, you need to fully-qualify the class name for the widget, as seen in the `main.xml` layout for the `Views/Meter` project:

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

```
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:paddingTop="5px"
>
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Meter:"
  />
  <com.commonsware.android.widget.Meter
    android:id="@+id/meter"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    app:max="100"
    app:incr="1"
    app:decr="5"
  />
</LinearLayout>
```

You will also note that we have a new namespace (`xmlns:app = "http://schemas.android.com/apk/res/com.commonsware.android.widget"`), and that our custom attributes from above are in that namespace (e.g., `app:max`). The custom namespace is because our attributes are not official Android ones and so will not be recognized by the build tools in the `android:` namespace, so we have to create our own. The value of the namespace needs to be `http://schemas.android.com/apk/res/` plus the name of the package containing the styleable attributes (`com.commonsware.android.widget`).

With just the stock generated activity, we get the following UI:

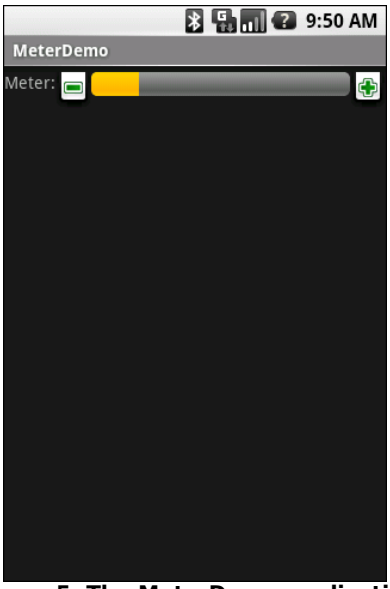


Figure 5. The MeterDemo application

Note that there is a significant shortcut we are taking here: our `Meter` implementation and its consumer (`MeterDemo`) are in the same Java package. We will expose this shortcut in a later chapter when we use the `Meter` widget in another project.

Change of State

Sometimes, we do not need to change the functionality of an existing widget, but we simply want to change how it looks. Maybe you want an oddly-shaped `Button`, or a `CheckBox` that is much larger, or something. In these cases, you may be able to tailor instances of the existing widget as you see fit, rather than have to roll a separate widget yourself.

Changing Button Backgrounds

Suppose you want a `Button` that looks like the second button shown below:

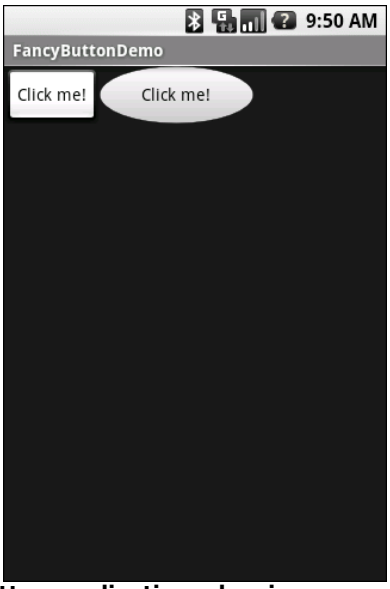


Figure 6. The FancyButton application, showing a normal oval-shaped button

Moreover, it needs to not just sit there, but also be focusable:

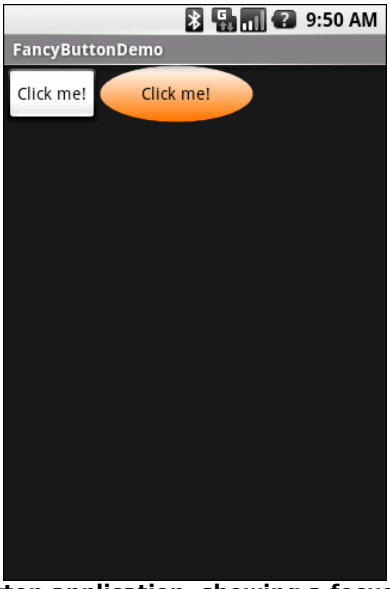


Figure 7. The FancyButton application, showing a focused oval-shaped button

...and it needs to be clickable:

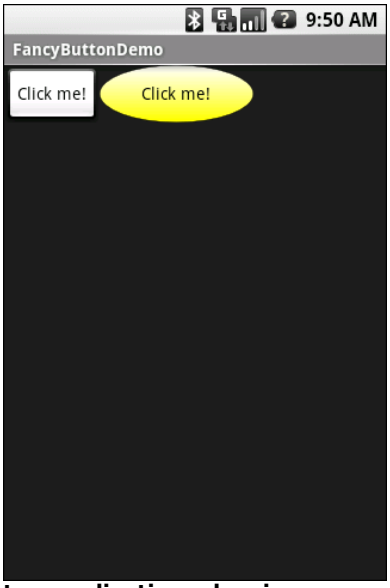


Figure 8. The FancyButton application, showing a pressed oval-shaped button

If you did not want the look of the `Button` to change, you could get by just with a simple `android:background` attribute on the `Button`, providing an oval PNG. However, if you want the `Button` to change looks based on state, you need to create another flavor of custom `Drawable` – the selector.

A selector `Drawable` is an XML file, akin to shapes with gradients. However, rather than specifying a shape, it specifies a set of other `Drawable` resources and the circumstances under which they should be applied, as described via a series of states for the widget using the `Drawable`.

For example, from `Views/FancyButton`, here is `res/drawable/fancybutton.xml`, implementing a selector `Drawable`:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:state_focused="true"
    android:state_pressed="false"
    android:drawable="@drawable/btn_oval_selected"
```

```
  />
  <item
    android:state_focused="true"
    android:state_pressed="true"
    android:drawable="@drawable/btn_oval_pressed"
  />
  <item
    android:state_focused="false"
    android:state_pressed="true"
    android:drawable="@drawable/btn_oval_pressed"
  />
  <item
    android:drawable="@drawable/btn_oval_normal"
  />
</selector>
```

There are four states being described in this selector:

1. Where the button is focused (`android:state_focused = "true"`) but not pressed (`android:state_pressed = "false"`)
2. Where the button is both focused and pressed
3. Where the button is not focused but is pressed
4. The default, where the button is neither focused nor pressed

In these four states, we specify three `Drawable` resources, for normal, focused, and pressed (the latter being used regardless of focus).

If we specify this selector `Drawable` resource as the `android:background` of a `Button`, Android will use the appropriate PNG based on the status of the `Button` itself:

```
<?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="wrap_content"
>
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me!"
  />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```
    android:text="Click me!"
    android:background="@drawable/fancybutton"
  />
</LinearLayout>
```

Changing CheckBox States

The same basic concept can be used to change the images used by a `CheckBox`.

In this case, the fact that Android is open source helps, as we can extract files and resources from Android and adjust them to create our own editions, without worrying about license hassles.

For example, here is a selector `Drawable` for a fancy `CheckBox`, showing a dizzying array of possible states:

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

    <!-- Enabled states -->

    <item android:state_checked="true" android:state_window_focused="false"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_on" />
    <item android:state_checked="false" android:state_window_focused="false"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_off" />

    <item android:state_checked="true" android:state_pressed="true"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_on_pressed" />
    <item android:state_checked="false" android:state_pressed="true"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_off_pressed" />

    <item android:state_checked="true" android:state_focused="true"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_on_selected" />
    <item android:state_checked="false" android:state_focused="true"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_off_selected" />

    <item android:state_checked="false"
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_off" />
    <item android:state_checked="true"
```

```
          android:state_enabled="true"
          android:drawable="@drawable/btn_check_on" />


    <!-- Disabled states -->

    <item android:state_checked="true" android:state_window_focused="false"
          android:drawable="@drawable/btn_check_on_disable" />
    <item android:state_checked="false" android:state_window_focused="false"
          android:drawable="@drawable/btn_check_off_disable" />

    <item android:state_checked="true" android:state_focused="true"
          android:drawable="@drawable/btn_check_on_disable_focused" />
    <item android:state_checked="false" android:state_focused="true"
          android:drawable="@drawable/btn_check_off_disable_focused" />

    <item android:state_checked="false"
android:drawable="@drawable/btn_check_off_disable" />
    <item android:state_checked="true"
android:drawable="@drawable/btn_check_on_disable" />

</selector>
```

Each of the referenced PNG images can be extracted from the `android.jar` file in your Android SDK, or obtained from various online resources. In the case of `Views/FancyCheck`, we zoomed each of the images to 200% of original size, to make a set of large (albeit fuzzy) checkbox images.

Figure 9. An example of a zoomed CheckBox image

In our layout, we can specify that we want to use our `res/drawable/fancycheck.xml` selector `Drawable` as our background:

```
<?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="wrap_content"
>
  <CheckBox
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="I'm normal!"
  />
  <CheckBox
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="        "
    android:button="@drawable/fancycheck"
    android:background="@drawable/btn_check_label_background"
  />
</LinearLayout>
```

This gives us a look like this:

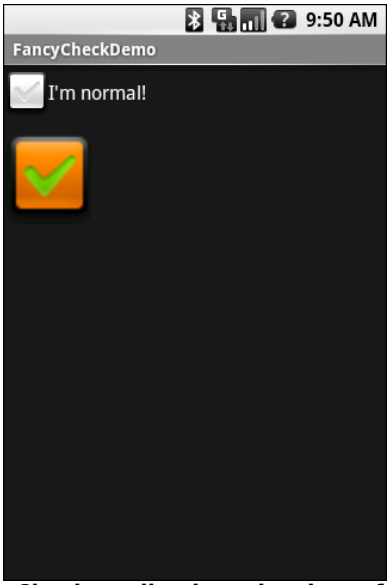


Figure 10. The FancyCheck application, showing a focused and checked CheckBox

Note that our `CheckBox` text is blank. The reason is that `CheckBox` is expecting the graphics to be 38px wide. Since ours are substantially larger, the `CheckBox` images overlap the text. Fixing this would require substantial work. It is simplest to fill the `CheckBox` text with some whitespace, then use a separate `TextView` for our `CheckBox` caption.

CHAPTER 3

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.

First, we look at using shape XML files to create gradient effects that can be resized to accommodate different contents. We then examine StateListDrawable and how it can be used for button backgrounds, tab icons, map icons, and the like. We wrap by looking at nine-patch bitmaps, for places where a shape file will not work but that the image still needs to be resized, such as a `Button` background.

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="#44FFFF00"
    android:endColor="#FFFFFF00"
    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 yellow, 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 Drawable/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,
                                                parent, false);
        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:

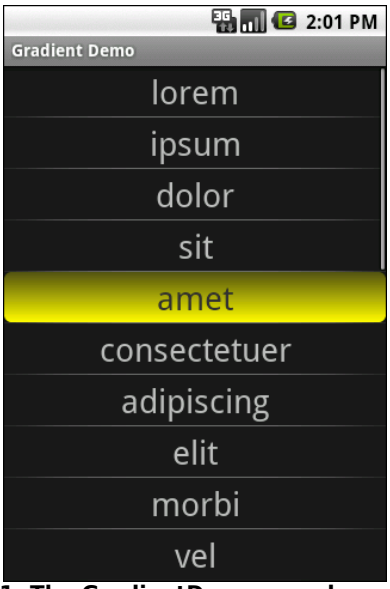


Figure 11. The GradientDemo sample application

Note that because the list background is black, the yellow 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 yellow mixed with white, as determined by the alpha channel specified on the gradient's top color.

State Law

Gradients and other shapes are not the only types of `Drawable` resource you can define using XML. One, the `StateListDrawable`, is key if you want to have different images when widgets are in different states.

Take for example the humble `Button`. Somewhere along the line, you have probably tried setting the background of the `Button` to a different color, perhaps via the `android:background` attribute in layout XML. If you have not tried this before, give it a shot now.

When you replace the `Button` background with a color, the `Button` becomes...well...flat. There is no defined border. There is no visual response

when you click the `Button`. There is no orange highlight if you select the `Button` with the D-pad or trackball.

This is because what makes a `Button` visually be a `Button` is its background. Your new background is a flat color, which will be used no matter what is going on with the `Button` itself. The original background, however, was a `StateListDrawable`, one that looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

 http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
-->

<selector xmlns:android="http://schemas.android.com/apk/res/android">
 <item android:state_window_focused="false" android:state_enabled="true"
 android:drawable="@drawable/btn_default_normal" />
 <item android:state_window_focused="false" android:state_enabled="false"
 android:drawable="@drawable/btn_default_normal_disable" />
 <item android:state_pressed="true"
 android:drawable="@drawable/btn_default_pressed" />
 <item android:state_focused="true" android:state_enabled="true"
 android:drawable="@drawable/btn_default_selected" />
 <item android:state_enabled="true"
 android:drawable="@drawable/btn_default_normal" />
 <item android:state_focused="true"
 android:drawable="@drawable/btn_default_normal_disable_focused" />
 <item
 android:drawable="@drawable/btn_default_normal_disable" />
</selector>
```

The XML has a `<selector>` root element, indicating this is a `StateListDrawable`. The `<item>` elements inside the root describe what Drawable resource should be used if the `StateListDrawable` is being used in some state. For example, if the "window" (think activity or dialog) does not have the focus (`android:state_window_focused="false"`) and the `Button` is enabled (`android:state_enabled="true"`), then we use the

`@drawable/btn_default_normal` `Drawable` resource. That resource, as it turns out, is a nine-patch PNG file, described later in this chapter.

Android applies each rule in turn, top-down, to find the `Drawable` to use for a given state of the `StateListDrawable`. The last rule has no `android:state_*` attributes, meaning it is the overall default image to use if none of the other rules match.

So, if you want to change the background of a `Button`, you need to:

1. Copy the above resource, found in your Android SDK as `res/drawable/btn_default.xml`, into your project
2. Copy each of the `Button` state nine-patch images into your project
3. Modify whichever of those nine-patch images you want, to affect the visual change you seek
4. If need be, tweak the states and images defined in the `StateListDrawable` XML you copied
5. Reference the local `StateListDrawable` as the background for your `Button`

You can also use this technique for tab icons – the currently-selected tab will use the image defined as `android:state_selected="true"`, while the other tabs will use images with `android:state_selected="false"`.

We will see `StateListDrawable` used later in this book, in the chapter on maps, showing you how you can have different icons in an overlay for normal and selected states of an overlay item.

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.

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.

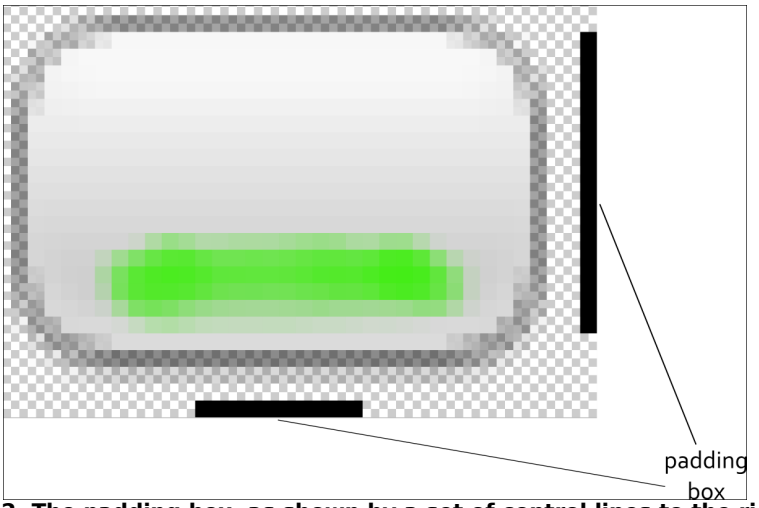


Figure 12. The padding box, as shown by a set of control lines to the right and bottom of the stretchable image

Stretch 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 stretch 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:

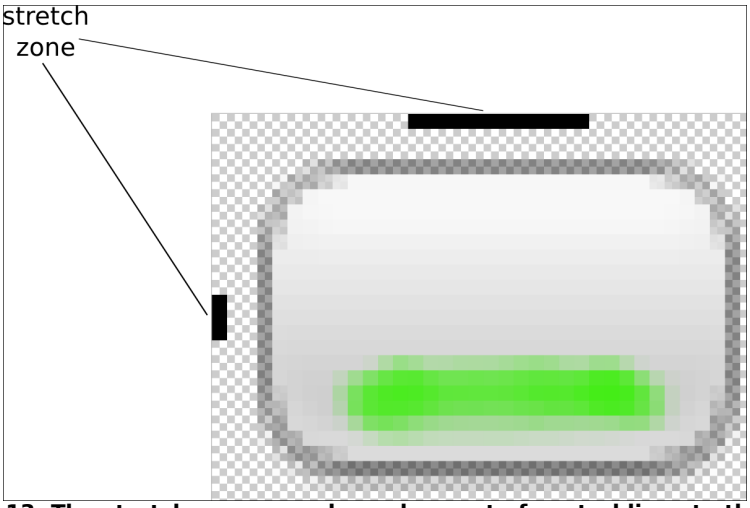


Figure 13. The stretch zones, as shown by a set of control lines to the right and bottom of the stretchable image

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

Some additional rules to bear in mind:

- If you have multiple discrete stretch 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 stretch 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:

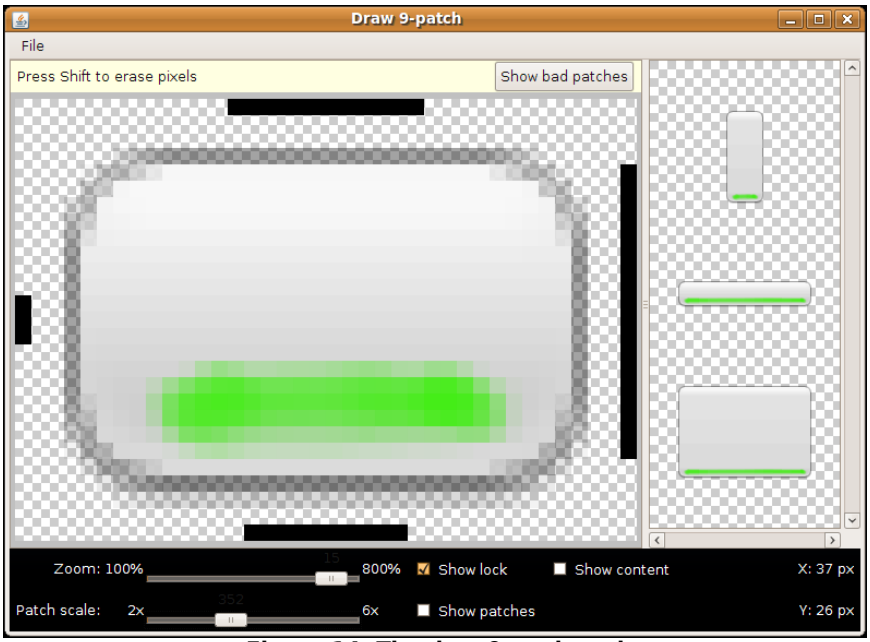


Figure 14. The draw9patch tool

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.

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:

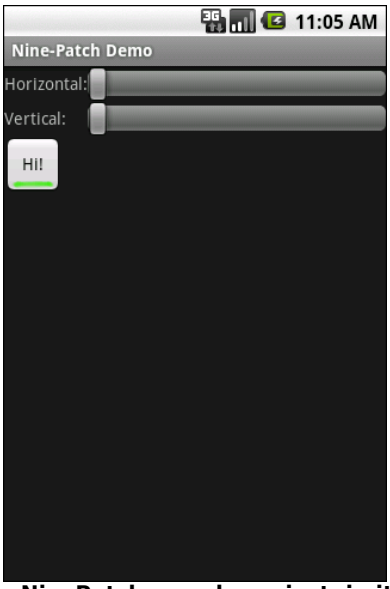


Figure 15. The NinePatch sample project, in its initial state

Figure 16. The NinePatch sample project, after making it bigger horizontally

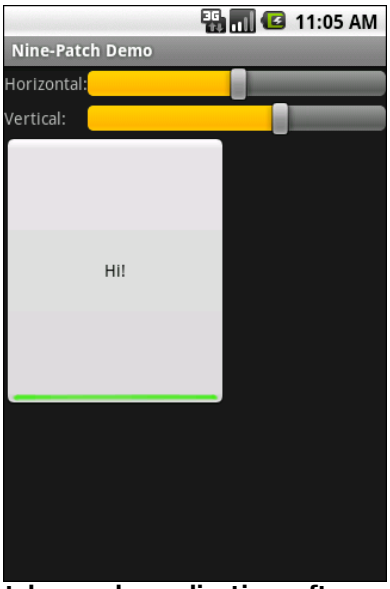


Figure 17. The NinePatch sample application, after making it bigger in both dimensions

CHAPTER 4

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 `ListView`s", 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:

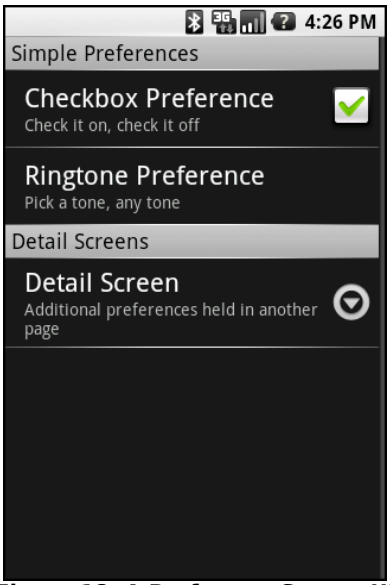


Figure 18. A PreferenceScreen UI

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.

Introducing MergeAdapter

CommonsWare – the publishers of this book – have released a number of open source reusable Android libraries, collectively called the CommonsWare Android Components (CWAC, pronounced "quack"). Several of these will come into play for adding headings to a list, primarily `MergeAdapter`.

MergeAdapter takes a collection of ListAdapter objects and other View widgets and consolidates them into a single master ListAdapter that can be poured into a ListView. You supply the contents – MergeAdapter handles the ListAdapter interface to make them all appear to be a single contiguous list.

In the case of ListView with section headings, we can use MergeAdapter to alternate between headings (each a View) and the rows inside each heading (e.g., a CursorAdapter wrapping content culled from a database).

We will see how MergeAdapter works in greater detail in an upcoming edition of this book. Here, we will see how you can apply a MergeAdapter to achieve the desired ListView look and feel.

Lists via Merges

The pattern to use MergeAdapter for sectioned lists 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.).
- Create heading Views for each heading (e.g., a custom-styled TextView)
- Create a MergeAdapter and sequentially add each heading and content Adapter in turn
- 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"
/>
```

Our activity's `onCreate()` method wraps our list of nonsense words in an `ArrayAdapter` three times, first with the original list and twice on randomly shuffled editions of the list. It pops each of those into the `MergeAdapter` after a related heading, inflated from a custom layout:

```
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);
    }
  };
}
```

he result is much as you might expect:

Figure 19. A ListView using a MergeAdapter, showing one header and part of a list

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.Handler;
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;
import java.util.concurrent.atomic.AtomicBoolean;

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"};
  private long startTime=SystemClock.uptimeMillis();
  private Handler handler=new Handler();
  private AtomicBoolean areWeDeadYet=new AtomicBoolean(false);

  @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));
  }

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

    areWeDeadYet.set(true);
  }

  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");

    if (!areWeDeadYet.get()) {
      handler.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:

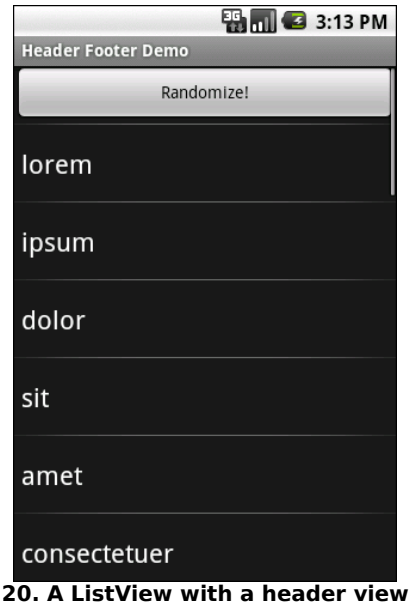


Figure 20. A ListView with a header view shown

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

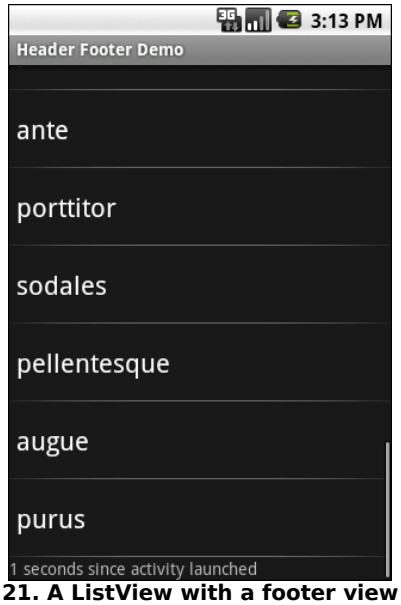


Figure 21. A ListView with a footer view shown

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="#FFFFFF00"
    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 yellow (`android:background = "#FFFFFF00"`) 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 yellow 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,
                                                parent, false);
        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 yellow bar:

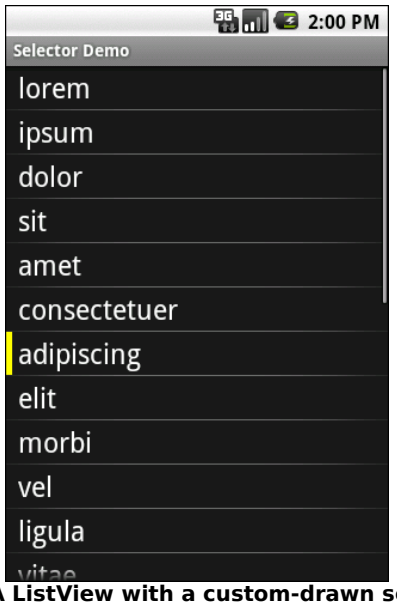


Figure 22. A ListView with a custom-drawn selector icon

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.

Stating Your Selection

In the previous section, we removed the default `ListView` selection bar and implemented our own in Java code. That works, but there is another option: defining a custom selection bar `Drawable` resource.

In the chapter on custom `Drawable` resources, we introduced the `StateListDrawable`. This is an XML-defined resource that declares different `Drawable` resources to use when the `StateListDrawable` is in different states.

The standard `ListView` selector is, itself, a `StateListDrawable`, one that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project
```

```
 Licensed under the Apache License, Version 2.0 (the "License");
 you may not use this file except in compliance with the License.
 You may obtain a copy of the License at

 http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
-->

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

 <item android:state_window_focused="false"
 android:drawable="@color/transparent" />

 <!-- Even though these two point to the same resource, have two states so the
drawable will invalidate itself when coming out of pressed state. -->
 <item android:state_focused="true" android:state_enabled="false"
 android:state_pressed="true"
 android:drawable="@drawable/list_selector_background_disabled" />
 <item android:state_focused="true" android:state_enabled="false"
 android:drawable="@drawable/list_selector_background_disabled" />

 <item android:state_focused="true" android:state_pressed="true"
 android:drawable="@drawable/list_selector_background_transition" />
 <item android:state_focused="false" android:state_pressed="true"
 android:drawable="@drawable/list_selector_background_transition" />

 <item android:state_focused="true"
 android:drawable="@drawable/list_selector_background_focus" />

</selector>
```

Now, the most common reason people seem to want to change the selector is that they hate the orange bar. Perhaps it clashes with their application's color scheme, or they are allergic to citrus fruits, or something.

The `android:state_focused="true"` rule at the bottom of that XML is the one that defines the actual selection bar, in terms of what is seen when the user navigates with the D-pad or trackball. It points to a nine-patch PNG file, with different copies for different screen densities (one in `res/drawable-hdpi/`, etc.).

Hence, another approach to changing the selection bar is to:

1. Copy the above XML (found in `res/drawable/list_selector_background.xml` in your SDK) into your project
2. Copy the various other `Drawable` resources pointed to by that XML into your project
3. Modify the nine-patch images as needed to change the colors
4. Reference the local copy of the `StateListDrawable` in the `android:listSelector` attribute

CHAPTER 5

Home Screen App Widgets

One of the oft-requested features added in Android 1.5 is the ability to add live elements to the home screen. Called "app widgets", these can be added by users via a long-tap on the home screen and choosing an appropriate widget from the available roster. Android ships with a few app widgets, such as a music player, but developers can add their own – in this chapter, we will see how this is done.

For the purposes of this book, "app widgets" will refer to these items that go on the home screen. Other uses of the term "widget" will be reserved for the UI widgets, subclasses of `View`, usually found in the `android.widget` Java package.

In this chapter, we briefly touch on the security ramifications of app widgets, before continuing on to discuss how Android offers a secure app widget framework. We then go through all the steps of creating a basic app widget. Next, we discuss how to deal with multiple instances of your app widget, the app widget lifecycle, alternative models for updating app widgets, and how to offer multiple layouts for your app widget (perhaps based on device characteristics). We wrap with some notes about hosting your own app widgets in your own home screen implementation.

East is East, and West is West...

Part of the reason it took as long as it did for app widgets to become available is security.

Android's security model is based heavily on Linux user, file, and process security. Each application is (normally) associated with a unique user ID. All of its files are owned by that user, and its process(es) run as that user. This prevents one application from modifying the files of another or otherwise injecting their own code into another running process.

In particular, the core Android team wanted to find a way that would allow app widgets to be displayed by the home screen application, yet have their content come from another application. It would be dangerous for the home screen to run arbitrary code itself or somehow allow its UI to be directly manipulated by another process.

The app widget architecture, therefore, is set up to keep the home screen application independent from any code that puts app widgets on that home screen, so bugs in one cannot harm the other.

The Big Picture for a Small App Widget

The way Android pulls off this bit of security is through the use of `RemoteViews`.

The application component that supplies the UI for an app widget is not an `Activity`, but rather a `BroadcastReceiver` (often in tandem with a `Service`). The `BroadcastReceiver`, in turn, does not inflate a normal `View` hierarchy, like an `Activity` would, but instead inflates a layout into a `RemoteViews` object.

`RemoteViews` encapsulates a limited edition of normal widgets, in such a fashion that the `RemoteViews` can be "easily" transported across process boundaries. You configure the `RemoteViews` via your `BroadcastReceiver` and make those `RemoteViews` available to Android. Android in turn delivers the

RemoteViews to the app widget host (usually the home screen), which renders them to the screen itself.

This architectural choice has many impacts:

1. You do not have access to the full range of widgets and containers. You can use FrameLayout, LinearLayout, and RelativeLayout for containers, and AnalogClock, Button, Chronometer, ImageButton, ImageView, ProgressBar, and TextView for widgets.
2. The only user input you can get is clicks of the Button and ImageButton widgets. In particular, there is no EditText for text input.
3. Because the app widgets are rendered in another process, you cannot simply register an OnClickListener to get button clicks; rather, you tell RemoteViews a PendingIntent to invoke when a given button is clicked.
4. You do not hold onto the RemoteViews and reuse them yourself. Rather, the pattern appears to be that you create and send out a brand-new RemoteViews whenever you want to change the contents of the app widget. This, coupled with having to transport the RemoteViews across process boundaries, means that updating the app widget is rather expensive in terms of CPU time, memory, and battery life.
5. Because the component handling the updates is a BroadcastReceiver, you have to be quick (lest you take too long and Android consider you to have timed out), you cannot use background threads, and your component itself is lost once the request has been completed. Hence, if your update might take a while, you will probably want to have the BroadcastReceiver start a Service and have the Service do the long-running task and eventual app widget update.

Crafting App Widgets

This will become somewhat easier to understand in the context of some sample code. In the AppWidget/TwitterWidget project, you will find an app

widget that shows the latest tweet in your Twitter timeline. If you have read *Android Programming Tutorials*, you will recognize the JTwitter JAR we will use for accessing the Twitter Web service.

The Manifest

First, we need to register our `BroadcastReceiver` (and, if relevant, `Service`) implementation in our `AndroidManifest.xml` file, along with a few extra features:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.appwidget"
      android:versionCode="1"
      android:versionName="1.0">
 <uses-sdk
      android:minSdkVersion="6"
      android:targetSdkVersion="6"
  />
  <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="@string/app_name"
        android:icon="@drawable/cw">
        <activity android:name=".TWPrefs"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action
android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
            </intent-filter>
      </activity>
        <receiver android:name=".TwitterWidget"
            android:label="@string/app_name"
            android:icon="@drawable/tw_icon">
            <intent-filter>
                <action
                    android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data
                android:name="android.appwidget.provider"
                android:resource="@xml/widget_provider" />
        </receiver>
        <service android:name=".TwitterWidget$UpdateService" />
    </application>
</manifest>
```

Here we have an `<activity>`, a `<receiver>`, and a `<service>`. Of note:

- Our `<receiver>` has `android:label` and `android:icon` attributes, which are not normally needed on `BroadcastReceiver` declarations. However, in this case, those are used for the entry that goes in the menu of available widgets to add to the home screen. Hence, you will probably want to supply values for both of those, and use appropriate resources in case you want translations for other languages.
- Our `<receiver>` has an `<intent-filter>` for the `android.appwidget.action.APPWIDGET_UPDATE` action. This means we will get control whenever Android wants us to update the content of our app widget. There may be other actions we want to monitor – more on this in a later section.
- Our `<receiver>` also has a `<meta-data>` element, indicating that its `android.appwidget.provider` details can be found in the `res/xml/widget_provider.xml` file. This metadata is described in the next section.
- Our `<activity>` has two `<intent-filter>` elements, the normal "put me in the Launcher" one and one looking for an action of `android.appwidget.action.APPWIDGET_CONFIGURE`.

The Metadata

Next, we need to define the app widget provider metadata. This has to reside at the location indicated in the manifest – in this case, in `res/xml/widget_provider.xml`:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="292dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:configure="com.commonsware.android.appwidget.TWPrefs"
  android:initialLayout="@layout/widget"
/>
```

Here, we provide four pieces of information:

- The minimum width and height of the app widget (`android:minWidth` and `android:minHeight`). These are approximate – the app widget host (e.g., home screen) will tend to convert these values into "cells" based upon the overall layout of the UI where the app widgets will reside. However, they should be no smaller than the minimums cited here.
- The frequency in which Android should request an update of the widget's contents (`android:updatePeriodMillis`). This is expressed in terms of milliseconds, so a value of `3600000` is a 60-minute update cycle. Note that the minimum value for this attribute is 30 minutes – values less than that will be ignored.
- An activity class that will be used to configure the widget when it is first added to the screen (`android:configure`). This will be described in greater detail in a later section.

The configuration activity is optional. However, if you skip the configuration activity, you do need to tell Android the initial layout to use for the app widget, via an `android:initialLayout` attribute.

The Layout

Eventually, you are going to need a layout that describes what the app widget looks like. So long as you stick to the widget and container classes noted above, this layout can otherwise look like any other layout in your project.

For example, here is the layout for the `TwitterWidget`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#FF000088"
    >
  <ImageButton android:id="@+id/refresh"
    android:layout_alignParentTop="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/refresh"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <ImageButton android:id="@+id/configure"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/configure"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <TextView android:id="@+id/friend"
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"
    android:layout_toLeftOf="@id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="left"
    android:textStyle="bold"
    android:singleLine="true"
    android:ellipsize="end"
  />
  <TextView android:id="@+id/status"
    android:layout_below="@id/friend"
    android:layout_alignParentLeft="true"
    android:layout_toLeftOf="@id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:gravity="top"
    android:singleLine="false"
    android:lines="4"
  />
</RelativeLayout>
```

All we have is a `TextView` to show the latest tweet, plus another one for the person issuing the tweet, and a pair of `ImageButton` widgets to allow the user to manually refresh the latest tweet and launch the configuration activity.

The BroadcastReceiver

Next, we need a `BroadcastReceiver` that can get control when Android wants us to update our `RemoteViews` for our app widget. To simplify this, Android supplies an `AppWidgetProvider` class we can extend, instead of the normal `BroadcastReceiver`. This simply looks at the received `Intent` and calls out to an appropriate lifecycle method based on the requested action.

The one method that invariably needs to be implemented on the provider is `onUpdate()`. Other lifecycle methods may be of interest and are discussed later in this chapter.

For example, here is the `onUpdate()` implementation of the `AppWidgetProvider` for `TwitterWidget`:

```
@Override
public void onUpdate(Context ctxt,
                     AppWidgetManager mgr,
                     int[] appWidgetIds) {
  ctxt.startService(new Intent(ctxt, UpdateService.class));
}
```

If our `RemoteViews` could be rapidly constructed, we could do the work right here. However, in our case, we need to make a Web service call to Twitter, which might take a while, so we instead call `startService()` on the `Service` we declared in our manifest, to have it make the updates.

The Service

The real work for `TwitterWidget` is mostly done in an `UpdateService` inner class of `TwitterWidget`.

`UpdateService` does not extend `Service`, but rather extends `IntentService`. `IntentService` is designed for patterns like this one, where our service is started multiple times, with each "start" representing a distinct piece of work to be accomplished (in this case, updating an app widget from Twitter). `IntentService` allows us to implement `onHandleIntent()` to do this work, and it arranges for `onHandleIntent()` to be called on a background thread. Hence, we do not need to deal with starting or stopping our thread, or even stopping our service when there is no more work to be done – Android handles that automatically.

Here is the `onHandleIntent()` implementation from `UpdateService`:

```
@Override
public void onHandleIntent(Intent intent) {
  ComponentName me=new ComponentName(this,
```

```
                                  TwitterWidget.class);
  AppWidgetManager mgr=AppWidgetManager.getInstance(this);

  mgr.updateAppWidget(me, buildUpdate(this));
}
```

To update the `RemoteViews` for our app widget, we need to build those `RemoteViews` (delegated to a `buildUpdate()` helper method) and tell an `AppWidgetManager` to update the widget via `updateAppWidget()`. In this case, we use a version of `updateAppWidget()` that takes a `ComponentName` as the identifier of the widget to be updated. Note that this means that we will update all instances of this app widget presently in use – the concept of multiple app widget instances is covered in greater detail later in this chapter.

Working with `RemoteViews` is a bit like trying to tie your shoes while wearing mittens – it may be possible, but it is a bit clumsy. In this case, rather than using methods like `findViewById()` and then calling methods on individual widgets, we need to call methods on `RemoteViews` itself, providing the identifier of the widget we wish to modify. This is so our requests for changes can be serialized for transport to the home screen process. It does, however, mean that our view-updating code looks a fair bit different than it would if this were the main `View` of an activity or row of a `ListView`.

For example, here is the `buildUpdate()` method from `UpdateService`, which builds a `RemoteViews` containing the latest Twitter information, using account information pulled from shared preferences:

```
private RemoteViews buildUpdate(Context context) {
  RemoteViews updateViews=new RemoteViews(context.getPackageName(),
                                          R.layout.widget);
  String user=prefs.getString("user", null);
  String password=prefs.getString("password", null);

  if (user!=null && password!=null) {
    Twitter client=new Twitter(user, password);
    List<Twitter.Status> timeline=client.getFriendsTimeline();

    if (timeline.size()>0) {
      Twitter.Status s=timeline.get(0);

      updateViews.setTextViewText(R.id.friend,
                                  s.user.screenName);
```

```
      updateViews.setTextViewText(R.id.status,
                                  s.text);

      Intent i=new Intent(this, TwitterWidget.class);
      PendingIntent pi=PendingIntent.getBroadcast(context,
                                                  0  , i,
                                                  0);

      updateViews.setOnClickPendingIntent(R.id.refresh,
                                          pi);

      i=new Intent(this, TWPrefs.class);
      pi=PendingIntent.getActivity(context, 0  , i, 0);
      updateViews.setOnClickPendingIntent(R.id.configure,
                                          pi);
    }
  }

  return(updateViews);
}
```

To create the `RemoteViews`, we use a constructor that takes our package name and the identifier of our layout. This gives us a `RemoteViews` that contains all of the widgets we declared in that layout, just as if we inflated the layout using a `LayoutInflater`. The difference, of course, is that we have a `RemoteViews` object, not a `View`, as the result.

We then use methods like:

- `setTextViewText()` to set the text on a `TextView` in the `RemoteViews`, given the identifier of the `TextView` within the layout we wish to manipulate
- `setOnClickPendingIntent()` to provide a `PendingIntent` that should get fired off when a `Button` or `ImageButton` is clicked

Note, of course, that Android does not know anything about Twitter – the Twitter object comes from a JTwitter JAR located in the `libs/` directory of our project.

The Configuration Activity

Way back in the manifest, we included an `<activity>` element for a `TWPrefs` activity. And, in our widget metadata XML file, we said that `TWPrefs` was the

android:configure attribute value. In our RemoteViews for the widget itself, we connect a configure button to launch TWPrefs when clicked.

The net of all of this is that TWPrefs is the configuration activity. Specifically:

- It will be launched when we request to add this widget to our home screen
- It will be re-launched whenever we click the configure button in the widget itself

For the latter scenario, the activity need be nothing special. In fact, TWPrefs is mostly just a PreferenceActivity, updating the SharedPreferences for this application with the user's Twitter screen name and password, used for logging into Twitter and fetching the latest timeline entry.

The former scenario – defining a configuration activity in the metadata – requires a bit more work, though.

If we were to leave this out, and not have an android:configure attribute in the metadata, once the user chose to add our widget to their home screen, the widget would immediately appear. Behind the scenes, Android would ask our AppWidgetProvider to supply the RemoteViews for the widget body right away.

However, when we declare that we want a configuration activity, we must build the initial RemoteViews ourselves and return them as the activity's result. Behind the scenes, Android uses startActivityForResult() to launch our configuration activity, then looks at the result and uses the associated RemoteViews to create the initial look of the widget.

This approach is prone to code duplication, and it is not completely clear why Android elected to build the widget framework this way.

That being said, here is the implementation of TWPrefs:

```
package com.commonsware.android.appwidget;

import android.app.Activity;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.preference.PreferenceActivity;
import android.view.KeyEvent;
import android.widget.RemoteViews;

public class TWPrefs extends PreferenceActivity {
  private static String
CONFIGURE_ACTION="android.appwidget.action.APPWIDGET_CONFIGURE";

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

    addPreferencesFromResource(R.xml.preferences);
  }

  @Override
  public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode==KeyEvent.KEYCODE_BACK &&
        Integer.parseInt(Build.VERSION.SDK)<5) {
      onBackPressed();
    }

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

  @Override
  public void onBackPressed() {
    if (CONFIGURE_ACTION.equals(getIntent().getAction())) {
      Intent intent=getIntent();
      Bundle extras=intent.getExtras();

      if (extras!=null) {
        int id=extras.getInt(AppWidgetManager.EXTRA_APPWIDGET_ID,
                             AppWidgetManager.INVALID_APPWIDGET_ID);
        AppWidgetManager mgr=AppWidgetManager.getInstance(this);
        RemoteViews views=new RemoteViews(getPackageName(),
                                          R.layout.widget);

        mgr.updateAppWidget(id, views);

        Intent result=new Intent();

        result.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                        id);
        setResult(RESULT_OK, result);
        sendBroadcast(new Intent(this,
```

```
                                TwitterWidget.class));
      }
    }

    super.onBackPressed();
  }
}
```

We are using the same activity for two cases: for the initial configuration and for later on-demand reconfiguration via the configure button in the widget. We need to tell these apart. More importantly, we need to get control at an appropriate time to set our activity result in the initial configuration case. Alas, the normal activity lifecycle methods (e.g., `onDestroy()`) are too late, and `PreferenceActivity` offers no other explicit hook to find out when the user dismisses the preference screen.

So, we have to cheat a bit.

Specifically, we hook `onBackPressed()` and watch for the back button. When the back button is pressed, if we were launched by a widget configuration `Intent` (`CONFIGURE_ACTION.equals(getIntent().getAction())`), then we go through and:

- Get our widget instance identifier (described in greater detail later in this chapter)
- Get our `AppWidgetManager` and create a new `RemoteViews` inflated from our widget layout
- Pass the empty `RemoteViews` to the `AppWidgetManager` via `updateAppWidget()`
- Call `setResult()` with an `Intent` wrapping our widget instance identifier, so Android knows we have properly configured our widget
- Raise a broadcast `Intent` to ask our `WidgetProvider` to do the *real* initial version of the widget

This minimizes code duplication, but it does mean there is a slight hiccup, where the widget initially appears blank, before the first timeline entry appears. This is largely unavoidable in this case – we cannot wait for Twitter

to respond since `onBackPressed()` is called on the UI thread and we need to call `setResult()` now rather than wait for Twitter's response.

Undoubtedly, there are other patterns for handling this situation.

Note that `onBackPressed()` is new to Android 2.0. For earlier versions of Android, you will instead want to override `onKeyDown()` and look for `KeyEvent.KEYCODE_BACK` events.

The Result

If you compile and install all of this, you will have a new widget entry available when you long-tap on the home screen background:

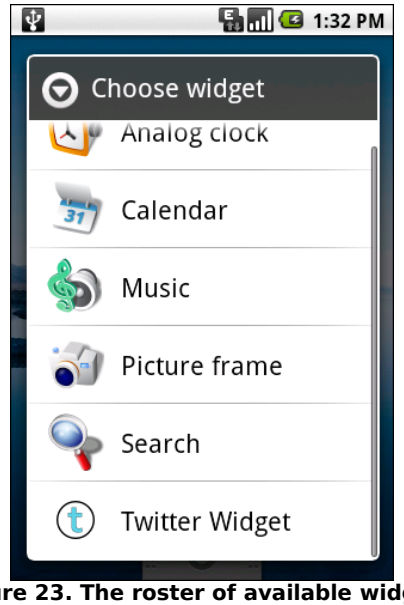


Figure 23. The roster of available widgets

When you choose Twitter Widget, you will initially be presented with the configuration activity:

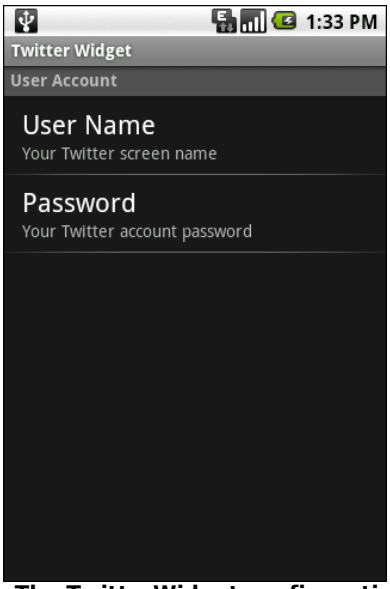


Figure 24. The TwitterWidget configuration activity

Once you set your Twitter screen name and password, and press the back button to exit the activity, your widget will appear with no contents:

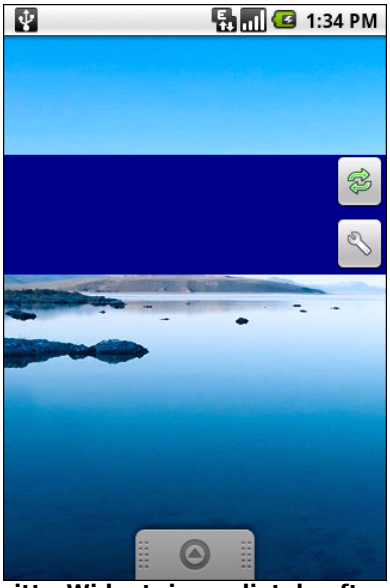


Figure 25. TwitterWidget, immediately after being added

After a moment, though, it will appear with the latest in your Twitter friends timeline:

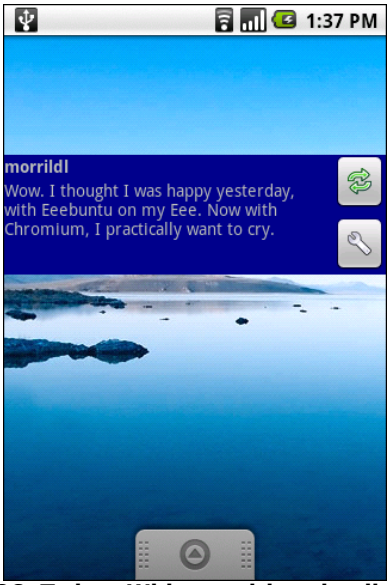


Figure 26. TwitterWidget, with a timeline entry

To change your Twitter credentials, you can either tap the configure icon in the widget or run the Twitter Widget application in your launcher. And, clicking the refresh button, or waiting 15 minutes, will cause the widget to update its contents.

Another and Another

As indicated above, you can have multiple instances of the same app widget outstanding at any one time. For example, one might have multiple picture frames, or multiple "show-me-the-latest-RSS-entry" app widgets, one per feed. You will distinguish between these in your code via the identifier supplied in the relevant `AppWidgetProvider` callbacks (e.g., `onUpdate()`).

If you want to support separate app widget instances, you will need to store your state on a per-app-widget-identifier basis. For example, while `TwitterWidget` uses preferences for the Twitter account details, you might

need multiple preference files, or use a SQLite database with an app widget identifier column, or something to distinguish one app widget instance from another. You will also need to use an appropriate version of `updateAppWidget()` on `AppWidgetManager` when you update the app widgets, one that takes app widget identifiers as the first parameter, so you update the proper app widget instances.

Conversely, there is nothing requiring you to support multiple instances as independent entities. For example, if you add more than one `TwitterWidget` to your home screen, nothing blows up – they just show the same tweet. In the case of `TwitterWidget`, they might not even show the same tweet all the time, since they will update on independent cycles, so one will get newer tweets before another.

App Widgets: Their Life and Times

TwitterWidget overrode two `AppWidgetProvider` methods:

- `onUpdate()`, invoked when the `android:updatePeriodMillis` time has elapsed
- `onReceive()`, the standard `BroadcastReceiver` callback, used to detect when we are invoked with no action, meaning we want to force an update due to the refresh button being clicked

There are three other lifecycle methods that `AppWidgetProvider` offers that you may be interested in:

- `onEnabled()` will be called when the first widget instance is created for this particular widget provider, so if there is anything you need to do once for all supported widgets, you can implement that logic here
- `onDeleted()` will be called when a widget instance is removed from the home screen, in case there is any data you need to clean up specific to that instance

- `onDisabled()` will be called when the last widget instance for this provider is removed from the home screen, so you can clean up anything related to all such widgets

Note, however, that there is a bug in Android 1.5, where `onDeleted()` will not be properly called. You will need to implement `onReceive()` and watch for the `ACTION_APPWIDGET_DELETED` action in the received `Intent` and call `onDeleted()` yourself. This should be fixed in a future edition of Android.

Controlling Your (App Widget's) Destiny

As `TwitterWidget` illustrates, you are not limited to updating your app widget only based on the timetable specified in your metadata. That timetable is useful if you can get by with a fixed schedule. However, there are cases in which that will not work very well:

- If you want the user to be able to configure the polling period (the metadata is baked into your APK and therefore cannot be modified at runtime)
- If you want the app widget to be updated based on external factors, such as a change in location

The recipe shown in `TwitterWidget` will let you use `AlarmManager` (described in a later chapter) or proximity alerts or whatever to trigger updates. All you need to do is:

- Arrange for something to broadcast an `Intent` that will be picked up by the `BroadcastReceiver` you are using for your app widget provider
- Have the provider process that `Intent` directly or pass it along to a `Service` (such as an `IntentService` as shown in `TwitterWidget`)

Also, note that the `updateTimeMillis` setting not only tells the app widget to update every so often, it will even *wake up the phone* if it is asleep so the widget can perform its update. On the plus side, this means you can easily keep your widgets up to date regardless of the state of the device. On the minus side, this will tend to drain the battery, particularly if the period is too fast. If you want to avoid this wakeup behavior, set `updateTimeMillis` to

`0` and use `AlarmManager` to control the timing and behavior of your widget updates.

Change Your Look

If you have been doing most of your development via the Android emulator, you are used to all "devices" having a common look and feel, in terms of the home screen, lock screen, and so forth. This is the so-called "Google Experience" look, and many actual Android devices have it.

However, some devices have their own presentation layers. HTC has "Sense", seen on the HTC Hero and HTC Tattoo, among other devices. Motorola has MOTOBLUR, seen on the Motorola CLIQ and DEXT. Other device manufacturers are sure to follow suit. These presentation layers replace the home screen and lock screen, among other things. Moreover, they usually come with their own suite of app widgets with their own look and feel. Your app widget may look fine on a Google Experience home screen, but the look might clash when viewed on a Sense or MOTOBLUR device.

Fortunately, there are ways around this. You can set your app widget's look on the fly at runtime, to choose the layout that will look the best on that particular device.

The first step is to create an app widget layout that is initially invisible (`res/layout/invisible.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:visibility="invisible"
    >
</RelativeLayout>
```

This layout is then the one you would reference from your app widget metadata, to be used when the app widget is first created:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="292dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:configure="com.commonsware.android.appwidget.TWPrefs"
  android:initialLayout="@layout/invisible"
/>
```

This ensures that when your app widget is initially added, you do not get the "Problem loading widget" placeholder, yet you also do not choose one layout versus another – it is simply invisible for a brief moment.

Then, in your `AppWidgetProvider` (or attached `IntentService`), you can make the choice of what layout to inflate as part of your `RemoteViews`. Rather than using the invisible one, you can choose one based on the device or other characteristics.

For example, here is a revised version of our app widget layout that uses a different color background (`res/layout/widget_alt.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#FF008800"
    >
  <ImageButton android:id="@+id/refresh"
    android:layout_alignParentTop="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <ImageButton android:id="@+id/configure"
    android:layout_alignParentBottom="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/configure"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <TextView android:id="@+id/friend"
    android:layout_alignParentTop="true"
    android:layout_alignParentLeft="true"
    android:layout_toLeftOf="@id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="left"
```

```
    android:textStyle="bold"
    android:singleLine="true"
    android:ellipsize="end"
  />
  <TextView android:id="@+id/status"
    android:layout_below="@id/friend"
    android:layout_alignParentLeft="true"
    android:layout_toLeftOf="@id/refresh"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:gravity="top"
    android:singleLine="false"
    android:lines="4"
  />
</RelativeLayout>
```

We can modify our IntentService to choose which layout – `widget.xml` or `widget_alt.xml` – to use. For example, the following code shows how we could use a specific layout for Android 2.1 devices:

```
int layout=R.layout.widget;

if (android.os.Build.VERSION.SDK_INT==7) {
  layout=R.layout.widget_alt;
}

RemoteViews updateViews=new RemoteViews(context.getPackageName(),
                                        layout);
```

The biggest challenge is that there is no good way to determine what presentation layer, if any, is in use on a device. For the time being, you will need to use the various fields in the `android.os.Build` class to "sniff" on the device model and make a decision that way.

Being a Good Host

In addition to creating your own app widgets, it is possible to host app widgets. This is mostly aimed for those creating alternative home screen applications, so they can take advantage of the same app widget framework and all the app widgets being built for it.

This is not very well documented at this juncture, but it apparently involves the `AppWidgetHost` and `AppWidgetHostView` classes. The latter is a `View` and so

should be able to reside in an app widget host's UI like any other ordinary widget.

CHAPTER 6

Searching with SearchManager

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

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

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

Hunting Season

There are two types of search in Android: local and global. Local search searches within the current application; global search searches the Web via Google's search engine. You can initiate either type of search in a variety of ways, including:

- You can call `onSearchRequested()` from a button or menu choice, which will initiate a local search (unless you override this method in your activity)
- You can directly call `startSearch()` to initiate a local or global search, including optionally supplying a search string to use as a starting point

- You can elect to have keyboard entry kick off a search via `setDefaultKeyMode()`, for either local search (`setDefaultKeyMode(DEFAULT_KEYS_SEARCH_LOCAL)`) or global search (`setDefaultKeyMode(DEFAULT_KEYS_SEARCH_GLOBAL)`)

In either case, the search appears as a set of UI components across the top of the screen, with a suggestion list (where available) and IME (where needed).

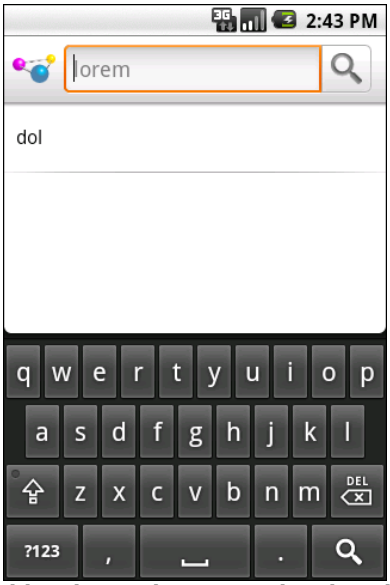


Figure 27. The Android local search popup, showing the IME and a previous search

Figure 28. The Android global search popup

Where that search suggestion comes from for your local searches will be covered later in this chapter.

Search Yourself

Over the long haul, there will be two flavors of search available via the Android search system:

1. Query-style search, where the user's search string is passed to an activity which is responsible for conducting the search and displaying the results
2. Filter-style search, where the user's search string is passed to an activity on every keypress, and the activity is responsible for updating a displayed list of matches

Since the latter approach is decidedly under-documented, let's focus on the first one.

Craft the Search Activity

The first thing you are going to want to do if you want to support query-style search in your application is to create a search activity. While it might be possible to have a single activity be both opened from the launcher and opened from a search, that might prove somewhat confusing to users. Certainly, for the purposes of learning the techniques, having a separate activity is cleaner.

The search activity can have any look you want. In fact, other than watching for queries, a search activity looks, walks, and talks like any other activity in your system.

All the search activity needs to do differently is check the intents supplied to `onCreate()` (via `getIntent()`) and `onNewIntent()` to see if one is a search, and, if so, to do the search and display the results.

For example, let's look at the `Search/Lorem` sample application. This starts off as a clone of the list-of-lorem-ipsum-words application originally encountered in *The Busy Coder's Guide to Android Development*. Now, we update it to support searching the list of words for ones containing the search string.

The main activity and the search activity both share a common layout: a `ListView` plus a `TextView` showing the selected entry:

```
<?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" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
```

```
  />
</LinearLayout>
```

In terms of Java code, most of the guts of the activities are poured into an abstract `LoremBase` class:

```
abstract public class LoremBase extends ListActivity {
  abstract ListAdapter makeMeAnAdapter(Intent intent);

  private static final int LOCAL_SEARCH_ID = Menu.FIRST+1;
  private static final int GLOBAL_SEARCH_ID = Menu.FIRST+2;
  private static final int CLOSE_ID = Menu.FIRST+3;
  TextView selection;
  ArrayList<String> items=new ArrayList<String>();

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

    try {
      XmlPullParser xpp=getResources().getXml(R.xml.words);

      while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
        if (xpp.getEventType()==XmlPullParser.START_TAG) {
          if (xpp.getName().equals("word")) {
            items.add(xpp.getAttributeValue(0));
          }
        }

        xpp.next();
      }
    }
    catch (Throwable t) {
      Toast
        .makeText(this, "Request failed: "+t.toString(), 4000)
        .show();
    }

    setDefaultKeyMode(DEFAULT_KEYS_SEARCH_LOCAL);

    onNewIntent(getIntent());
  }

  @Override
  public void onNewIntent(Intent intent) {
    ListAdapter adapter=makeMeAnAdapter(intent);

    if (adapter==null) {
      finish();
    }
```

```
    else {
      setListAdapter(adapter);
    }
  }

  public void onListItemClick(ListView parent, View v, int position,
                long id) {
    selection.setText(items.get(position).toString());
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(Menu.NONE, LOCAL_SEARCH_ID, Menu.NONE, "Local Search")
            .setIcon(android.R.drawable.ic_search_category_default);
    menu.add(Menu.NONE, GLOBAL_SEARCH_ID, Menu.NONE, "Global Search")
            .setIcon(R.drawable.search)
            .setAlphabeticShortcut(SearchManager.MENU_KEY);
    menu.add(Menu.NONE, CLOSE_ID, Menu.NONE, "Close")
            .setIcon(R.drawable.eject)
            .setAlphabeticShortcut('c');

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case LOCAL_SEARCH_ID:
        onSearchRequested();
        return(true);

      case GLOBAL_SEARCH_ID:
        startSearch(null, false, null, true);
        return(true);

      case CLOSE_ID:
        finish();
        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
}
```

This activity takes care of everything related to showing a list of words, even loading the words out of the XML resource. What it does not do is come up with the `ListAdapter` to put into the `ListView` – that is delegated to the subclasses.

The main activity – `LoremDemo` – just uses a `ListAdapter` for the whole word list:

```
package com.commonsware.android.search;

import android.content.Intent;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;

public class LoremDemo extends LoremBase {
  @Override
  ListAdapter makeMeAnAdapter(Intent intent) {
    return(new ArrayAdapter<String>(this,
                    android.R.layout.simple_list_item_1,
                    items));
  }
}
```

The search activity, though, does things a bit differently.

First, it inspects the `Intent` supplied to the abstract `makeMeAnAdapter()` method. That `Intent` comes from either `onCreate()` or `onNewIntent()`. If the intent is an `ACTION_SEARCH`, then we know this is a search. We can get the search query and, in the case of this silly demo, spin through the loaded list of words and find only those containing the search string. That list then gets wrapped in a `ListAdapter` and returned for display:

```
ListAdapter makeMeAnAdapter(Intent intent) {
  ListAdapter adapter=null;

  if (intent.getAction().equals(Intent.ACTION_SEARCH)) {
    String query=intent.getStringExtra(SearchManager.QUERY);
    List<String> results=searchItems(query);

    adapter=new ArrayAdapter<String>(this,
                    android.R.layout.simple_list_item_1,
                    results);
    setTitle("LoremSearch for: "+query);
  }

  return(adapter);
}
```

Update the Manifest

While this implements search, it doesn't tie it into the Android search system. That requires a few changes to the auto-generated `AndroidManifest.xml` file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.search">
 <uses-sdk
      android:minSdkVersion="3"
      android:targetSdkVersion="6"
  />
  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"
  />
  <application android:label="Lorem Ipsum"
    android:icon="@drawable/cw">
    <activity android:name=".LoremDemo" android:label="LoremDemo">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <meta-data android:name="android.app.default_searchable"
            android:value=".LoremSearch" />
    </activity>
    <activity
      android:name=".LoremSearch"
      android:label="LoremSearch"
      android:launchMode="singleTop">
      <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
      <meta-data android:name="android.app.searchable"
            android:resource="@xml/searchable" />
    </activity>
    <provider android:name=".LoremSuggestionProvider"
            android:authorities="com.commonsware.android.search.LoremSuggestionP
rovider" />
  </application>
</manifest>
```

The changes that are needed are:

1. The `LoremDemo` main activity gets a `meta-data` element, with an `android:name` of `android.app.default_searchable` and a `android:value` of the search implementation class (`.LoremSearch`)
2. The `LoremSearch` activity gets an intent filter for `android.intent.action.SEARCH`, so search intents will be picked up
3. The `LoremSearch` activity is set to have `android:launchMode = "singleTop"`, which means at most one instance of this activity will be open at any time, so we don't wind up with a whole bunch of little search activities cluttering up the activity stack

4. Add `android:label` and `android:icon` attributes to the `application` element – these will influence how your application appears in the Quick Search Box among other places
5. The `LoremSearch` activity gets a `meta-data` element, with an `android:name` of `android.app.searchable` and a `android:value` of an XML resource containing more information about the search facility offered by this activity (`@xml/searchable`)

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
  android:label="@string/searchLabel"
  android:hint="@string/searchHint"
  android:searchSuggestAuthority="com.commonsware.android.search.LoremSuggestion
Provider"
  android:searchSuggestSelection=" ? "
  android:searchSettingsDescription="@string/global"
  android:includeInGlobalSearch="true"
/>
```

That XML resource provides many bits of information, of which only two are needed for simple search-enabled applications:

1. What name should appear in the search domain button to the left of the search field, identifying to the user where she is searching (`android:label`)
2. What hint text should appear in the search field, to give the user a clue as to what they should be typing in (`android:hint`)

The other attributes found in that file, and the other search-related bits found in the manifest, will be covered later in this chapter.

Searching for Meaning In Randomness

Given all that, search is now available – Android knows your application is searchable, what search domain to use when searching from the main activity, and the activity knows how to do the search.

The options menu for this application has both local and global search options. In the case of local search, we just call `onSearchRequested()`; in the

case of global search, we call `startSearch()` with true in the last parameter, indicating the scope is global.

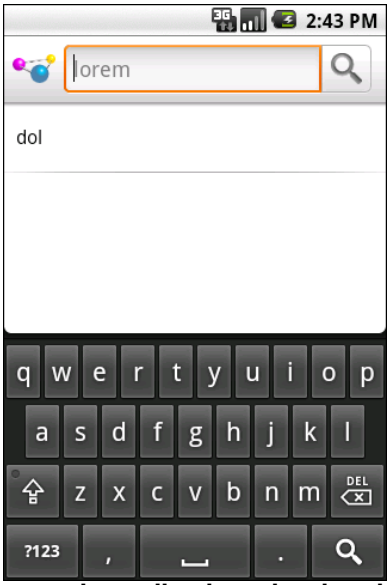


Figure 29. The Lorem sample application, showing the local search popup

Typing in a letter or two, then clicking Search, will bring up the search activity and the subset of words containing what you typed, with your search query in the activity title bar:

Figure 30. The results of searching for 'co' in the Lorem search sample

You can get the same effect if you just start typing in the main activity, since it is set up for triggering a local search.

May I Make a Suggestion?

When you do a global search, you are given "suggestions" of search words or phrases that may be what you are searching for, to save you some typing on a small keyboard:

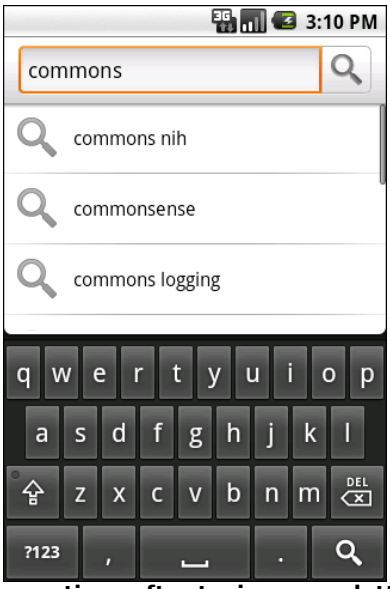


Figure 31. Search suggestions after typing some letters in global search

Your application, if it chooses, can offer similar suggestions. Not only will this give you the same sort of drop-down effect as you see with the global search above, but it also ties neatly into the Quick Search Box, as we will see later in this chapter.

To provide suggestions, you need to implement a `ContentProvider` and tie that provider into the search framework. You have two major choices for implementing a suggestion provider: use the built-in "recent" suggestion provider, or create your own from scratch.

SearchRecentSuggestionsProvider

The "recent" suggestions provider gives you a quick and easy way to remember past searches and offer those as suggestions on future searches.

To use this facility, you must first create a custom subclass of `SearchRecentSuggestionsProvider`. Your subclass may be very simple, perhaps just a two-line constructor with no other methods. However, since Android does not automatically record recent queries for you, you will also need to

give your search activity a way to record them such that the recent-suggestions provider can offer them as suggestions in the future.

Below, we have a `LoremSuggestionProvider`, extending `SearchRecentSuggestionsProvider`, that also supplies a "bridge" for the search activity to record searches:

```
package com.commonsware.android.search;

import android.content.Context;
import android.content.SearchRecentSuggestionsProvider;
import android.provider.SearchRecentSuggestions;

public class LoremSuggestionProvider
  extends SearchRecentSuggestionsProvider {
  private static String
AUTH="com.commonsware.android.search.LoremSuggestionProvider";

  static SearchRecentSuggestions getBridge(Context ctxt) {
    return(new SearchRecentSuggestions(ctxt, AUTH,
                                       DATABASE_MODE_QUERIES));
  }

  public LoremSuggestionProvider() {
      super();

      setupSuggestions(AUTH, DATABASE_MODE_QUERIES);
  }
}
```

The constructor, besides the obligatory chain to the superclass, simply calls `setupSuggestions()`. This takes two parameters:

- The authority under which you will register this provider in the manifest (see below)
- A flag indicating where the suggestions will come from – in this case, we supply the required `DATABASE_MODE_QUERIES` flag

Of course, since this is a `ContentProvider`, you will need to add it to your manifest:

```
android:label="LoremSearch"
android:launchMode="singleTop">
```

The other thing that `LoremSuggestionProvider` has is a static method that creates a properly-configured instance of a `SearchRecentSuggestions` object. This object knows how to save search queries to the database that the content provider uses, so they will be served up as future suggestions. It needs to know the same authority and flag that you provide to `setupSuggestions()`.

That `SearchRecentSuggestions` is then used by our `LoremSearch` class, inside its `searchItems()` method that actually examines the list of nonsense words for matches:

```
private List<String> searchItems(String query) {
  LoremSuggestionProvider
    .getBridge(this)
    .saveRecentQuery(query, null);

  List<String> results=new ArrayList<String>();

  for (String item : items) {
    if (item.indexOf(query)>-1) {
      results.add(item);
    }
  }

  return(results);
}
```

In this case, we always record the search, though you can imagine that some applications might not save searches that are invalid for one reason or another.

Custom Suggestion Providers

If you want to provide search suggestions based on something else – actual data, searches conducted by others that you aggregate via a Web service, etc. – you will need to implement your own `ContentProvider` that supplies that information. As with `SearchRecentSuggestionsProvider`, you will need to add your `ContentProvider` to the manifest so that Android knows it exists.

The details for doing this will be covered in a future edition of this book. For now, you are best served with the Android `SearchManager` documentation on the topic.

Integrating Suggestion Providers

Before your suggestions will appear, though, you need to tell Android to use your `ContentProvider` as the source of suggestions. There are two attributes on your `searchable` XML that make this connection:

- `android:searchSuggestAuthority` indicates the content authority for your suggestions – this is the same authority you used for your `ContentProvider`
- `android:searchSuggestSelection` is how the suggestion should be packaged as a query in the `ACTION_SEARCH Intent` – unless you have some reason to do otherwise, `" ? "` is probably a fine value to use

The result is that when we do our local search, we get the drop-down of past searches as suggestions:

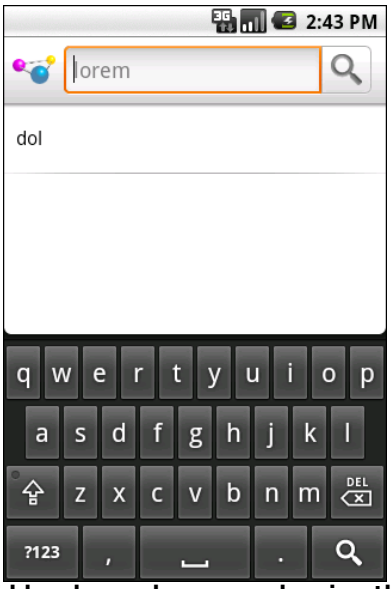


Figure 32. The Android local search popup, showing the IME and a previous search

There is also a `clearHistory()` method on `SearchRecentSuggestions` that you can use, perhaps from a menu choice, to clear out the search history, in case it is cluttered beyond usefulness.

Putting Yourself (Almost) On Par with Google

The Quick Search Box is Android's new term for the search widget at the top of the home screen. This is the same UI that appears when your application starts a global search. When you start typing, it shows possible matches culled from both the device and the Internet. If you choose one of the suggestions, it takes you to that item – choose a contact, and you visit the contact in the Contacts application. If you choose a Web search term, or you just submit whatever you typed in, Android will fire up a Browser instance showing you search results from Google. The order of suggestions is adaptive, as Android will attempt to show the user the sorts of things the user typically searches for (e.g., if the user clicks on contacts a lot in prior searches, it may prioritize suggested contacts in the suggestion list).

Your application can be tied into the Quick Search Box. However, it is important to understand that being in the Quick Search Box does *not* mean that your content will be searched. Instead, your *suggestions provider* will be queried based on what the user has typed in, and those suggestions will be blended into the overall results.

And, your application will not show up in Quick Search Box suggestions automatically – the user has to "opt in" to have your results included.

And, until the user demonstrates an interest in your results, your application's suggestions will be buried at the bottom of the list.

This means that integrating with the Quick Search Box, while still perhaps valuable, is not exactly what some developers will necessarily have in mind. That being said, here is how to achieve this integration.

Implement a Suggestions Provider

Your first step is to implement a suggestions provider, as described in the previous section. Again, Android does not search your application, but rather queries your suggestions provider. If you do not have a suggestions provider, you will not be part of the Quick Search Box. As we will see below, this approach means you will need to think about what sort of suggestion provider to create.

Augment the Metadata

Next, you need to tell Android to tie your application into the Quick Search Box suggestion list. To do that, you need to add the `android:includeInGlobalSearch` attribute to your `searchable` XML, setting it to `true`. You probably also should consider adding the `android:searchSettingsDescription`, as this will be shown in the UI for the user to configure what suggestions the Quick Search Box shows.

Convince the User

Next, the user needs to activate your application to be included in the Quick Search Box suggestion roster. To do that, the user needs to go into Settings > Search > Searchable Items and check the checkbox associated with your application:

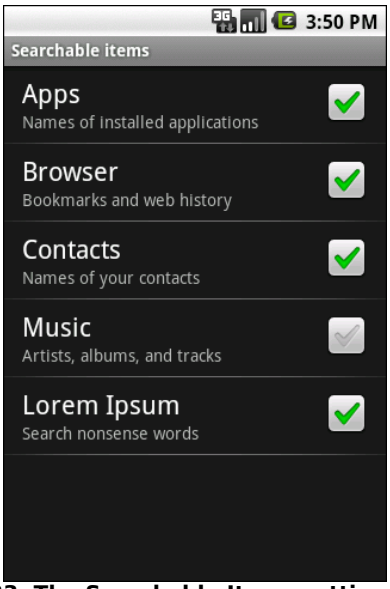


Figure 33. The Searchable Items settings screen

Your application's label and the value of `android:searchSettingsDescription` are what appears to the left of the checkbox.

You have no way of toggling this on yourself – the user has to do it. You may wish to mention this in the documentation for your application.

The Results

If you and the user do all of the above, now when the user initiates a search, your suggestions will be poured into the suggestions list, at the bottom:

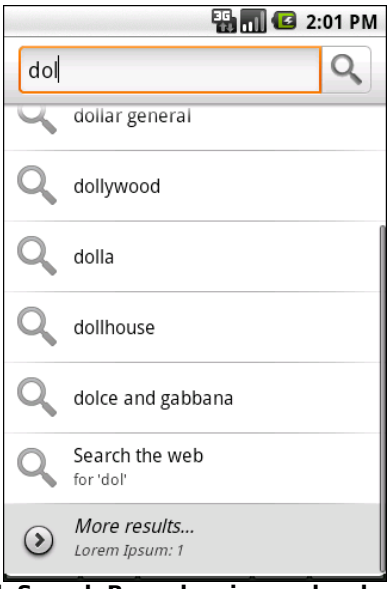


Figure 34. The Quick Search Box, showing a placeholder for application-supplied suggestions

To actually see your suggestions, the user also needs to click the arrow to "fold open" the actual suggestions:

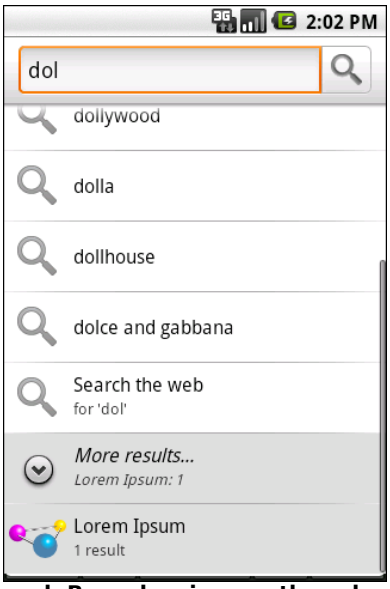


Figure 35. The Quick Search Box, showing another placeholder for application-supplied suggestions

Even here, we do not see the actual suggestion. However, if the user clicks on that item, your suggestions then take over the list:

Figure 36. The Quick Search Box, showing application-supplied suggestions

Again, Android is not showing *actual data* from your application – our list of nonsense words does not contain the value "dol". Instead, Android is showing *suggestions* from your suggestion provider based on what the user typed in. In this case, our application's suggestion provider is based on the built-in `SearchRecentSuggestionsProvider` class, meaning the suggestions are *past queries*, not actual results.

Hence, what you want to have appear in the Quick Search Box suggestion list will heavily influence what sort of suggestion provider you wish to create. While a `SearchRecentSuggestionsProvider` is simple, what you get in the Quick Search Box suggestions may not be that useful to users. Instead, you may wish to create your own custom suggestions provider, providing suggestions from actual data or other more useful sources, perhaps in addition to saved searches.

CHAPTER 7

Interactive Maps

You probably have learned about basic operations with Google Maps elsewhere, perhaps in *The Busy Coder's Guide to Android Development*. As you may recall, after going through a fair amount of hassle to obtain and manage an API key, you need to put a `MapView` in a layout used by a `MapActivity`. Then, between the `MapView` and its `MapController`, you can manage what gets displayed on the map and, to a lesser extent, get user input from the map. Notably, you can add overlays that display things on top of the map that are tied to geographic coordinates (`GeoPoint` objects), so Android can keep the overlays in sync with the map contents as the user pans and zooms.

This chapter will get into some more involved topics in the use of `MapView`, such as displaying pop-up panels when the user taps on overlay items.

The examples in this chapter are based on the original `Maps/NooYawk` example from *The Busy Coder's Guide to Android Development*. That example does two things: it displays overlay items for four New York City landmarks, and it makes a mockery of Brooklyn accents (via the unusual spelling of the project name). If you have access to *The Busy Coder's Guide to Android Development*, you may wish to review that chapter and the original example before reading further here.

Get to the Point

By default, it appears that, when the user taps on one of your `OverlayItem` icons in an `ItemizedOverlay`, all you find out is which `OverlayItem` it is, courtesy of an index into your collection of items. However, Android does provide means to find out where that item is, both in real space and on the screen.

Getting the Latitude and Longitude

You supplied the latitude and longitude – in the form of a `GeoPoint` – when you created the `OverlayItem` in the first place. Not surprisingly, you can get that back via a `getPoint()` method on `OverlayItem`. So, in an `onTap()` method, you can do this to get the `GeoPoint`:

```
@Override
protected boolean onTap(int i) {
  OverlayItem item=getItem(i);
  GeoPoint geo=item.getPoint();

  // other good stuff here

  return(true);
}
```

Getting the Screen Position

If you wanted to find the screen coordinates for that `GeoPoint`, you might be tempted to find out where the map is centered (via `getCenter()` on `MapView`) and how big the map is in terms of screen size (`getWidth()`, `getHeight()` on `MapView`) and geographic area (`getLatitudeSpan()`, `getLongitudeSpan()` on `MapView`), and do all sorts of calculations.

Good news! You do not have to do any of that.

Instead, you can get a `Projection` object from the `MapView` via `getProjection()`. This object can do the conversions for you, such as `toPixels()` to convert a `GeoPoint` into a screen `Point` for the X/Y position.

For example, take a look at the `onTap()` implementation from the `NooYawk` class in the `Maps/NooYawkRedux` sample project:

```
@Override
protected boolean onTap(int i) {
  OverlayItem item=getItem(i);
  GeoPoint geo=item.getPoint();
  Point pt=map.getProjection().toPixels(geo, null);

  String message=String.format("Lat: %f | Lon: %f\nX: %d | Y %d",
                               geo.getLatitudeE6()/1000000.0,
                               geo.getLongitudeE6()/1000000.0,
                               pt.x, pt.y);

  Toast.makeText(NooYawk.this,
                 message,
                 Toast.LENGTH_LONG).show();

  return(true);
}
```

Here, we get the `GeoPoint` (as in the previous section), get the `Point` (via `toPixels()`), and use those to customize a message for use with our `Toast`.

Note that our `Toast` message has an embedded newline (`\n`), so it is split over two lines:

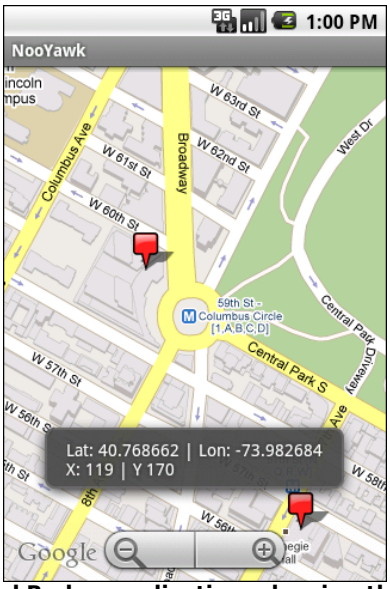


Figure 37. The NooYawkRedux application, showing the Toast with GeoPoint and Point data

Not-So-Tiny Bubbles

Of course, just because somebody taps on an item in your `ItemizedOverlay`, nothing really happens, other than letting you know of the tap. If you want something visual to occur – like the `Toast` displayed in the `Maps/NooYawkRedux` project – you have to do it yourself. And while a `Toast` is easy to implement, it tends not to be terribly useful in many cases.

A more likely reaction is to pop up some sort of bubble or panel on the screen, providing more details about the item that was tapped upon. That bubble might be display-only or fully interactive, perhaps leading to another activity for information beyond what the panel can hold.

While the techniques in this section will be couched in terms of pop-up panels over a `MapView`, the same basic concepts can be used just about anywhere in Android.

Options for Pop-up Panels

A pop-up panel is simply a `View` (typically a `ViewGroup` with contents, like a `RelativeLayout` containing widgets) that appears over the `MapView` on demand. To make one `View` appear over another, you need to use a common container that supports that sort of "Z-axis" ordering. The best one for that is `RelativeLayout`: children later in the roster of children of the `RelativeLayout` will appear over top of children that are earlier in the roster. So, if you have a `RelativeLayout` parent, with a full-screen `MapView` child followed by another `ViewGroup` child, that latter `ViewGroup` will appear to float over the `MapView`. In fact, with the use of a translucent background, you can even see the map peeking through the `ViewGroup`.

Given that, here are two main strategies for implementing pop-up panels.

One approach is to have the panel be part of the activity's layout from the beginning, but use a visibility of GONE to have it not be visible. In this case, you would define the panel in the main layout XML file, set `android:visibility="gone"`, and use `setVisibility()` on that panel at runtime to hide and show it. This works well, particularly if the panel itself is not changing much, just becoming visible and gone.

The other approach is to inflate the panel at runtime and dynamically add and remove it as a child of the `RelativeLayout`. This works well if there are many possible panels, perhaps dependent on the type of thing represented by an `OverlayItem` (e.g., restaurant versus hotel versus used car dealership).

In this section, we will examine the latter approach, as shown in the `Maps/EvenNooerYawk` sample project.

Defining a Panel Layout

The new version of `NooYawk` is designed to display panels when the user taps on items in the map, replacing the original Toast.

To do this, first, we need the actual content of a panel, as found in `res/layout/popup.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:stretchColumns="1,3"
  android:background="@drawable/popup_frame">
  <TableRow>
    <TextView
      android:text="Lat:"
      android:layout_marginRight="10dip"
    />
    <TextView android:id="@+id/latitude" />
    <TextView
      android:text="Lon:"
      android:layout_marginRight="10dip"
    />
    <TextView android:id="@+id/longitude" />
  </TableRow>
  <TableRow>
    <TextView
      android:text="X:"
      android:layout_marginRight="10dip"
    />
    <TextView android:id="@+id/x" />
    <TextView
      android:text="Y:"
      android:layout_marginRight="10dip"
    />
    <TextView android:id="@+id/y"/>
  </TableRow>
</TableLayout>
```

Here, we have a `TableLayout` containing our four pieces of data (latitude, longitude, X, and Y), with a translucent gray background (courtesy of a nine-patch graphic image).

The intent is that we will inflate instances of this class when needed. And, as we will see, we will only need one in this example, though it is possible that other applications might need more.

Creating a PopupPanel Class

To manage our panel, `NooYawk` has an inner class named `PopupPanel`. It takes the resource ID of the layout as a parameter, so it could be used to manage several different types of panels, not just the one we are using here.

Its constructor inflates the layout file (using the map's parent – the `RelativeLayout` – as the basis for inflation rules) and also hooks up a click listener to a `hide()` method (described below):

```
PopupPanel(int layout) {
  ViewGroup parent=(ViewGroup)map.getParent();

  popup=getLayoutInflater().inflate(layout, parent, false);

  popup.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
      hide();
    }
  });
}
```

`PopupPanel` also tracks an `isVisible` data member, reflecting whether or not the panel is presently on the screen.

Showing and Hiding the Panel

When it comes time to show the panel, either it is already being shown, or it is not. The former would occur if the user tapped on one item in the overlay, then tapped another right away. The latter would occur, for example, for the first tap.

In either case, we need to determine where to position the panel. Having the panel obscure what was tapped upon would be poor form. So, `PopupPanel` will put the panel either towards the top or bottom of the map, depending on where the user tapped – if they tapped in the top half of the map, the panel will go on the bottom. Rather than have the panel abut the edges of the map directly, `PopupPanel` also adds some margins – this is also important for making sure the panel and the Google logo on the map do not interfere.

If the panel is visible, `PopupPanel` calls `hide()` to remove it, then adds the panel's `View` as a child of the `RelativeLayout` with a `RelativeLayout.LayoutParams` that incorporates the aforementioned rules:

```
void show(boolean alignTop) {
  RelativeLayout.LayoutParams lp=new RelativeLayout.LayoutParams(
        RelativeLayout.LayoutParams.WRAP_CONTENT,
        RelativeLayout.LayoutParams.WRAP_CONTENT
  );

  if (alignTop) {
    lp.addRule(RelativeLayout.ALIGN_PARENT_TOP);
    lp.setMargins(0, 20, 0, 0);
  }
  else {
    lp.addRule(RelativeLayout.ALIGN_PARENT_BOTTOM);
    lp.setMargins(0, 0, 0, 60);
  }

  hide();

  ((ViewGroup)map.getParent()).addView(popup, lp);
  isVisible=true;
}

void hide() {
```

The `hide()` method, in turn, removes the panel from the `RelativeLayout`:

```
void hide() {
  if (isVisible) {
    isVisible=false;
    ((ViewGroup)popup.getParent()).removeView(popup);
  }
}
```

`PopupPanel` also has a `getView()` method, so the overlay can get at the panel `View` in order to fill in the pieces of data at runtime:

```
View getView() {
  return(popup);
}
```

Tying It Into the Overlay

To use the panel, `NooYawk` creates an instance of one as a data member of the `ItemizedOverlay` class:

```
private  PopupPanel panel=new PopupPanel(R.layout.popup);
```

Then, in the new `onTap()` method, the overlay gets the `View`, populates it, and shows it, indicating whether it should appear towards the top or bottom of the screen:

```
@Override
protected boolean onTap(int i) {
  OverlayItem item=getItem(i);
  GeoPoint geo=item.getPoint();
  Point pt=map.getProjection().toPixels(geo, null);

  View view=panel.getView();

  ((TextView)view.findViewById(R.id.latitude))
    .setText(String.valueOf(geo.getLatitudeE6()/1000000.0));
  ((TextView)view.findViewById(R.id.longitude))
    .setText(String.valueOf(geo.getLongitudeE6()/1000000.0));
  ((TextView)view.findViewById(R.id.x))
                          .setText(String.valueOf(pt.x));
  ((TextView)view.findViewById(R.id.y))
                          .setText(String.valueOf(pt.y));

  panel.show(pt.y*2>map.getHeight());

  return(true);
}
```

Here is the complete implementation of `NooYawk` from `Maps/EvenNooerYawk`, including the revised overlay class and the new `PopupPanel` class:

```
package com.commonsware.android.maps;

import android.app.Activity;
import android.graphics.Canvas;
import android.graphics.Point;
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.view.KeyEvent;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
```

```
import android.widget.RelativeLayout;
import android.widget.TextView;
import android.widget.Toast;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.ItemizedOverlay;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;
import com.google.android.maps.MapView.LayoutParams;
import com.google.android.maps.MyLocationOverlay;
import com.google.android.maps.OverlayItem;
import java.util.ArrayList;
import java.util.List;

public class NooYawk extends MapActivity {
  private MapView map=null;
  private MyLocationOverlay me=null;

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

    map=(MapView)findViewById(R.id.map);

    map.getController().setCenter(getPoint(40.76793169992044,
                                           -73.98180484771729));
    map.getController().setZoom(17);
    map.setBuiltInZoomControls(true);

    Drawable marker=getResources().getDrawable(R.drawable.marker);

    marker.setBounds(0, 0, marker.getIntrinsicWidth(),
                           marker.getIntrinsicHeight());

    map.getOverlays().add(new SitesOverlay(marker));

    me=new MyLocationOverlay(this, map);
    map.getOverlays().add(me);
  }

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

    me.enableCompass();
  }

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

    me.disableCompass();
  }
```

```
@Override
protected boolean isRouteDisplayed() {
  return(false);
}

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_S) {
    map.setSatellite(!map.isSatellite());
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_Z) {
    map.displayZoomControls(true);
    return(true);
  }

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

private GeoPoint getPoint(double lat, double lon) {
  return(new GeoPoint((int)(lat*1000000.0),
                      (int)(lon*1000000.0)));
}

private class SitesOverlay extends ItemizedOverlay<OverlayItem> {
  private List<OverlayItem> items=new ArrayList<OverlayItem>();
  private Drawable marker=null;
  private  PopupPanel panel=new PopupPanel(R.layout.popup);

  public SitesOverlay(Drawable marker) {
    super(marker);
    this.marker=marker;

    items.add(new OverlayItem(getPoint(40.748963847316034,
                                       -73.96807193756104),
                              "UN", "United Nations"));
    items.add(new OverlayItem(getPoint(40.76866299974387,
                                       -73.98268461227417),
                              "Lincoln Center",
                              "Home of Jazz at Lincoln Center"));
    items.add(new OverlayItem(getPoint(40.765136435316755,
                                       -73.97989511489868),
                              "Carnegie Hall",
            "Where you go with practice, practice, practice"));
    items.add(new OverlayItem(getPoint(40.70686417491799,
                                       -74.01572942733765),
                              "The Downtown Club",
                    "Original home of the Heisman Trophy"));

    populate();
  }

  @Override
```

```
    protected OverlayItem createItem(int i) {
      return(items.get(i));
    }

    @Override
    public void draw(Canvas canvas, MapView mapView,
                     boolean shadow) {
      super.draw(canvas, mapView, shadow);

      boundCenterBottom(marker);
    }

    @Override
    protected boolean onTap(int i) {
      OverlayItem item=getItem(i);
      GeoPoint geo=item.getPoint();
      Point pt=map.getProjection().toPixels(geo, null);

      View view=panel.getView();

      ((TextView)view.findViewById(R.id.latitude))
        .setText(String.valueOf(geo.getLatitudeE6()/1000000.0));
      ((TextView)view.findViewById(R.id.longitude))
        .setText(String.valueOf(geo.getLongitudeE6()/1000000.0));
      ((TextView)view.findViewById(R.id.x))
                             .setText(String.valueOf(pt.x));
      ((TextView)view.findViewById(R.id.y))
                             .setText(String.valueOf(pt.y));

      panel.show(pt.y*2>map.getHeight());

      return(true);
    }

    @Override
    public int size() {
      return(items.size());
    }
  }

  class PopupPanel {
    View popup;
    boolean isVisible=false;

    PopupPanel(int layout) {
      ViewGroup parent=(ViewGroup)map.getParent();

      popup=getLayoutInflater().inflate(layout, parent, false);

      popup.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
          hide();
        }
      });
```

```
    }

    View getView() {
      return(popup);
    }

    void show(boolean alignTop) {
      RelativeLayout.LayoutParams lp=new RelativeLayout.LayoutParams(
            RelativeLayout.LayoutParams.WRAP_CONTENT,
            RelativeLayout.LayoutParams.WRAP_CONTENT
      );

      if (alignTop) {
        lp.addRule(RelativeLayout.ALIGN_PARENT_TOP);
        lp.setMargins(0, 20, 0, 0);
      }
      else {
        lp.addRule(RelativeLayout.ALIGN_PARENT_BOTTOM);
        lp.setMargins(0, 0, 0, 60);
      }

      hide();

      ((ViewGroup)map.getParent()).addView(popup, lp);
      isVisible=true;
    }

    void hide() {
      if (isVisible) {
        isVisible=false;
        ((ViewGroup)popup.getParent()).removeView(popup);
      }
    }
  }
}
```

The resulting panel looks like this when it is towards the bottom of the screen:

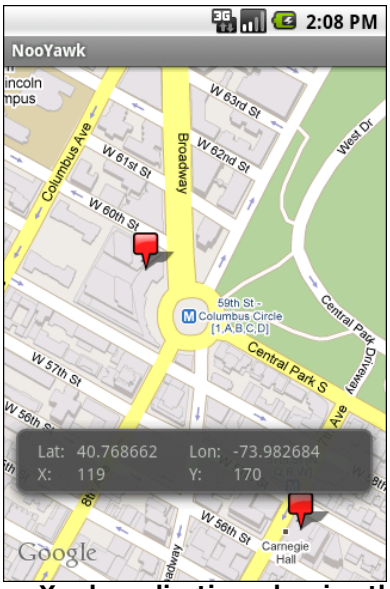


Figure 38. The EvenNooerYawk application, showing the PopupPanel towards the bottom

...and like this when it is towards the top:

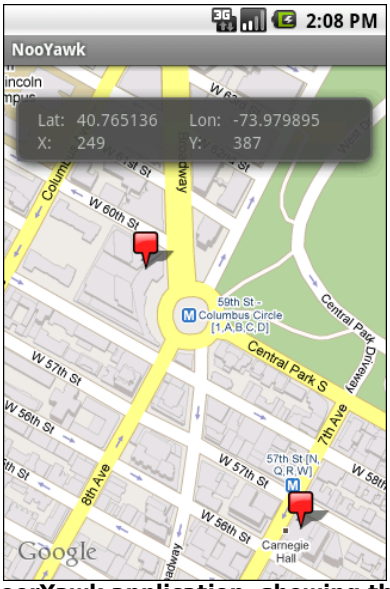


Figure 39. The EvenNooerYawk application, showing the PopupPanel towards the top

Sign, Sign, Everywhere a Sign

Our examples for Manhattan have treated each of the four locations as being the same – they are all represented by the same sort of marker. That is the natural approach to creating an `ItemizedOverlay`, since it takes the marker `Drawable` as a constructor parameter.

It is not the only option, though.

Selected States

One flaw in our current one-`Drawable`-for-everyone approach is that you cannot tell which item was selected by the user, either by tapping on it or by using the D-pad (or trackball or whatever). A simple PNG icon will look the same as it will in every other state.

However, back in the chapter on Drawable techniques, we saw the `StateListDrawable` and its accompanying XML resource format. We can use one of those here, to specify a separate icon for selected and regular states.

In the `Maps/ILuvNooYawk` sample, we change up the icons used for our four `OverlayItem` objects. Specifically, in the next section, we will see how to associate a distinct `Drawable` for each item. Those `Drawable` resources will actually be `StateListDrawable` objects, using XML such as:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:state_selected="true"
    android:drawable="@drawable/blue_sel_marker"
  />
  <item
    android:drawable="@drawable/blue_marker"
  />
</selector>
```

This indicates that we should use one PNG in the default state and a different PNG (one with a yellow highlight) when the `OverlayItem` is selected.

Per-Item Drawables

To use a different `Drawable` per `OverlayItem`, we need to create a custom `OverlayItem` class. Normally, you can skip this, and just use `OverlayItem` directly. But, `OverlayItem` has no means to change its `Drawable` used for the marker, so we have to extend it and override `getMarker()` to handle a custom `Drawable`.

Here is one possible implementation of a `CustomItem` class:

```
class CustomItem extends OverlayItem {
  Drawable marker=null;

  CustomItem(GeoPoint pt, String name, String snippet,
             Drawable marker) {
    super(pt, name, snippet);

    this.marker=marker;
  }

  @Override
  public Drawable getMarker(int stateBitset) {
    setState(marker, stateBitset);

    return(marker);
  }
}
```

This class takes the `Drawable` to use as a constructor parameter, holds onto it, and returns it in the `getMarker()` method. However, in `getMarker()`, we also need to call `setState()` – if we are using `StateListDrawable` resources, the call to `setState()` will cause the `Drawable` to adopt the appropriate state (e.g., selected).

Of course, we need to prep and feed a `Drawable` to each of the `CustomItem` objects. In the case of `ILuvNooYawk`, when our `SitesOverlay` creates its items, it uses a `getMarker()` method to access each item's `Drawable`:

```
private Drawable getMarker(int resource) {
  Drawable marker=getResources().getDrawable(resource);

  marker.setBounds(0, 0, marker.getIntrinsicWidth(),
                   marker.getIntrinsicHeight());
  boundCenter(marker);
```

```
  return(marker);
}
```

Here, we get the `Drawable` resources, set its bounds (for use with hit testing on taps), and use `boundCenter()` to control the way the shadow falls. For icons like the original push pin used by `NooYawk`, `boundCenterBottom()` will cause the icon and its shadow to make it seem like the icon is rising up off the face of the map. For icons like `ILuvNooYawk` uses, `boundCenter()` will cause the icon and shadow to make it seem like the icon is hovering flat over top of the map.

Changing Drawables Dynamically

It is also possible to change the `Drawable` used by a item at runtime, beyond simply changing it from normal to selected state. For example, `ILuvNooYawk` allows you to press the H key and toggle the selected item from its normal icon to a heart:

```
@Override
 public boolean onKeyDown(int keyCode, KeyEvent event) {
   if (keyCode == KeyEvent.KEYCODE_S) {
     map.setSatellite(!map.isSatellite());
     return(true);
   }
   else if (keyCode == KeyEvent.KEYCODE_Z) {
     map.displayZoomControls(true);
     return(true);
   }
   else if (keyCode == KeyEvent.KEYCODE_H) {
     sites.toggleHeart();

     return(true);
   }

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

To make this work, our `SitesOverlay` needs to implement `toggleHeart()`:

```
void toggleHeart() {
  CustomItem focus=getFocus();

  if (focus!=null) {
```

```
    focus.toggleHeart();
  }

  map.invalidate();
}
```

Here, we just find the selected item and delegate `toggleHeart()` to it. This, of course, assumes both that `CustomItem` has a `toggleHeart()` implementation and knows what heart to use.

So, rather than the simple `CustomItem` shown above, we need a more elaborate implementation:

```
class CustomItem extends OverlayItem {
  Drawable marker=null;
  boolean isHeart=false;
  Drawable heart=null;

  CustomItem(GeoPoint pt, String name, String snippet,
             Drawable marker, Drawable heart) {
    super(pt, name, snippet);

    this.marker=marker;
    this.heart=heart;
  }

  @Override
  public Drawable getMarker(int stateBitset) {
    Drawable result=(isHeart ? heart : marker);

    setState(result, stateBitset);

    return(result);
  }

  void toggleHeart() {
    isHeart=!isHeart;
  }
}
```

Here, the `CustomItem` gets its own icon and the heart icon in the constructor, and `toggleHeart()` just toggles between them. The key is that we `invalidate()` the `MapView` in the `SitesOverlay` implementation of `toggleHeart()` – that causes the map, and its overlay items, to be redrawn, causing the icon `Drawable` to change on the screen.

This means that while we start with custom icons per item:

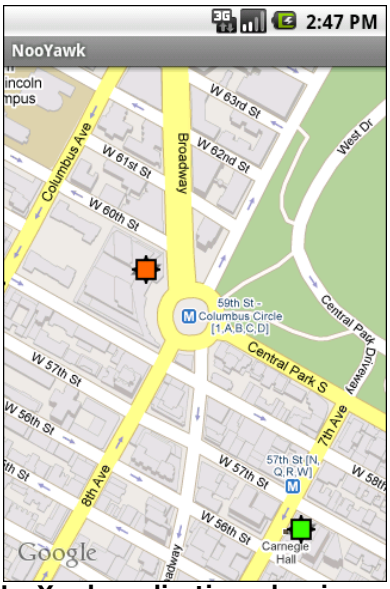


Figure 40. The ILuvNooYawk application, showing custom icons per item

...we can change those by clicking on an item and pressing the H key:

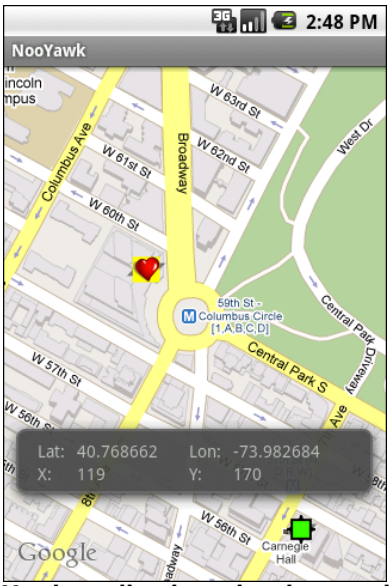


Figure 41. The ILuvNooYawk application, showing one item's icon toggled to a heart (and selected)

In A New York Minute. Or Hopefully a Bit Faster.

In the case of `NooYawk`, we have all our data points for the overlay items up front – they are hard-wired into the code. This is not going to be the case in most applications. Instead, the application will need to load the items out of a database or a Web service.

In the case of a database, assuming a modest number of items, the difference between having the items hard-wired in code or in the database is slight. Yes, the actual implementation will be substantially different, but you can query the database and build up your `ItemizedOverlay` all in one shot, when the map is slated to appear on-screen.

Where things get interesting is when you need to use a Web service or similar slow operation to get the data.

Where things get even more interesting is when you want that data to change after it was already loaded – on a timer, on user input, etc. For example, it may be that you have hundreds of thousands of data points, only a tiny fraction of which will be visible on the map at any time. If the user elects to visit a different portion of the map, you need to dump the old overlay items and grab a new set.

In either case, you can use an `AsyncTask` to populate your `ItemizedOverlay` and add it to the map once the data is ready. You can see this in `Maps/NooYawkAsync`, where we kick off an `OverlayTask` in the `NooYawk` implementation of `onCreate()`:

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

  map=(MapView)findViewById(R.id.map);

  map.getController().setCenter(getPoint(40.76793169992044,
                                         -73.98180484771729));
  map.getController().setZoom(17);
  map.setBuiltInZoomControls(true);

  me=new MyLocationOverlay(this, map);
  map.getOverlays().add(me);

  new OverlayTask().execute();
}
```

...and then use that to load the data in the background, in this case using a `sleep()` call to simulate real work:

```
class OverlayTask extends AsyncTask<Void, Void, Void> {
  @Override
  public void onPreExecute() {
    if (sites!=null) {
      map.getOverlays().remove(sites);
      map.invalidate();
      sites=null;
    }
  }

  @Override
  public Void doInBackground(Void... unused) {
    SystemClock.sleep(5000);            // simulated work
```

```
    sites=new SitesOverlay();

    return(null);
  }

  @Override
  public void onPostExecute(Void unused) {
    map.getOverlays().add(sites);
    map.invalidate();
  }
}
```

As with changing an item's `Drawable` on the fly, you need to `invalidate()` the map to make sure it draws the overlay and its items.

In this case, we also hook up the R key to simulate a manual refresh of the data. This just invokes another `OverlayTask`, which removes the old overlay and creates a fresh one:

```
@Override
 public boolean onKeyDown(int keyCode, KeyEvent event) {
   if (keyCode == KeyEvent.KEYCODE_S) {
     map.setSatellite(!map.isSatellite());
     return(true);
   }
   else if (keyCode == KeyEvent.KEYCODE_Z) {
     map.displayZoomControls(true);
     return(true);
   }
   else if (keyCode == KeyEvent.KEYCODE_H) {
     sites.toggleHeart();

     return(true);
   }
   else if (keyCode == KeyEvent.KEYCODE_R) {
     new OverlayTask().execute();

     return(true);
   }

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

PART II – Advanced Media

CHAPTER 8

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.

After an overview of the role of the animation framework, we go in-depth to animate the movement of a widget across the screen. We then look at alpha animations, for fading widgets in and out. We then see how you can get control during the lifecycle of an animation, how to control the acceleration of animations, and how to group animations together for parallel execution. Finally, we see how the same framework can now be used to control the animation for the switching of activities.

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 Sliding Panel

Some Android applications employ a sliding panel, one that is off-screen most of the time but can be called up by the user (e.g., via a menu) when desired. When anchored at the bottom of the screen, the effect is akin to the Android menu system, with a container that slides up from the bottom and slides down and out when being removed. However, while menus are limited to menu choices, Android's animation framework lets one create a sliding panel containing whatever widgets you might want.

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

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

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

Introducing SlidingPanel

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

This class implements a layout that works as a panel, anchored to the bottom of the screen. A toggle() method can be called by the activity to hide or show the panel. The panel itself is a LinearLayout, so you can put whatever contents you want in there.

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

Here is the opening animation:

```
anim=new TranslateAnimation(0.0f, 0.0f,
                            getLayoutParams().height,
                            0.0f);
```

Our fromXDelta and toXDelta are both 0, since we are not shifting the panel's position along the horizontal axis. Our fromYDelta is the panel's height according to its layout parameters (representing how big we want the panel to be), because we want the panel to start the animation at the bottom of the screen; our toYDelta is 0 because we want the panel 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,
                            getLayoutParams().height);
```

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

The result is a container that can be closed:

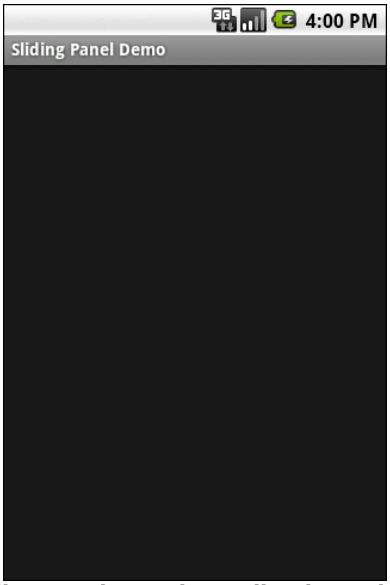


Figure 42. The SlidingPanel sample application, with the panel closed

...or open, in this case toggled via a menu choice in the `SlidingPanelDemo` activity:

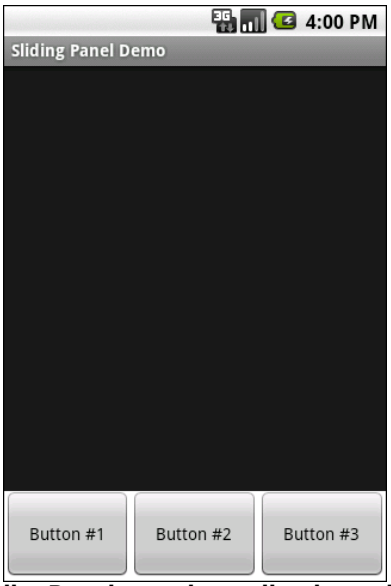


Figure 43. The SlidingPanel sample application, with the panel open

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 `SlidingPanel` 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/SlidingPanelEx`, which demonstrates a panel 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 panel, 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 `SlidingPanel`, keeping the panel closed is a matter of calling `setVisibility()` on the contents with `GONE`.

However, you cannot do that when the animation begins; otherwise, the panel 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 animation 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 `SlidingPanel`:

```
Animation.AnimationListener collapseListener=new Animation.AnimationListener() {
  public void onAnimationEnd(Animation animation) {
    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 panel.

Loose Fill

You will see attributes, available on `Animation`, named `android:fillEnabled` and `android:fillAfter`. Reading those, you may think that you can dispense with the `AnimationListener` and just use those to arrange to have your widget wind up being "permanently" in the state represented by the end of the animation. All you would have to do is set each of those to true in your animation XML (or the equivalent in Java), and you would be set.

At least for `TranslateAnimation`, you would be mistaken.

It actually will look like it works – the animated widgets will be drawn in their new location. However, if those widgets are clickable, the will not be clicked in their new location, but rather in their old one. This, of course, is not terribly useful.

Hence, even though it is annoying, you will want to use the `AnimationListener` techniques described in this chapter.

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 `SlidingPanel`:

```
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.

Android 1.6 added some new interpolators. Notable are `BounceInterpolator` (which gives a bouncing effect as the animation nears the end) and

`OvershootInterpolator` (which goes beyond the end of the animation range, then returns to the endpoint).

Animate. Set. Match.

For the `Animation/SlidingPanelEx` project, though, we want the panel 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 `SlidingPanel` implementation in `Animation/SlidingPanelEx`:

```
public void toggle() {
  TranslateAnimation anim=null;
  AnimationSet set=new AnimationSet(true);

  isOpen=!isOpen;

  if (isOpen) {
    setVisibility(View.VISIBLE);
    anim=new TranslateAnimation(0.0f, 0.0f,
                                getLayoutParams().height,
                                0.0f);
  }
  else {
    anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                getLayoutParams().height);
    anim.setAnimationListener(collapseListener);
    set.addAnimation(fadeOut);
  }

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

If the panel is to be opened, we make the contents visible (so we can animate the motion upwards), and create a `TranslateAnimation` for the upward movement. If the panel 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.

Active Animations

Starting with Android 1.5, users could indicate if they wanted to have inter-activity animations: a slide-in/slide-out effect as they switched from activity to activity. However, at that time, they could merely toggle this setting on or off, and applications had no control over these animations whatsoever.

Starting in Android 2.0, though, developers have a bit more control. Specifically:

- Developers can call `overridePendingTransition()` on an `Activity`, typically after calling `startActivity()` to launch another activity or `finish()` to close up the current activity. The `overridePendingTransition()` indicates an in/out animation pair that should be applied as control passes from this activity to the next one, whether that one is being started (`startActivity()`) or is the one previous on the stack (`finish()`).
- Developers can start an activity via an `Intent` containing the `FLAG_ACTIVITY_NO_ANIMATION` flag. As the name suggests, this flag requests that animations on the transitions involving this activity be suppressed.

These are prioritized as follows:

1. Any call to `overridePendingTransition()` is always taken into account
2. Lacking that, `FLAG_ACTIVITY_NO_ANIMATION` will be taken into account
3. In the normal case, where neither of the two are used, whatever the user's preference, via the Settings application, is applied

CHAPTER 9

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.

First, we examine how to set up an activity showing a preview of the camera's output, much like the LCD viewfinder on a dedicated digital camera. We then extend that example to actually take and store a picture. After a brief discussion of auto-focus, we wrap with material on other parameters you may be able to set to control the actual picture being taken.

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-sdk
      android:minSdkVersion="3"
      android:targetSdkVersion="6"
  />
  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"
  />
  <uses-feature android:name="android.hardware.camera" />
  <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 Manifest

Starting with Android 1.6, if your application absolutely needs a camera, you can include a `<uses-feature>` element in the `AndroidManifest.xml` file to declare that requirement, alongside your `<uses-permission>` element for the `CAMERA` permission:

```
<uses-feature android:name="android.hardware.camera" />
```

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();

  try {
    camera.setPreviewDisplay(previewHolder);
  }
  catch (Throwable t) {
    Log.e("PreviewDemo-surfaceCallback",
          "Exception in setPreviewDisplay()", t);
    Toast
      .makeText(PreviewDemo.this, t.getMessage(), Toast.LENGTH_LONG)
      .show();
  }
}
```

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.

Here is the full `SurfaceHolder.Callback` implementation:

```
SurfaceHolder.Callback surfaceCallback=new SurfaceHolder.Callback() {
  public void surfaceCreated(SurfaceHolder holder) {
    camera=Camera.open();

    try {
      camera.setPreviewDisplay(previewHolder);
    }
    catch (Throwable t) {
      Log.e("PreviewDemo-surfaceCallback",
            "Exception in setPreviewDisplay()", t);
      Toast
        .makeText(PreviewDemo.this, t.getMessage(), Toast.LENGTH_LONG)
        .show();
    }
  }

  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();
  }

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

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, Android devices do 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 ||
      keyCode==KeyEvent.KEYCODE_SEARCH) {
    takePicture();

    return(true);
  }

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

Since the HTC Magic does not have a hardware camera button, we also watch for `KEYCODE_SEARCH` for the dedicated search key, which is in the upper-right portion of the Magic's face when the device is held in landscape mode. You could similarly watch for a D-pad center button click or whatever you wish.

Taking a Picture

Once it is time to take a picture, all you need to do is 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 Android devices presently only support 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.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. 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.

Using AsyncTask

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.

Android 1.5 offers a work queue model, in the form of `AsyncTask`. `AsyncTask` manages a thread pool and work queue – all we need to do is hand it the work to be done.

So, we can create an `AsyncTask` implementation, called `SavePhotoTask`, as follows:

```
class SavePhotoTask extends AsyncTask<byte[], String, String> {
  @Override
  protected String doInBackground(byte[]... jpeg) {
    File photo=new File(Environment.getExternalStorageDirectory(),
                        "photo.jpg");

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

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

      fos.write(jpeg[0]);
      fos.close();
    }
    catch (java.io.IOException e) {
      Log.e("PictureDemo", "Exception in photoCallback", e);
    }

    return(null);
  }
}
```

Our `doInBackground()` implementation gets the byte array we received from Android. 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. 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.

Finally, we arrange for our `PhotoCallback` to execute our `SavePhotoTask`:

```
Camera.PictureCallback photoCallback=new Camera.PictureCallback() {
  public void onPictureTaken(byte[] data, Camera camera) {
    new SavePhotoTask().execute(data);
    camera.startPreview();
  }
};
```

Maintaining Your Focus

Android devices may support auto-focus. As with the camera itself, auto-focus is a device-specific capability and may not be available on all devices.

If you *need* auto-focus in your application, you will first need to add another `<uses-feature>` element to your manifest, to declare your interest in auto-focus:

```
<uses-feature android:name="android.hardware.camera.autofocus" />
```

Next, you need to determine when to apply auto-focus. For devices with a dedicated camera hardware button, that button might support a "half-press" that raises a `KEYCODE_FOCUS` `KeyEvent`. The T-Mobile G1 offers this, for example.

Then, to trigger auto-focus itself in your code, call `autoFocus()` on the `Camera` object. You will need to supply a callback object that will be notified when the focus operation is complete, so you know it is safe to take a picture, for example. If a device does not support auto-focus, the callback object will be notified anyway, so you can always rely upon the callback being notified when the camera is as focused as it will ever be.

Note that if you can take advantage of auto-focus but do not absolutely need it, there is an `android:required` attribute you can add to your `<uses-feature>` element – setting that to false means your application can use auto-focus methods but will still install on devices that lack an auto-focus camera (e.g., HTC Tattoo). Note that `android:required` is not presently documented, though that appears to be a documentation bug. To find out if auto-focus is available on a given device, call `getFocusMode()` on your `Camera.Parameters` object to see if it returns `FOCUS_MODE_FIXED`, in which case auto-focus is unavailable.

All the Bells and Whistles

Starting with Android 2.0, the Camera.Parameters object offers a wide range of settings that you can control over how a picture gets taken, much more than merely the size and file type. Settings you can manage include:

- Anti-banding effects
- Color effects (e.g., "negative" or inverse image, sepia-tone image)
- Flash settings (on? off? always on? anti-red-eye mode?)
- Focus mode (fixed? macro? infinity?)
- JPEG quality levels, for both the image and the thumbnail representation of the image
- White balance levels

For all of these, and others, not only can you get the current setting and change it, but you can also obtain a list of the available settings, perhaps to populate a ListView or selection dialog for the user.

You can now also supply GPS data to the camera, which will encode that information into the EXIF data of the JPEG image.

CHAPTER 10

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 make use of 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

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

If you want to play back music, particularly material in MP3 format, you will want to use 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();
    pause.setEnabled(false);
    stop.setEnabled(false);
```

```
    try {
      mp.prepare();
      mp.seekTo(0);
      play.setEnabled(true);
    }
    catch (Throwable t) {
      goBlooey(t);
    }
  }

  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 `prepare()` on the existing `MediaPlayer` to enable it to be played again and `seekTo()` to move the playback point to the beginning. If we were using an external file as our media source, it would be better to call `prepareAsync()`.

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

Figure 44. The AudioDemo sample application

Moving Pictures

In addition to perhaps using `MediaPlayer`, video clips get their own widget, the `VideoView`. Put it in a layout, feed it an MP4 video clip, and you get playback! We will see using `MediaPlayer` for video in the next section.

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(Environment.getExternalStorageDirectory(),
                       "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:

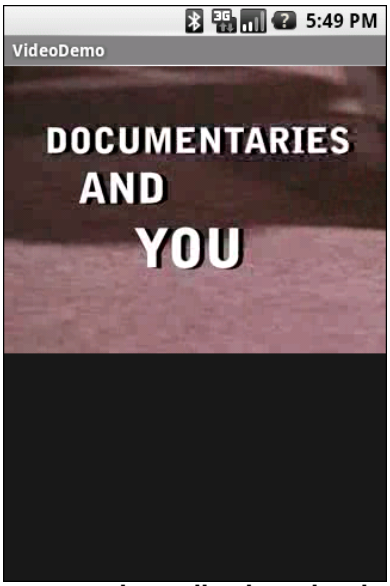


Figure 45. The VideoDemo sample application, showing a Creative Commons-licensed video clip

Tapping on the video will pop up the playback controls:

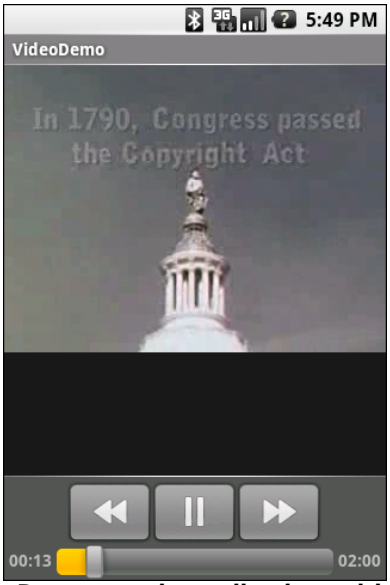


Figure 46. The VideoDemo sample application, with the media controls displayed

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

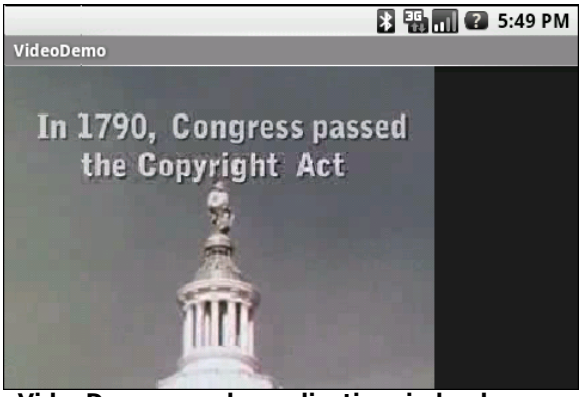


Figure 47. The VideoDemo sample application, in landscape mode, with the video clip scaled to fit

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 when it works at all, while playback on the T-Mobile G1 is very smooth.

Pictures in the Stream

`VideoView` is nice, but you get a bit more control if you use `MediaPlayer`. It is somewhat more involved to set up, though, in part because it involves a `SurfaceView`, introduced in the chapter on the camera.

The sample code for this project is released as a separate open source project, called `vidtry`, as it allows you to try video clips, with an emphasis on streaming video. You can find the complete source code to `vidtry` out on Github. You may want to have the full source code with you when reviewing this section, as it is a bit more extensive than most.

NOTE: playing video on the Android emulator may work for you, but it is not terribly likely. Video playback requires graphic acceleration to work well, and the emulator does not have graphics acceleration – regardless of

the capabilities of the actual machine the emulator runs on. Hence, if you try playing back video in the emulator, expect problems. If you are serious about doing Android development with video playback, you definitely need to acquire a piece of Android hardware.

At its core, `vidtry` simply plays back video, much like the example of `VideoView` in the preceding section:

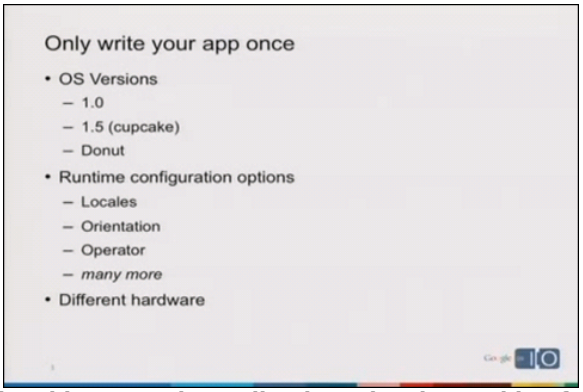


Figure 48. The vidtry sample application, showing a video from the 2009 Google I/O Conference

However, `vidtry` also supports streaming video and custom pop-up control panels:

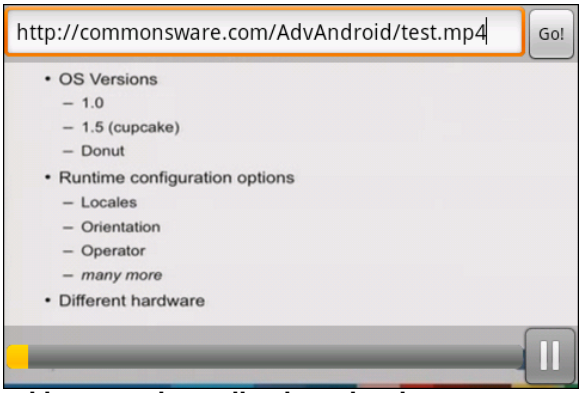


Figure 49. The vidtry sample application, showing pop-up panels overlaying the video

Rules for Streaming

Streaming video with Android is a dicey proposition. If you are in control of the media being streamed, getting it to work is eminently possible. If you are trying to stream existing media not designed for use with Android, as they say in the United States, "your mileage may vary".

This section focuses on HTTP streaming, as that is what most people would be in position to serve up. RTSP streaming should also be available, but there are far fewer RTSP servers than Web servers.

Here are some guidelines for serving HTTP streaming video to Android:

1. The media in question needs to be "safe for streaming". For MP4 files, for example, the rule is "the moov atom must appear before the mdat atom". That may happen as a result of how you create the MP4 files. If not, you may need to use tools to add "hints" to the MP4 file to achieve this atom ordering. For example, on Linux, you can use `MP4Box -hint` to accomplish this, where `MP4Box` can be found in the `gpac` package for Ubuntu.
2. There used to be a rule that the height and width each had to be divisible by 16. It is unclear if that is still a rule or merely an optimization at this point.
3. If you have the space to store multiple editions of the video for serving, consider creating ones for commonplace sizes, such as one designed to work on a 480x320 landscape screen. The less work the device has to do to scale the image, the better battery life will be.

Establishing the Surface

Setting up a `SurfaceView` for video playback works much the same way as setting up a `SurfaceView` for the camera preview. You create the `SurfaceView` and get its corresponding `SurfaceHolder`, then start using the surface once the surface has been prepared.

For example, here is where we set up a `SurfaceView` in vidtry, in the `Player` activity's `onCreate()` method:

```
surface=(TappableSurfaceView)findViewById(R.id.surface);
surface.addTapListener(onTap);
holder=surface.getHolder();
holder.addCallback(this);
holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
```

Note that we are using a `TappableSurfaceView`. This is a custom subclass of `SurfaceView` that supports touch events – more on this in a later section. Outside of touch behavior, though, `TappableSurfaceView` works identically to a regular `SurfaceView`.

So, we get the surface out of our layout, add a listener for touch events, get its `SurfaceHolder`, tell the `SurfaceHolder` to keep the `Player` informed of the surface's own lifecycle, and set the type of the surface to be `SURFACE_TYPE_PUSH_BUFFERS` (meaning lower level code gets to write directly to the surface). That, plus the regular view creation process, will trigger the `SurfaceView` to be constructed and made available for use.

Floating Panels

The `SurfaceView` is set up to take up whatever space it needs to play back the video. Typically, this will involve filling one of the two axes, depending on the aspect ratio of the video and the device's display.

Full-screen video playback is fairly normal for an application like this. However, what may not be obvious is how to handle pop-up control panels, where controls for pausing playback and such appear to float over top of the video.

There are three components of the technique for making that work:

1. In layouts, anything later in the container (e.g., later in the XML listing of the layout file) appears higher in the Z-axis. That means if you define the `SurfaceView` first, and other widgets later, those other widgets will appear to float over top of the video.

2. Since you control the visibility of any widget, you can arrange to have those floating widgets be invisible (or gone) normally, and only show up when the user requests, perhaps as a result of a screen tap.

3. If you have several controls that you want grouped in a translucent panel, just put them in one container (e.g., `RelativeLayout`) and set the background color of that container to be a translucent value (e.g., `#40808080` for a translucent light gray).

For example, here is the layout that drives the `Player` activity (`res/layout/main.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.commonsware.android.vidtry.TappableSurfaceView
    android:id="@+id/surface"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center">
  </com.commonsware.android.vidtry.TappableSurfaceView>
  <RelativeLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  >
    <LinearLayout
      android:id="@+id/top_panel"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal"
      android:background="#40808080"
      android:visibility="visible"
      android:layout_alignParentTop="true"
    >
      <AutoCompleteTextView android:id="@+id/address"
        android:layout_width="0px"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:completionThreshold="1"
      />
      <Button android:id="@+id/go"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/go"
        android:enabled="false"
      />
    </LinearLayout>
    <LinearLayout
```

```
      android:id="@+id/bottom_panel"
      android:layout_height="wrap_content"
      android:layout_width="fill_parent"
      android:orientation="horizontal"
      android:background="#40808080"
      android:visibility="gone"
      android:layout_alignParentBottom="true"
    >
      <ProgressBar android:id="@+id/timeline"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="0px"
        android:layout_weight="1"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:paddingLeft="2px"
      />
      <ImageButton android:id="@+id/media"
        style="@style/MediaButton"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/ic_media_pause"
        android:enabled="false"
      />
    </LinearLayout>
  </RelativeLayout>
</FrameLayout>
```

You will see that, in addition to our `TappableSurfaceView`, the layout has a pair of `LinearLayout` widgets with the aforementioned background color. One, on the top, contains an `AutoCompleteTextView` to be used for entering URLs of videos to watch, plus a button to trigger playback of that video. The other contains a `ProgressBar` that will serve as the video playback timeline, plus a button to pause or resume playback. The bottom panel is set to have `android:visibility = "gone"`, so only the top panel will be visible when you first run the application.

Playing Video

When the user types in a URL and clicks the "go" button, we call `playVideo()` on our `Player`:

```
private void playVideo(String url) {
   try {
     media.setEnabled(false);

     if (player==null) {
```

```
        player=new MediaPlayer();
        player.setScreenOnWhilePlaying(true);
      }
      else {
        player.stop();
        player.reset();
      }

      player.setDataSource(url);
      player.setDisplay(holder);
      player.setAudioStreamType(AudioManager.STREAM_MUSIC);
      player.setOnPreparedListener(this);
      player.prepareAsync();
//      player.setOnBufferingUpdateListener(this);
      player.setOnCompletionListener(this);
    }
    catch (Throwable t) {
      Log.e(TAG, "Exception in media prep", t);
      goBlooey(t);
    }
  }
```

Here, we do several things of significance:

- We either create a new `MediaPlayer` (if this is the first video we have played) or `stop()` and `reset()` the existing player
- We tell the `MediaPlayer` to load the user-supplied URL into our `SurfaceView` (via its `SurfaceHolder`)
- We tell the `MediaPlayer` to let us know when the video is prepared and has finished playback
- We tell the `MediaPlayer` to `prepareAsync()`, which will begin streaming down the initial portion of the video file

Note that we also call `setScreenOnWhilePlaying()` – this will keep the screen lock from taking over while video is actually playing back.

After a few moments, `MediaPlayer` should have downloaded enough information to begin actually playing the video. At that point, it will call us back via the `onPrepared()` in the `Player`, as is required by the `MediaPlayer.OnPreparedListener` interface we are implementing and used in `setOnPreparedListener()`.

```
public void onPrepared(MediaPlayer mediaplayer) {
  width=player.getVideoWidth();
  height=player.getVideoHeight();

  if (width!=0 && height!=0) {
    holder.setFixedSize(width, height);
    timeline.setProgress(0);
    timeline.setMax(player.getDuration());
    player.start();
  }

  media.setEnabled(true);
}
```

Here, we:

- Get the height and width of the video file from the `MediaPlayer`
- Tell the `SurfaceView` to use the same height and width – it will automatically determine appropriate scaling if the video is larger than the screen size
- Reset the timeline `ProgressBar` to `0` and set its maximum to be the duration of the video clip, as reported by the `MediaPlayer`
- Start actual playback of the video

Note that Android is very finicky about its streaming video. A video that might work fine on one device will not work well on another. If you are going to be developing applications that rely upon streaming video, it is best if you obtain 2-3 devices, with different screen sizes and from different manufacturers, and test your videos on those devices to ensure they will work.

Touchable Controls

We still have not done much about those two panels. One, containing the URL field and button, is still visible. The other, containing the timeline and play/pause button, is gone. It would be nice if both would be gone while the video is playing, yet still be retrievable when the user wants them.

The panels are set to "automatically" hide after a period of inactivity. That is accomplished by:

- Tracking the `lastActionTime` on any user input event (`lastActionTime = SystemClock.elapsedRealtime()`), so we know when the user last did something
- Use `postDelayed()` to set up a one-per-second check to see if enough time has elapsed since `lastActionTime`, at which point bottom panel is hidden
- The back button is used to close the top panel, when it is displayed

Bringing the panels up again is handled via touch events on our `SurfaceView`, implemented in a `TappableSurfaceView` class:

```
package com.commonsware.android.vidtry;

import android.content.Context;
import android.view.GestureDetector;
import android.view.GestureDetector.SimpleOnGestureListener;
import android.view.MotionEvent;
import android.view.SurfaceView;
import android.util.AttributeSet;
import java.util.ArrayList;

public class TappableSurfaceView extends SurfaceView {
  private ArrayList<TapListener> listeners=new ArrayList<TapListener>();
  private GestureDetector gesture=null;

  public TappableSurfaceView(Context context,
                             AttributeSet attrs) {
    super(context, attrs);
  }

  public boolean onTouchEvent(MotionEvent event) {
    if (event.getAction()==MotionEvent.ACTION_UP) {
      gestureListener.onSingleTapUp(event);
    }

    return(true);
  }

  public void addTapListener(TapListener l) {
    listeners.add(l);
  }

  public void removeTapListener(TapListener l) {
    listeners.remove(l);
  }

  private GestureDetector.SimpleOnGestureListener gestureListener=
    new GestureDetector.SimpleOnGestureListener() {
```

```
    @Override
    public boolean onSingleTapUp(MotionEvent e) {
      for (TapListener l : listeners) {
        l.onTap(e);
      }

      return(true);
    }
  };

  public interface TapListener {
    void onTap(MotionEvent event);
  }
}
```

This crude touch interface watches for single taps on the screen, relaying those to a roster of supplied "tap listeners", which will do something on those taps.

The `Player` activity registers an `onTap` listener that displays either the top or bottom panel depending on which half of the screen the user tapped upon:

```
private TappableSurfaceView.TapListener onTap=
  new TappableSurfaceView.TapListener() {
  public void onTap(MotionEvent event) {
    lastActionTime=SystemClock.elapsedRealtime();

    if (event.getY()<surface.getHeight()/2) {
      topPanel.setVisibility(View.VISIBLE);
    }
    else {
      bottomPanel.setVisibility(View.VISIBLE);
    }
  }
};
```

More coverage of touch interfaces will be added in another chapter in a future edition of this book.

The same once-a-second `postDelayed()` loop also updates our timeline, reflecting how much of the video has been played back:

```
private Runnable onEverySecond=new Runnable() {
  public void run() {
    if (lastActionTime>0 &&
        SystemClock.elapsedRealtime()-lastActionTime>3000) {
```

```
      clearPanels(false);
    }

    if (player!=null) {
      timeline.setProgress(player.getCurrentPosition());
    }

    if (!isPaused) {
      surface.postDelayed(onEverySecond, 1000);
    }
  }
};
```

Other Ways to Make Noise

While `MediaPlayer` is the primary audio playback option, particularly for content along the lines of MP3 files, there are other alternatives if you are looking to build other sorts of applications, notably games and custom forms of streaming audio.

SoundPool

The `SoundPool` class's claim to fame is the ability to overlay multiple sounds, and do so in a prioritized fashion, so your application can just ask for sounds to be played and `SoundPool` deals with each sound starting, stopping, and blending while playing.

This may make more sense with an example.

Suppose you are creating a first-person shooter. Such a game may have several sounds going on at any one time:

- The sound of the wind whistling amongst the trees on the battlefield
- The sound of the surf crashing against the beach in the landing zone
- The sound of booted feet crunching on the sand
- The sound of the character's own panting as the character runs on the beach

- The sound of orders being barked by a sergeant positioned behind the character
- The sound of machine gun fire aimed at the character and the character's squad mates
- The sound of explosions from the gun batteries of the battleship providing suppression fire

And so on.

In principle, `SoundPool` can blend all of those together into a single audio stream for output. Your game might set up the wind and surf as constant background sounds, toggle the feet and panting on and off based on the character's movement, randomly add the barked orders, and tie the gunfire based on actual game play.

In reality, your average smartphone will lack the CPU power to handle all of that audio without harming the frame rate of the game. So, to keep the frame rate up, you tell `SoundPool` to play at most two streams at once. This means that when nothing else is happening in the game, you will hear the wind and surf, but during the actual battle, those sounds get dropped out – the user might never even miss them – so the game speed remains good.

AudioTrack

The lowest-level Java API for playing back audio is `AudioTrack`. It has two main roles:

- Its primary role is to support streaming audio, where the streams come in some format other than what `MediaPlayer` handles. While `MediaPlayer` can handle RTSP, for example, it does not offer SIP. If you want to create a SIP client (perhaps for a VOIP or Web conferencing application), you will need to convert the incoming data stream to PCM format, then hand the stream off to an `AudioTrack` instance for playback.
- It can also be used for "static" (versus streamed) bits of sound that you have pre-decoded to PCM format and want to play back with as

little latency as possible. For example, you might use this for a game for in-game sounds (beeps, bullets, or "boing"s). By pre-decoding the data to PCM and caching that result, then using `AudioTrack` for playback, you will use the least amount of overhead, minimizing CPU impact on game play and on battery life.

ToneGenerator

If you want your phone to sound like...well...a phone, you can use `ToneGenerator` to have it play back dual-tone multi-frequency (DTMF) tones. In other words, you can simulate the sounds played by a regular "touch-tone" phone in response to button presses. This is used by the Android dialer, for example, to play back the tones when users dial the phone using the on-screen keypad, as an audio reinforcement.

Note that these will play through the phone's earpiece, speaker, or attached headset. They do not play through the outbound call stream. In principle, you might be able to get `ToneGenerator` to play tones through the speaker loud enough to be picked up by the microphone, but this probably is not a recommended practice.

PART III – Advanced System

CHAPTER 11

The Contacts Content Provider

One of the more popular stores of data on your average Android device is the contact list. This is particularly true with Android 2.0 and newer versions, which track contacts across multiple different "accounts", or sources of contacts. Some may come from your Google account, while others might come from Exchange or other services.

This chapter will walk you through some of the basics for accessing the contacts on the device. Along the way, we will revisit and expand upon our knowledge of using a `ContentProvider`.

First, we will review the contacts APIs, past and present. We will then demonstrate how you can connect to the contacts engine to let users pick and view contacts...all without your application needing to know much of how contacts work. We will then show how you can query the contacts provider to obtain contacts and some of their details, like email addresses and phone numbers. We wrap by showing how you can invoke a built-in activity to let the user add a new contact, possibly including some data supplied by your application.

Introducing You to Your Contacts

Android makes contacts available to you via a complex `ContentProvider` framework, so you can access many facets of a contact's data – not just their name, but addresses, phone numbers, groups, etc. Working with the

contacts `ContentProvider` set is simple...only if you have an established pattern to work with. Otherwise, it may prove somewhat daunting.

ContentProvider Recap

As you may recall from *The Busy Coder's Guide to Android Development* (or other Android programming books), a `ContentProvider` is an abstraction around a data source. Consumers of a `ContentProvider` can use a `ContentResolver` to query, insert, update, or delete data, or use `managedQuery()` on an `Activity` to do a query. In the latter case, the resulting `Cursor` is managed, meaning that it will be deactivated when the activity is stopped, requeried when the activity is later restarted, and closed when the activity is destroyed.

Content providers use a "projection" to describe the columns to work with. One `ContentProvider` may expose many facets of data, which you can think of as being tables. However, bear in mind that content providers do not necessarily have to store their content in SQLite, so you will need to consult the documentation for the content provider to determine query language syntax, transaction support, and the like.

Organizational Structure

The contacts `ContentProvider` framework can be found as the set of `ContactsContract` classes and interfaces in the `android.provider` package. Unfortunately, there is a dizzying array of inner classes to `ContactsContract`.

Contacts can be broken down into two types: raw and aggregate. Raw contacts come from a sync provider or are hand-entered by a user. Aggregate contacts represent the sum of information about an individual culled from various raw contacts. For example, if your Exchange sync provider has a contact with an email address of `jdoe@foo.com`, and your Facebook sync provider has a contact with an email address of `jdoe@foo.com`, Android may recognize that those two raw contacts represent the same person and therefore combine those in the aggregate contact for the user.

The classes relating to raw contacts usually have `Raw` somewhere in their name, and these normally would be used only by custom sync providers.

The `ContactsContract.Contacts` and `ContactsContract.Data` classes represent the "entry points" for the `ContentProvider`, allowing you to query and obtain information on a wide range of different pieces of information. What is retrievable from these can be found in the various `ContactsContract.CommonDataKinds` series of classes. We will see examples of these operations later in this chapter.

A Look Back at Android 1.6

Prior to Android 2.0, Android had no contact synchronization built in. As a result, all contacts were in one large pool, whether they were hand-entered by users or were added via third-party applications. The API used for this is the `Contacts ContentProvider`.

In principle, the `Contacts ContentProvider` should still work, as it is merely deprecated in Android 2.0.1, not removed. In practice, you may encounter some issues, since the emulator may not have the same roster of synchronization providers as does a device, and so there may be differences in behavior.

Pick a Peck of Pickled People

Let's start by finding a contact. After all, that's what the contacts system is for.

Contacts, like anything stored in a `ContentProvider`, is identified by a `Uri`. Hence, we need a `Uri` we can use in the short term, perhaps to read some data, or perhaps just to open up the contact detail activity for the user.

We could ask for a raw contact, or we could ask for an aggregate contact. Since most consumers of the contacts `ContentProvider` will want the aggregate contact, we will use that.

For example, take a look at `Contacts/Pick` in the sample applications, as this shows how to pick a contact from a collection of contacts, then display the contact detail activity. This application gives you a really big "Gimme!" button, which when clicked will launch the contact-selection logic:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pick"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:text="Gimme a contact!"
  android:layout_weight="1"
/>
```

Our first step is to determine the `Uri` to use to reference the collection of contacts we want to pick from. In the long term, there should be just one answer for aggregate contacts: `android.provider.ContactsContract.Contacts.CONTENT_URI`. However, that only works for Android 2.0 (SDK level 5) and higher. On older versions of Android, we need to stick with the original `android.provider.ContactsContract.Contacts.CONTENT_URI`. To accomplish this, we will use a pinch of reflection to determine our `Uri` via a static initializer when our activity starts:

```
private static Uri CONTENT_URI=null;

static {
  int sdk=new Integer(Build.VERSION.SDK).intValue();

  if (sdk>=5) {
    try {
      Class clazz=Class.forName("android.provider.ContactsContract$Contacts");

      CONTENT_URI=(Uri)clazz.getField("CONTENT_URI").get(clazz);
    }
    catch (Throwable t) {
      Log.e("PickDemo", "Exception when determining CONTENT_URI", t);
    }
  }
  else {
    CONTENT_URI=Contacts.People.CONTENT_URI;
  }
}
```

Then, you need to create an `Intent` for the `ACTION_PICK` on the chosen `Uri`, then start a sub activity (via `startActivityForResult()`) to allow the user to pick a piece of content of the specified type:

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

  if (CONTENT_URI==null) {
    Toast
      .makeText(this, "We are experiencing technical difficulties...",
                Toast.LENGTH_LONG)
      .show();
    finish();

    return;
  }

  setContentView(R.layout.main);

  Button btn=(Button)findViewById(R.id.pick);

  btn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View view) {
      Intent i=new Intent(Intent.ACTION_PICK, CONTENT_URI);

      startActivityForResult(i, PICK_REQUEST);
    }
  });
}
```

When that sub-activity completes with `RESULT_OK`, the `ACTION_VIEW` is invoked on the resulting contact `Uri`, as obtained from the `Intent` returned by the pick activity:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==PICK_REQUEST) {
    if (resultCode==RESULT_OK) {
        startActivity(new Intent(Intent.ACTION_VIEW,
                                 data.getData()));
    }
  }
}
```

The result: the user chooses a collection, picks a piece of content, and views it.

Figure 50. The PickDemo sample application, as initially launched

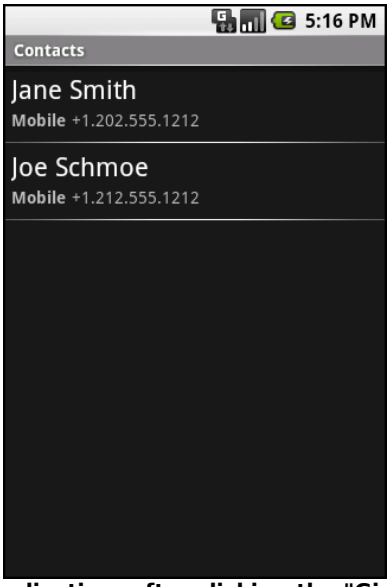


Figure 51. The same application, after clicking the "Gimme!" button, showing the list of available people

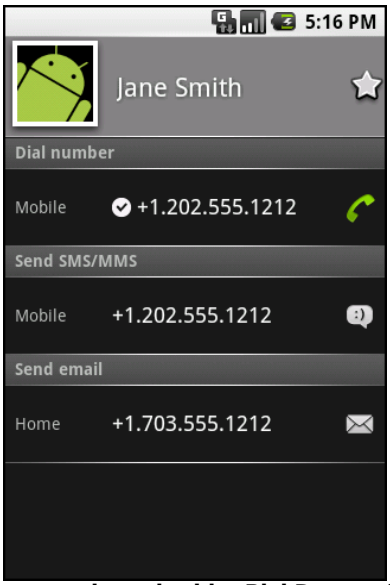


Figure 52. A view of a contact, launched by PickDemo after choosing one of the people from the pick list

Note that the `Uri` we get from picking the contact is valid in the short term, but should not be held onto in a persistent fashion (e.g., put in a database). If you need to try to store a reference to a contact for the long term, you will need to get a "lookup `Uri`" on it, to help deal with the fact that the aggregate contact may shift over time as raw contact information for that person comes and goes.

Spin Through Your Contacts

The preceding example allows you to work with contacts, yet not actually have any contact data other than a transient `Uri`. All else being equal, it is best to use the contacts system this way, as it means you do not need any extra permissions that might raise privacy issues.

Of course, all else is rarely equal.

Your alternative, therefore, is to execute queries against the contacts `ContentProvider` to get actual contact detail data back, such as names,

phone numbers, and email addresses. The `Contacts/Spinners` sample application will demonstrate this technique.

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 `ContactsContract` content and `WRITE_CONTACTS` to add, modify, or remove contacts from the system.

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

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.contacts.spinners"
      android:versionCode="1"
      android:versionName="1.0">
  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-sdk
        android:minSdkVersion="3"
        android:targetSdkVersion="6"
    />
    <application android:label="@string/app_name">
        <activity android:name=".ContactSpinners"
                  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 `ContactsContract` 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, you can query for phone numbers and get the contact name and other data along with the number – you do not have to do this 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.contacts.spinners;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.Spinner;

public class ContactSpinners 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]=ContactsAdapterBridge.INSTANCE.buildNameAdapter(this);
    listAdapters[1]=ContactsAdapterBridge.INSTANCE.buildPhonesAdapter(this);
    listAdapters[2]=ContactsAdapterBridge.INSTANCE.buildEmailAdapter(this);
  }

}
```

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.

Dealing with API Versions

Of course, once again, we have to ponder different API levels.

Querying `ContactsContract` and querying `Contacts` is similar, yet different, both in terms of the `Uri` each uses for the query and in terms of the available column names for the resulting projection.

Rather than using reflection, this time we ruthlessly exploit a feature of the VM: classes are only loaded when first referenced. Hence, we can have a class that refers to new APIs (`ContactsContract`) on a device that lacks those APIs, so long as we do not reference that class.

To accomplish this, we define an abstract base class, `ContactsAdapterBridge`, that will have a singleton instance capable of running our queries and building a `ListAdapter` for each. Then, we create two concrete subclasses, one for the old API:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

class OldContactsAdapterBridge extends ContactsAdapterBridge {
  ListAdapter buildNameAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.People._ID,
                                        Contacts.PeopleColumns.NAME
                                     };
    Cursor c=a.managedQuery(Contacts.People.CONTENT_URI,
                            PROJECTION, null, null,
                            Contacts.People.DEFAULT_SORT_ORDER);

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

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

```
                          PROJECTION, null, null,
                          Contacts.Phones.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter(  a,
                                  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
                                  }));
  }

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

    return(new SimpleCursorAdapter(  a,
                                  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
                                  }));
  }
}
```

...and one for the new API:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Email;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  ListAdapter buildNameAdapter(Activity a) {
```

```
  String[] PROJECTION=new String[] {  Contacts._ID,
                                      Contacts.DISPLAY_NAME,
                                    };
  Cursor c=a.managedQuery(Contacts.CONTENT_URI,
                          PROJECTION, null, null, null);

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

ListAdapter buildPhonesAdapter(Activity a) {
  String[] PROJECTION=new String[] {  Contacts._ID,
                                      Contacts.DISPLAY_NAME,
                                      Phone.NUMBER
                                    };
  Cursor c=a.managedQuery(Phone.CONTENT_URI,
                          PROJECTION, null, null, null);

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

ListAdapter buildEmailAdapter(Activity a) {
  String[] PROJECTION=new String[] {  Contacts._ID,
                                      Contacts.DISPLAY_NAME,
                                      Email.DATA
                                    };
  Cursor c=a.managedQuery(Email.CONTENT_URI,
                          PROJECTION, null, null, null);

  return(new SimpleCursorAdapter(  a,
                                 android.R.layout.simple_list_item_2,
                                 c,
                                 new String[] {
                                   Contacts.DISPLAY_NAME,
                                   Email.DATA
                                 },
                                 new int[] {
```

```
                                    android.R.id.text1,
                                    android.R.id.text2
                                  }));
  }
}
```

Our `ContactsAdapterBridge` class then uses the SDK level to determine which of those two classes to use as the singleton:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.os.Build;
import android.widget.ListAdapter;

abstract class ContactsAdapterBridge {
  abstract ListAdapter buildNameAdapter(Activity a);
  abstract ListAdapter buildPhonesAdapter(Activity a);
  abstract ListAdapter buildEmailAdapter(Activity a);

  public static final ContactsAdapterBridge INSTANCE=buildBridge();

  private static ContactsAdapterBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<5) {
      return(new OldContactsAdapterBridge());
    }

    return(new NewContactsAdapterBridge());
  }
}
```

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:

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

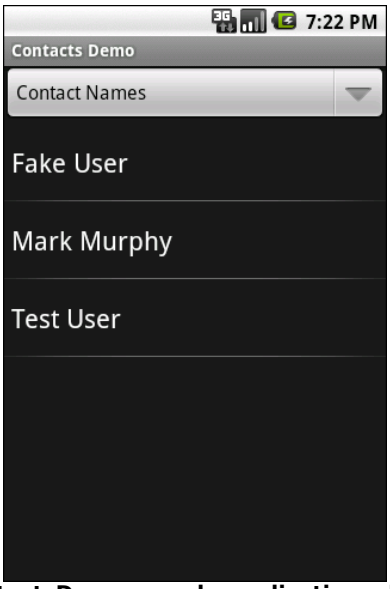


Figure 53. The ContactsDemo sample application, showing all contacts

Accessing Phone Numbers

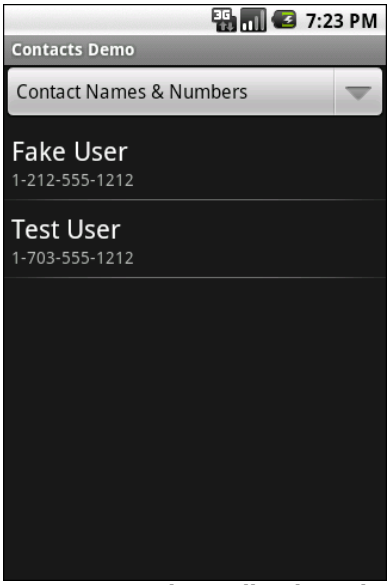


Figure 54. The ContactsDemo sample application, showing all contacts that have phone numbers

Accessing Email Addresses

Similarly, to get a list of all the email addresses, we can use the `CONTENT_URI` content provider. Again, the results are displayed via a two-line `SimpleCursorAdapter`:

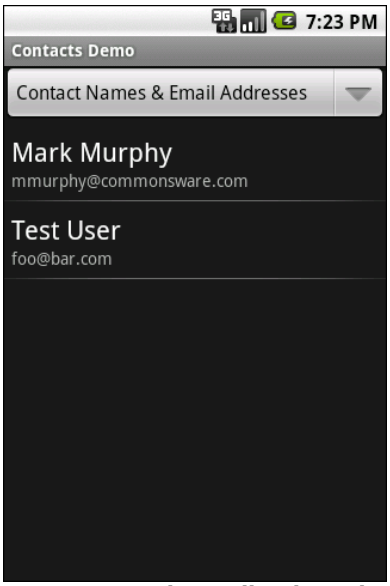


Figure 55. The ContactsDemo sample application, showing all contacts with email addresses

Makin' Contacts

Let's now take a peek at the reverse direction: adding contacts to the system. This was never particularly easy and now is...well, different.

First, we need to distinguish between sync providers and other apps. Sync providers are the guts underpinning the accounts system in Android, bridging some existing source of contact data to the Android device. Hence, you can have sync providers for Exchange, Facebook, and so forth. These will need to create raw contacts for newly-added contacts to their backing stores that are being sync'd to the device for the first time. Creating sync providers is outside of the scope of this book for now.

It is possible for other applications to create contacts. These, by definition, will be phone-only contacts, lacking any associated account, no different than if the user added the contact directly. The recommended approach to doing this is to collect the data you want, then spawn an activity to let the user add the contact – this avoids your application needing the `WRITE_CONTACTS` permission and all the privacy/data integrity issues that creates. In this case, we will stick with the new `ContactsContract` content provider, to simplify our code, at the expense of requiring Android 2.0 or newer.

To that end, take a look at the `Contacts/Inserter` sample project. It defines a simple activity with a two-field UI, with one field apiece for the person's first name and phone number:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1"
  >
  <TableRow>
    <TextView
      android:text="First name:"
    />
    <EditText android:id="@+id/name"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Phone:"
    />
    <EditText android:id="@+id/phone"
      android:inputType="phone"
    />
  </TableRow>
  <Button android:id="@+id/insert" android:text="Insert!" />
</TableLayout>
```

The trivial UI also sports a button to add the contact:

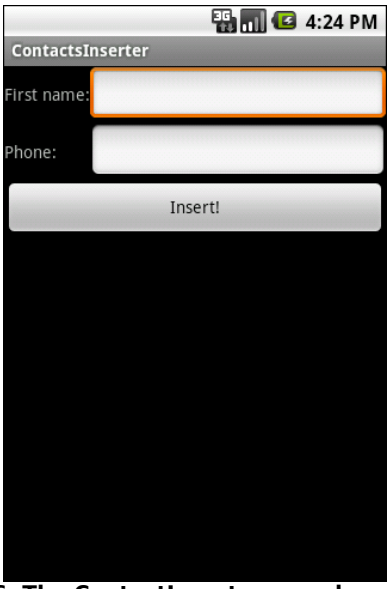


Figure 56. The ContactInserter sample application

When the user clicks the button, the activity gets the data and creates an `Intent` to be used to launch the add-a-contact activity. This uses the `ACTION_INSERT_OR_EDIT` action and a couple of extras from the `ContactsContract.Intents.Insert` class:

```
package com.commonsware.android.inserter;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.Intents.Insert;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

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

    Button btn=(Button)findViewById(R.id.insert);

    btn.setOnClickListener(onInsert);
  }
```

```
  View.OnClickListener onInsert=new View.OnClickListener() {
    public void onClick(View v) {
      EditText fld=(EditText)findViewById(R.id.name);
      String name=fld.getText().toString();

      fld=(EditText)findViewById(R.id.phone);

      String phone=fld.getText().toString();
      Intent i=new Intent(Intent.ACTION_INSERT_OR_EDIT);

      i.setType(Contacts.CONTENT_ITEM_TYPE);
      i.putExtra(Insert.NAME, name);
      i.putExtra(Insert.PHONE, phone);
      startActivity(i);
    }
  };
}
```

We also need to set the MIME type on the `Intent` via `setType()`, to be `CONTENT_ITEM_TYPE`, so Android knows what sort of data we want to actually insert. Then, we call `startActivity()` on the resulting `Intent`. That brings up an add-or-edit activity:

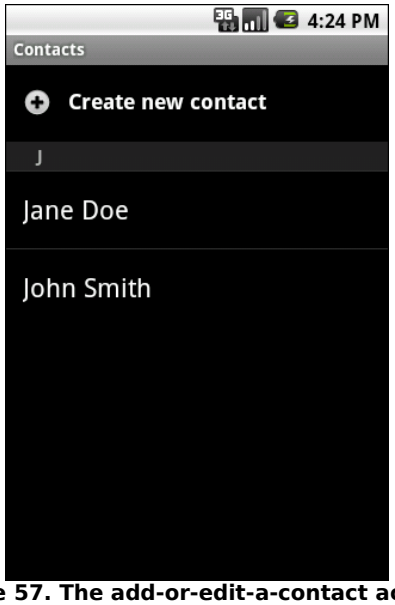


Figure 57. The add-or-edit-a-contact activity

...where if the user chooses "Create new contact", they are taken to the ordinary add-a-contact activity, with our data pre-filled in:

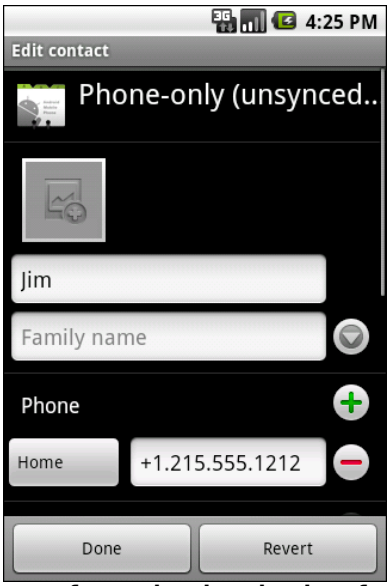


Figure 58. The edit-contact form, showing the data from the ContactInserter activity

Note that the user could choose an existing contact, rather than creating a new contact. If they choose an existing contact, the first name of that contact will be overwritten with the data supplied by the `ContactsInserter` activity, and a new phone number will be added from those `Intent` extras.

CHAPTER 12

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`:

```
mgr=(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

SensorEventListener and respond to changes in the sensor values. To do this, simply call registerListener() with your SensorEventListener and the Sensor you wish to hear from. You can get the Sensor by asking the SensorManager for the default Sensor for a particular type. For example, from the Sensor/Compass sample project, here is where we register our listener:

```
mgr.registerListener(listener,
                     mgr.getDefaultSensor(Sensor.TYPE_ORIENTATION),
                     SensorManager.SENSOR_DELAY_UI);
```

Note that you also specify the rate at which sensor updates will be received. Here, we use SENSOR_DELAY_UI, but you could say SENSOR_DELAY_FASTEST or various other values.

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();
  mgr.unregisterListener(listener);
}
```

Your SensorEventListener 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 SensorEvent containing a float[] of values for the sensor. The tricky part is determining what these sensor values mean.

In the case of TYPE_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 update a `TextView` with each reading:

```
private SensorEventListener listener=new SensorEventListener() {
  public void onSensorChanged(SensorEvent e) {
    if (e.sensor.getType()==Sensor.TYPE_ORIENTATION) {
      degrees.setText(String.valueOf(e.values[0]));
    }
  }

  public void onAccuracyChanged(Sensor 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.

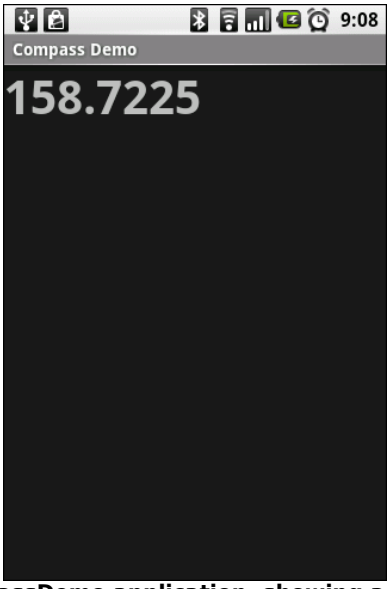


Figure 59. The CompassDemo application, showing a T-Mobile G1 pointing south-by-southeast

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 `TYPE_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 `SensorEventListener` implementation:

```
private SensorEventListener listener=new SensorEventListener() {
  public void onSensorChanged(SensorEvent e) {
    if (e.sensor.getType()==Sensor.TYPE_ORIENTATION) {
      float orientation=e.values[1];

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

        prevOrientation=e.values[1];
      }
    }
  }

  public void onAccuracyChanged(Sensor 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.

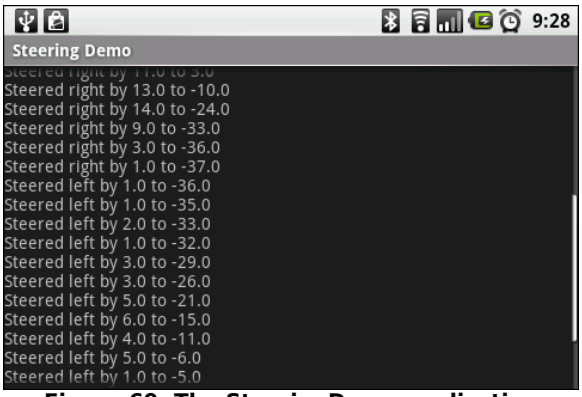


Figure 60. The SteeringDemo application

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 (`TYPE_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.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.SystemClock;
import java.util.ArrayList;
import java.util.List;

public class Shaker {
  private SensorManager mgr=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*threshold;
    this.threshold=this.threshold
                   *SensorManager.GRAVITY_EARTH
                   *SensorManager.GRAVITY_EARTH;
    this.gap=gap;
    this.cb=cb;

    mgr=(SensorManager)ctxt.getSystemService(Context.SENSOR_SERVICE);
    mgr.registerListener(listener,
                         mgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),
                         SensorManager.SENSOR_DELAY_UI);
  }

  public void close() {
    mgr.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 SensorEventListener listener=new SensorEventListener() {
```

```
    public void onSensorChanged(SensorEvent e) {
      if (e.sensor.getType()==Sensor.TYPE_ACCELEROMETER) {
        double netForce=e.values[0]*e.values[0];

        netForce+=e.values[1]*e.values[1];
        netForce+=e.values[2]*e.values[2];

        if (threshold<netForce) {
          isShaking();
        }
        else {
          isNotShaking();
        }
      }
    }

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

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

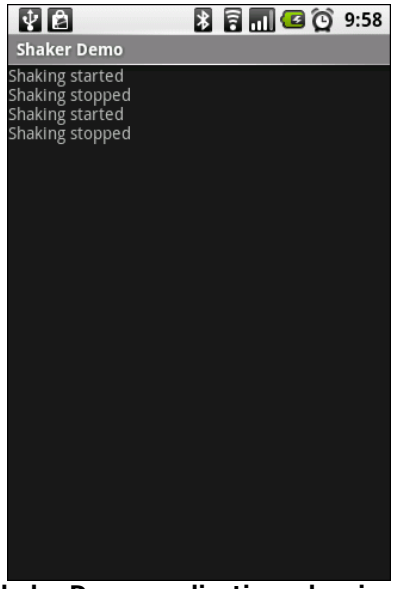


Figure 61. The ShakerDemo application, showing a pair of shakes

CHAPTER 13

Handling System Events

If you have ever looked at the list of available `Intent` actions in the SDK documentation for the `Intent` class, you will see that there are lots of possible actions.

There are even actions that are not listed in that spot in the documentation, but are scattered throughout the rest of the SDK documentation.

The vast majority of these you will never raise yourself. Instead, they are broadcast by Android, to signify certain system events that have occurred and that you might want to take note of, if they affect the operation of your application.

This chapter examines a few of these, to give you the sense of what is possible and how to make use of these sorts of events.

Get Moving, First Thing

A popular request is to have a service get control when the device is powered on.

This is doable but somewhat dangerous, in that too many on-boot requests slow down the device startup and may make things sluggish for the user. Moreover, the more services that are running all the time, the worse the device performance will be.

A better pattern is to get control on boot to arrange for a service to do something periodically using the `AlarmManager` or via other system events. In this section, we will examine the on-boot portion of the problem – in the next chapter, we will investigate `AlarmManager` and how it can keep services active yet not necessarily resident in memory all the time.

The Permission

In order to be notified when the device has completed is system boot process, you will need to request the `RECEIVE_BOOT_COMPLETED` permission. Without this, even if you arrange to receive the boot broadcast Intent, it will not be dispatched to your receiver.

As the Android documentation describes it:

> *Though holding this permission does not have any security implications, it can have a negative impact on the user experience by increasing the amount of time it takes the system to start and allowing applications to have themselves running without the user being aware of them. As such, you must explicitly declare your use of this facility to make that visible to the user.*

The Receiver Element

There are two ways you can receive a broadcast `Intent`. One is to use `registerReceiver()` from an existing `Activity`, `Service`, or `ContentProvider`. The other is to register your interest in the `Intent` in the manifest in the form of a `<receiver>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.sysevents.boot"
      android:versionCode="1"
      android:versionName="1.0">
 <uses-sdk
      android:minSdkVersion="3"
```

```
      android:targetSdkVersion="6"
  />
  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"
  />
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <application android:label="@string/app_name">
        <receiver android:name=".OnBootReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

The above `AndroidManifest.xml`, from the `SystemEvents/OnBoot` sample project, shows that we have registered a broadcast receiver named `OnBootReceiver`, set to be given control when the `android.intent.action.BOOT_COMPLETED` Intent is broadcast.

In this case, we have no choice but to implement our receiver this way – by the time any of our other components (e.g., an `Activity`) were to get control and be able to call `registerReceiver()`, the `BOOT_COMPLETED Intent` will be long gone.

The Receiver Implementation

Now that we have told Android that we would like to be notified when the boot has completed, and given that we have been granted permission to do so by the user, we now need to actually do something to receive the Intent. This is a simple matter of creating a `BroadcastReceiver`, such as seen in the `OnBootCompleted` implementation shown below:

```
package com.commonsware.android.sysevents.boot;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
  @Override
```

```
  public void onReceive(Context context, Intent intent) {
    Log.d("OnBootReceiver", "Hi, Mom!");
  }
}
```

A `BroadcastReceiver` is not a `Context`, and so it gets passed a suitable `Context` object in `onReceive()` to use for accessing resources and the like. The `onReceive()` method also is passed the `Intent` that caused our `BroadcastReceiver` to be created, in case there are "extras" we need to pull out (none in this case).

In `onReceive()`, we can do whatever we want, subject to some limitations:

1. We are not a `Context`, like an `Activity`, so we cannot modify a UI or anything such as that
2. If we want to do anything significant, it is better to delegate that logic to a service that we start from here (e.g., calling `startService()` on the supplied `Context`) rather than actually doing it here, since `BroadcastReceiver` implementations need to be fast
3. We cannot start any background threads, directly or indirectly, since the `BroadcastReceiver` gets discarded as soon as `onReceive()` returns

In this case, we simply log the fact that we got control. In the next chapter, we will see what else we can do at boot time, to ensure one of our services gets control later on as needed.

To test this, install it on an emulator (or device), shut down the emulator, then restart it.

I Sense a Connection Between Us...

Generally speaking, Android applications do not care what sort of Internet connection is being used – 3G, GPRS, WiFi, lots of trained carrier pigeons, or whatever. So long as there is an Internet connection, the application is happy.

Sometimes, though, you may specifically want WiFi. This would be true if your application is bandwidth-intensive and you want to ensure that, should WiFi stop being available, you cut back on your work so as not to consume too much 3G/GPRS bandwidth, which is usually subject to some sort of cap or metering.

There is an `android.net.wifi.WIFI_STATE_CHANGED Intent` that will be broadcast, as the name suggests, whenever the state of the WiFi connection changes. You can arrange to receive this broadcast and take appropriate steps within your application.

This `Intent` requires no special permission, unlike the `BOOT_COMPLETED` Intent from the previous section. Hence, all you need to do is register a `BroadcastReceiver` for `android.net.wifi.WIFI_STATE_CHANGED`, either via `registerReceiver()`, or via the `<receiver>` element in `AndroidManifest.xml`, such as the one shown below, from the `SystemEvents/OnWiFiChange` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.sysevents.wifi"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:label="@string/app_name">
        <receiver android:name=".OnWiFiChangeReceiver">
            <intent-filter>
                <action android:name="android.net.wifi.WIFI_STATE_CHANGED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

All we do in the manifest is tell Android to create an `OnWiFiChangeReceiver` object when a `android.net.wifi.WIFI_STATE_CHANGED Intent` is broadcast, so the receiver can do something useful.

In the case of `OnWiFiChangeReceiver`, it examines the value of the `EXTRA_WIFI_STATE` "extra" in the supplied `Intent` and logs an appropriate message:

```
package com.commonsware.android.sysevents.wifi;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.wifi.WifiManager;
import android.util.Log;

public class OnWiFiChangeReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    int state=intent.getIntExtra(WifiManager.EXTRA_WIFI_STATE, -1);
    String msg=null;

    switch (state) {
      case WifiManager.WIFI_STATE_DISABLED:
        msg="is disabled";
        break;

      case WifiManager.WIFI_STATE_DISABLING:
        msg="is disabling";
        break;

      case WifiManager.WIFI_STATE_ENABLED:
        msg="is enabled";
        break;

      case WifiManager.WIFI_STATE_ENABLING:
        msg="is enabling";
        break;

      case WifiManager.WIFI_STATE_UNKNOWN :
        msg="has an error";
        break;

      default:
        msg="is acting strangely";
        break;
    }

    if (msg!=null) {
      Log.d("OnWiFiChanged", "WiFi "+msg);
    }
  }
}
```

The `EXTRA_WIFI_STATE` "extra" tells you what the state has become (e.g., we are now disabling or are now disabled), so you can take appropriate steps in your application.

Note that, to test this, you will need an actual Android device, as the emulator does not specifically support simulating WiFi connections.

Feeling Drained

One theme with system events is to use them to help make your users happier by reducing your impacts on the device while the device is not in a great state. In the preceding section, we saw how you could find out when WiFi was disabled, so you might not use as much bandwidth when on 3G/GPRS. However, not every application uses so much bandwidth as to make this optimization worthwhile.

However, most applications are impacted by battery life. Dead batteries run no apps.

So whether you are implementing a battery monitor or simply want to discontinue background operations when the battery gets low, you may wish to find out how the battery is doing.

There is an `ACTION_BATTERY_CHANGED` `Intent` that gets broadcast as the battery status changes, both in terms of charge (e.g., 80% charged) and charging (e.g., the device is now plugged into AC power). You simply need to register to receive this `Intent` when it is broadcast, then take appropriate steps.

One of the limitations of `ACTION_BATTERY_CHANGED` is that you have to use `registerReceiver()` to set up a `BroadcastReceiver` to get this `Intent` when broadcast. You cannot use a manifest-declared receiver as shown in the preceding two sections.

In `SystemEvents/OnBattery`, you will find a layout containing a `ProgressBar`, a `TextView`, and an `ImageView`, to serve as a battery monitor:

```
<?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"
```

```
  >
  <ProgressBar android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >
    <TextView android:id="@+id/level"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:textSize="16pt"
    />
    <ImageView android:id="@+id/status"
      android:layout_width="0px"
      android:layout_height="wrap_content"
      android:layout_weight="1"
    />
  </LinearLayout>
</LinearLayout>
```

This layout is used by a `BatteryMonitor` activity, which registers to receive the `ACTION_BATTERY_CHANGED` `Intent` in `onResume()` and unregisters in `onPause()`:

```
package com.commonsware.android.sysevents.battery;

import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.os.BatteryManager;
import android.widget.ProgressBar;
import android.widget.ImageView;
import android.widget.TextView;

public class BatteryMonitor extends Activity {
  private ProgressBar bar=null;
  private ImageView status=null;
  private TextView level=null;

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

    bar=(ProgressBar)findViewById(R.id.bar);
```

```
    status=(ImageView)findViewById(R.id.status);
    level=(TextView)findViewById(R.id.level);
  }

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

    registerReceiver(onBatteryChanged,
                     new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
  }

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

    unregisterReceiver(onBatteryChanged);
  }

  BroadcastReceiver onBatteryChanged=new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
      int pct=100*intent.getIntExtra("level", 1)/intent.getIntExtra("scale", 1);

      bar.setProgress(pct);
      level.setText(String.valueOf(pct));

      switch(intent.getIntExtra("status", -1)) {
        case BatteryManager.BATTERY_STATUS_CHARGING:
          status.setImageResource(R.drawable.charging);
          break;

        case BatteryManager.BATTERY_STATUS_FULL:
          int plugged=intent.getIntExtra("plugged", -1);

          if (plugged==BatteryManager.BATTERY_PLUGGED_AC ||
              plugged==BatteryManager.BATTERY_PLUGGED_USB) {
            status.setImageResource(R.drawable.full);
          }
          else {
            status.setImageResource(R.drawable.unplugged);
          }
          break;

        default:
          status.setImageResource(R.drawable.unplugged);
          break;
      }
    }
  };
}
```

The key to ACTION_BATTERY_CHANGED is in the "extras". Many "extras" are packaged in the Intent, to describe the current state of the battery, such as the following constants defined on the BatteryManager class:

- EXTRA_HEALTH, which should generally be BATTERY_HEALTH_GOOD
- EXTRA_LEVEL, which is the proportion of battery life remaining as an integer, specified on the scale described by the scale "extra"
- EXTRA_PLUGGED, which will indicate if the device is plugged into AC power (BATTERY_PLUGGED_AC) or USB power (BATTERY_PLUGGED_USB)
- EXTRA_SCALE, which indicates the maximum possible value of level (e.g., 100, indicating that level is a percentage of charge remaining)
- EXTRA_STATUS, which will tell you if the battery is charging (BATTERY_STATUS_CHARGING), full (BATTERY_STATUS_FULL), or discharging (BATTERY_STATUS_DISCHARGING)
- EXTRA_TECHNOLOGY, which indicates what sort of battery is installed (e.g., "Li-Ion")
- EXTRA_TEMPERATURE, which tells you how warm the battery is, in tenths of a degree Celsius (e.g., 213 is 21.3 degrees Celsius)
- EXTRA_VOLTAGE, indicating the current voltage being delivered by the battery, in millivolts

In the case of BatteryMonitor, when we receive an ACTION_BATTERY_CHANGED Intent, we do three things:

1. We compute the percentage of battery life remaining, by dividing the level by the scale
2. We update the ProgressBar and TextView to display the battery life as a percentage
3. We display an icon, with the icon selection depending on whether we are charging (status is BATTERY_STATUS_CHARGING), full but on the charger (status is BATTERY_STATUS_FULL and plugged is BATTERY_PLUGGED_AC or BATTERY_PLUGGED_USB), or are not plugged in

This only really works on a device, where you can plug and unplug it, plus get a varying charge level:

Figure 62. The BatteryMonitor application

Sticky Intents and the Battery

Android has a notion of "sticky broadcast `Intents`". Normally, a broadcast `Intent` will be delivered to interested parties and then discarded. A sticky broadcast `Intent` is delivered to interested parties and retained until the next matching `Intent` is broadcast. Applications can call `registerReceiver()` with an `IntentFilter` that matches the sticky broadcast, but with a `null` `BroadcastReceiver`, and get the sticky `Intent` back as a result of the `registerReceiver()` call.

This may sound confusing. Let's look at this in the context of the battery.

Earlier in this section, you saw how to register for `ACTION_BATTERY_CHANGED` to get information about the battery delivered to you. You can also, though, get the latest battery information without registering a receiver. Just create an `IntentFilter` to match `ACTION_BATTERY_CHANGED` (as shown above) and call `registerReceiver()` with that filter and a `null` `BroadcastReceiver`. The `Intent` you get back from `registerReceiver()` is the last `ACTION_BATTERY_CHANGED` `Intent` that was broadcast, with the same extras. Hence, you can use this to

get the current (or near-current) battery status, rather than having to bother registering an actual `BroadcastReceiver`.

Other Power Triggers

If you are only interested in knowing when the device has been attached to, or detached from, a source of external power, there are different broadcast Intent actions you can monitor: `ACTION_POWER_CONNECTED` and `ACTION_POWER_DISCONNECTED`. These are only broadcast when the power source changes, not just every time the battery changes charge level. Hence, these will be more efficient, as your code will be invoked less frequently. Better still, you can use manifest-registered broadcast receivers for these, bypassing the limits the system puts on `ACTION_BATTERY_CHANGED`.

CHAPTER 14

Using System Services

Android offers a number of system services, usually obtained by `getSystemService()` from your `Activity`, `Service`, or other `Context`. These are your gateway to all sorts of capabilities, from settings to volume to WiFi. Throughout the course of this book and its companion, we have seen several of these system services. In this chapter, we will take a look at others that may be of value to you in building compelling Android applications.

Get Alarmed

A common question when doing Android development is "where do I set up `cron` jobs?"

The `cron` utility – popular in Linux – is a way of scheduling work to be done periodically. You teach `cron` what to run and when to run it (e.g., weekdays at noon), and `cron` takes care of the rest. Since Android has a Linux kernel at its heart, one might think that `cron` might literally be available.

While `cron` itself is not, Android does have a system service named `AlarmManager` which fills a similar role. You give it a `PendingIntent` and a time (and optionally a period for repeating) and it will fire off the `Intent` as needed. By this mechanism, you can get a similar effect to `cron`.

There is one small catch, though: Android is designed to run on mobile devices, particularly ones powered by all-too-tiny batteries. If you want

your periodic tasks to be run even if the device is "asleep", you will need to take a fair number of extra steps, mostly stemming around the concept of the `WakeLock`.

Concept of WakeLocks

Most of the time in Android, you are developing code that will run while the user is actually using the device. Activities, for example, only really make sense when the device is fully awake and the user is tapping on the screen or keyboard.

Particularly with scheduled background tasks, though, you need to bear in mind that the device will eventually "go to sleep". In full sleep mode, the display, main CPU, and keyboard are all powered off, to maximize battery life. Only on a low-level system event, like an incoming phone call, will anything wake up.

Another thing that will partially wake up the phone is an `Intent` raised by the `AlarmManager`. So long as broadcast receivers are processing that `Intent`, the `AlarmManager` ensures the CPU will be running (though the screen and keyboard are still off). Once the broadcast receivers are done, the `AlarmManager` lets the device go back to sleep.

You can achieve the same effect in your code via a `WakeLock`, obtained via the `PowerManager` system service. When you acquire a "partial `WakeLock`" (`PARTIAL_WAKE_LOCK`), you prevent the CPU from going back to sleep until you release said `WakeLock`. By proper use of a partial `WakeLock`, you can ensure the CPU will not get shut off while you are trying to do background work, while still allowing the device to sleep most of the time, in between alarm events.

However, using a `WakeLock` is a bit tricky, particularly when responding to an alarm `Intent`, as we will see in the next few sections. The good news is that CommonsWare has packaged up a pattern for dealing with this situation – an alarm triggering work that needs to keep the device awake – in a component called the `WakefulIntentService`.

The WakeLock Problem

The `AlarmManager` will arrange for the device to stay awake, via a `WakeLock`, for as long as the `BroadcastReceiver`'s `onReceive()` method is executing. For some situations, that may be all that is needed. However, `onReceive()` is called on the main application thread, and Android will kill off the receiver if it takes too long.

Your natural inclination in this case is to have the `BroadcastReceiver` arrange for a `Service` to do the long-running work on a background thread, since `BroadcastReceiver` objects should not be starting their own threads. Perhaps you would use an `IntentService`, which packages up this "start a `Service` to do some work in the background" pattern. And, given the preceding section, you might try acquiring a partial `WakeLock` at the beginning of the work and release it at the end of the work, so the CPU will keep running while your `IntentService` does its thing.

This strategy will work...some of the time.

The problem is that there is a gap in `WakeLock` coverage, as depicted in the following diagram:

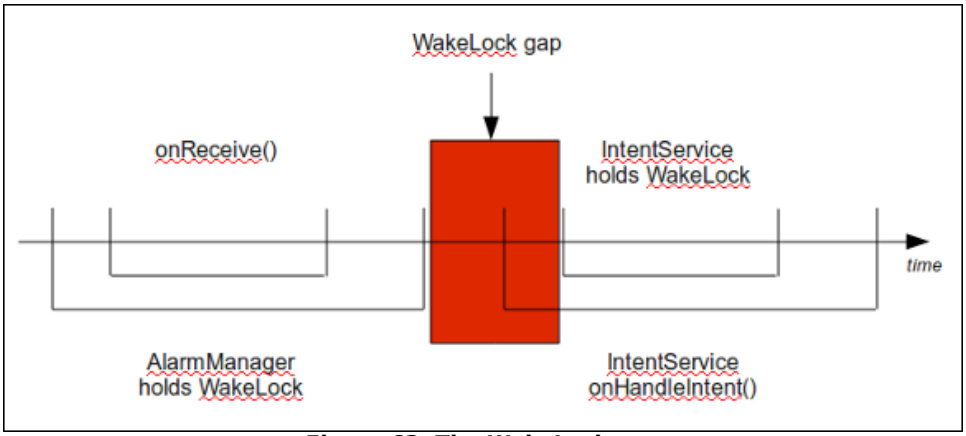


Figure 63. The WakeLock gap

The `BroadcastReceiver` will call `startService()` to send work to the `IntentService`, but that service will not start up until after `onReceive()` ends. As a result, there is a window of time between the end of `onReceive()` and when your `IntentService` can acquire its own `WakeLock`. During that window, the device might fall back asleep. Sometimes it will, sometimes it will not.

What you need to do, instead, is arrange for overlapping `WakeLock` instances. You need to acquire a `WakeLock` in your `BroadcastReceiver`, during the `onReceive()` execution, and hold onto that `WakeLock` until the work is completed by the `IntentService`:

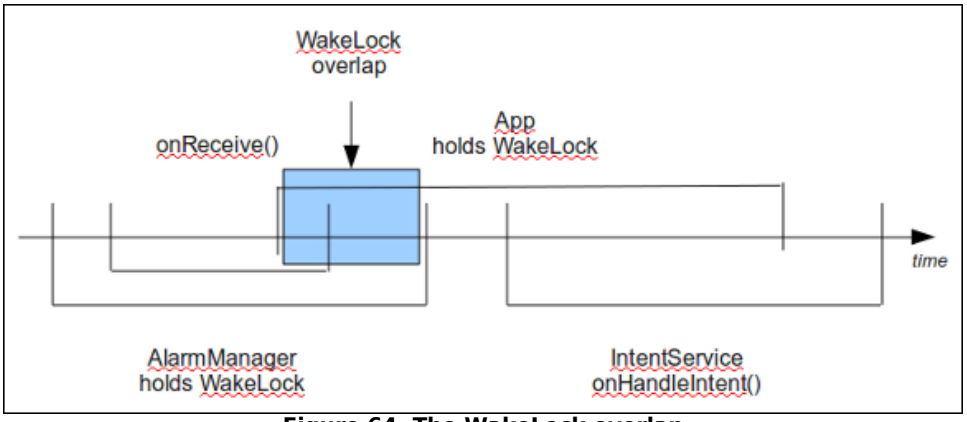


Figure 64. The WakeLock overlap

Then you are assured that the device will stay awake as long as the work remains to be done.

The following sections will show how you can achieve this effect.

Scheduling Alarms

The first step to creating a `cron` workalike is to arrange to get control when the device boots. After all, the `cron` daemon starts on boot as well, and we have no other way of ensuring that our background tasks start firing after a phone is reset.

We saw how to do that in a previous chapter – set up an RECEIVE_BOOT_COMPLETED BroadcastReceiver, with appropriate permissions. Here, for example, is the AndroidManifest.xml from SystemServices/Alarm:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.syssvc.alarm"
      android:versionCode="1"
      android:versionName="1.0">
 <uses-sdk
      android:minSdkVersion="3"
      android:targetSdkVersion="6"
  />
  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"
  />
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
  <uses-permission android:name="android.permission.WAKE_LOCK" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
  <application android:label="@string/app_name">
      <receiver android:name=".OnBootReceiver">
          <intent-filter>
              <action android:name="android.intent.action.BOOT_COMPLETED" />
          </intent-filter>
      </receiver>
      <receiver android:name=".OnAlarmReceiver">
      </receiver>
      <service android:name=".AppService">
      </service>
  </application>
</manifest>
```

We ask for an OnBootReceiver to get control when the device starts up, and it is in OnBootReceiver that we schedule our recurring alarm:

```
package com.commonsware.android.syssvc.alarm;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
  private static final int PERIOD=300000;  // 5 minutes

  @Override
```

```
  public void onReceive(Context context, Intent intent) {
    AlarmManager
mgr=(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(context, OnAlarmReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(context, 0,
                                          i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
                     SystemClock.elapsedRealtime()+60000,
                     PERIOD,
                     pi);
  }
}
```

We get the `AlarmManager` via `getSystemService()`, create an `Intent` referencing another `BroadcastReceiver` (`OnAlarmReceiver`), wrap that `Intent` in a `PendingIntent`, and tell the `AlarmManager` to set up a repeating alarm via `setRepeating()`. By saying we want a `ELAPSED_REALTIME_WAKEUP` alarm, we indicate that we want the alarm to wake up the device (even if it is asleep) and to express all times using the time base used by `SystemClock.elapsedRealtime()`. In this case, our alarm is set to go off every five minutes.

This will cause the `AlarmManager` to raise our `Intent` imminently, and every five minutes thereafter.

Arranging for Work From Alarms

When an alarm goes off, our `OnAlarmReceiver` will get control. It needs to arrange for a service (in this case, named `AppService`) to do its work in the background, but then release control quickly – `onReceive()` cannot take very much time.

Here is the tiny implementation of `OnAlarmReceiver` from `SystemServices/Alarm`:

```
package com.commonsware.android.syssvc.alarm;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
```

```
public class OnAlarmReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    WakefulIntentService.acquireStaticLock(context);

    context.startService(new Intent(context, AppService.class));
  }
}
```

While there is very little code in this class, it is merely deceptively simple.

First, we acquire a `WakeLock` from our `AppService`'s parent class, `WakefulIntentService`, via `acquireStaticLock()`, shown below:

```
private static PowerManager.WakeLock lockStatic=null;

public static void acquireStaticLock(Context context) {
  getLock(context).acquire();
}

synchronized private static PowerManager.WakeLock getLock(Context context) {
  if (lockStatic==null) {
    PowerManager
mgr=(PowerManager)context.getSystemService(Context.POWER_SERVICE);

    lockStatic=mgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                       LOCK_NAME_STATIC);
    lockStatic.setReferenceCounted(true);
  }

  return(lockStatic);
}
```

The `getLock()` implementation lazy-creates our `WakeLock` by getting the `PowerManager`, creating a new partial `WakeLock`, and setting it to be reference counted (meaning if it is acquired several times, it takes a corresponding number of `release()` calls to truly release the lock). If we have already retrieved the `WakeLock` in a previous invocation, we reuse the same lock.

Back in `OnAlarmReceiver`, up until this point, the CPU was running because `AlarmManager` held a partial `WakeLock`. Now, the CPU is running because both `AlarmManager` *and* `WakefulIntentService` hold a partial `WakeLock`.

Then, OnAlarmReceiver starts the AppService instance (remember: acquireStaticLock() was a *static* method) and exits. Notably, OnAlarmReceiver does not release the WakeLock it acquired. This is important, as we need to ensure that the service can get its work done while the CPU is running. Had we released the WakeLock before returning, it is possible that the device would fall back asleep before our service had a chance to acquire a fresh WakeLock. This is one of the keys of using WakeLock successfully – as needed, use overlapping WakeLock instances to ensure constant coverage as you pass from component to component.

Now, our service will start up and be able to do something, while the CPU is running due to our acquired WakeLock.

Staying Awake At Work

So, AppService will now get control, under an active WakeLock. At minimum, our service will be called via onStart(), and possibly also onCreate() if the service had been previously stopped. Our mission is to do our work and release the WakeLock.

Since services should not do long-running tasks in onStart(), we could fork a Thread, have it do the work in the background, then have it release the WakeLock. Note that we cannot release the WakeLock in onStart() in this case – just because we have a background thread does not mean the device will keep the CPU running.

There are issues with forking a Thread for every incoming request, though:

- If the work needed to be done sometimes takes longer than the alarm period, we could wind up with many background threads, which is inefficient. It also means our WakeLock management gets much trickier, since we will not have released the WakeLock before the alarm tries to acquire() it again.
- If we also are invoked in onStart() via some foreground activity, we might wind up with many more bits of work to be done, again

causing confusion with our `WakeLock` and perhaps slowing things down due to too many background threads.

Android has a class that helps with parts of this, `IntentService`. It arranges for a work queue of inbound `Intents` – rather than overriding `onStart()`, you override `onHandleIntent()`, which is called from a background thread. Android handles all the details of shutting down your service when there is no more outstanding work, managing the background thread, and so on.

However, `IntentService` does not do anything to hold a `WakeLock`.

Hence, this sample project uses `WakefulIntentService` as a subclass of `IntentService`. `WakefulIntentService` handles most of the `WakeLock` logic, so `AppService` (inheriting from `WakefulIntentService`) can just focus on the work it needs to do. You can find the `WakefulIntentService` in the CommonsWare set of github repositories, as it is one of the CommonsWare Android Components (CWAC), which we will explore in greater detail in a future edition of this book.

`WakefulIntentService` handles the `WakeLock` logic in two components:

1. It offers the public static method `acquireStaticLock()`, which needs to be called by whoever is calling `startService()` on our `WakefulIntentService` subclass.
2. In `onHandleIntent()`, it releases the static `WakeLock`. Since this `WakeLock` is reference-counted, the lock will only fully release once every `Intent` enqueued by `onStart()` has been handled by `onHandleIntent()`. It requires subclasses to implement `doWakefulWork()` – in there, the subclass can do whatever it might ordinarily have done in `onHandleIntent()`, just with the assurance that the device will stay awake while doing it.

With all that supporting us, `AppService` need only implement `doWakefulWork()` and do its work:

```
package com.commonsware.android.syssvc.alarm;

import android.content.Intent;
```

```
import android.os.Environment;
import android.util.Log;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Date;

public class AppService extends WakefulIntentService {
  public AppService() {
    super("AppService");
  }

  @Override
  protected void doWakefulWork(Intent intent) {
    File log=new File(Environment.getExternalStorageDirectory(),
                      "AlarmLog.txt");

    try {
      BufferedWriter out=new BufferedWriter(
                           new FileWriter(log.getAbsolutePath(),
                                          log.exists()));

      out.write(new Date().toString());
      out.write("\n");
      out.close();
    }
    catch (IOException e) {
      Log.e("AppService", "Exception appending to log file", e);
    }
  }
}
```

The "fake work" being done by this `AppService` is simply logging the fact that work needed to be done to a log file on the SD card.

Note that if you attempt to build and run this project that you will need an SD card in the device (or card image attached to your emulator).

Note that there is another kind of "wake lock": `WifiLock`. As the name suggests, this keeps the WiFi radio on even if the device is idle. Ordinarily, the WiFi radio shuts down after a period of inactivity, to save battery life. If you are downloading large files or otherwise need continuous WiFi access while your application is running, consider a `WifiLock`, but do not overuse it.

Setting Expectations

If you have an Android device, you probably have spent some time in the Settings application, tweaking your device to work how you want – ringtones, WiFi settings, USB debugging, etc. Many of those settings are also available via `Settings` class (in the `android.provider` package), and particularly the `Settings.System` and `Settings.Secure` public inner classes.

Basic Settings

`Settings.System` allows you to get and, with the `WRITE_SETTINGS` permission, alter these settings. As one might expect, there are a series of typed getter and setter methods on `Settings.System`, each taking a key as a parameter. The keys are class constants, such as:

- `INSTALL_NON_MARKET_APPS` to control whether you can install applications on a device from outside of the Android Market
- `LOCK_PATTERN_ENABLED` to control whether the user needs to enter a lock pattern to enable use of the device
- `LOCK_PATTERN_VISIBLE` to control whether the lock pattern is drawn on-screen as it is swiped by the user, or if the swipes are "invisible"

The `SystemServices/Settings` project has a `SettingsSetter` sample application that displays a checklist:

```
<?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"
/>
```

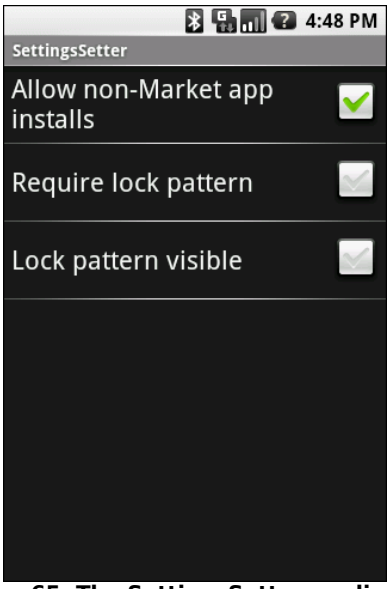


Figure 65. The SettingsSetter application

The checklist itself is filled with a few `BooleanSetting` objects, which map a display name with a `Settings.System` key:

```
static class BooleanSetting {
  String key;
  String displayName;
  boolean isSecure=false;

  BooleanSetting(String key, String displayName) {
    this(key, displayName, false);
  }

  BooleanSetting(String key, String displayName,
                 boolean isSecure) {
    this.key=key;
    this.displayName=displayName;
    this.isSecure=isSecure;
  }

  @Override
  public String toString() {
    return(displayName);
  }

  boolean isChecked(ContentResolver cr) {
    try {
      int value=0;
```

```
      if (isSecure) {
        value=Settings.Secure.getInt(cr, key);
      }
      else {
        value=Settings.System.getInt(cr, key);
      }

      return(value!=0);
    }
    catch (Settings.SettingNotFoundException e) {
      Log.e("SettingsSetter", e.getMessage());
    }

    return(false);
  }

  void setChecked(ContentResolver cr, boolean value) {
    try {
      if (isSecure) {
        Settings.Secure.putInt(cr, key, (value ? 1 : 0));
      }
      else {
        Settings.System.putInt(cr, key, (value ? 1 : 0));
      }
    }
```

Three such settings are put in the list, and as the checkboxes are checked and unchecked, the values are passed along to the settings themselves:

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

  getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
  setListAdapter(new ArrayAdapter(this,
                                  android.R.layout.simple_list_item_multiple_choi
ce,
                                  settings));

  ContentResolver cr=getContentResolver();

  for (int i=0;i<settings.size();i++) {
    BooleanSetting s=settings.get(i);

    getListView().setItemChecked(i, s.isChecked(cr));
  }
}

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

```
  super.onListItemClick(l, v, position, id);

  BooleanSetting s=settings.get(position);

  s.setChecked(getContentResolver(),
               l.isItemChecked(position));
}
```

The `SettingsSetter` activity also has an option menu containing four items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/app"
    android:title="Application"
    android:icon="@drawable/ic_menu_manage" />
  <item android:id="@+id/security"
    android:title="Security"
    android:icon="@drawable/ic_menu_close_clear_cancel" />
  <item android:id="@+id/wireless"
    android:title="Wireless"
    android:icon="@drawable/ic_menu_set_as" />
  <item android:id="@+id/all"
    android:title="All Settings"
    android:icon="@drawable/ic_menu_preferences" />
</menu>
```

These items correspond to four activity `Intent` values identified by the `Settings` class:

```
menuActivities.put(R.id.app,
                   Settings.ACTION_APPLICATION_SETTINGS);
menuActivities.put(R.id.security,
                   Settings.ACTION_SECURITY_SETTINGS);
menuActivities.put(R.id.wireless,
                   Settings.ACTION_WIRELESS_SETTINGS);
menuActivities.put(R.id.all,
                   Settings.ACTION_SETTINGS);
```

When an option menu is chosen, the corresponding activity is launched:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  String activity=menuActivities.get(item.getItemId());

  if (activity!=null) {
    startActivity(new Intent(activity));

    return(true);
  }
```

```
  return(super.onOptionsItemSelected(item));
}
```

This way, you have your choice of either directly manipulating the settings or merely making it easier for users to get to the Android-supplied activity for manipulating those settings.

Secure Settings

You will notice that if you use the above code and try changing the value of "Allow non-Market app installs", the application blows up.

Once upon a time – Android 1.1 and earlier – you could modify this setting.

Now, though, that setting is one that Android deems "secure". The constant has been moved from `Settings.System` to `Settings.Secure`, though the old constant is still there, flagged as deprecated.

These so-called "secure" settings are ones that Android does not allow applications to change. No permission resolves this problem. The only option is to display the official Settings activity and let the user change the setting.

Can You Hear Me Now? OK, How About Now?

The fancier the device, the more complicated controlling sound volume becomes.

On a simple MP3 player, there is usually only one volume control. That is because there is only one source of sound: the music itself, played through speakers or headphones.

In Android, though, there are several sources of sounds:

- Ringing, to signify an incoming call

- Voice calls
- Alarms, such as those raised by the Alarm Clock application
- System sounds (error beeps, USB connection signal, etc.)
- Music, as might come from the MP3 player

Android allows the user to configure each of these volume levels separately. Usually, the user does this via the volume rocker buttons on the device, in the context of whatever sound is being played (e.g., when on a call, the volume buttons change the voice call volume). Also, there is a screen in the Android Settings application that allows you to configure various volume levels.

The `AudioService` in Android allows you, the developer, to also control these volume levels, for all five "streams" (i.e., sources of sound). In the `SystemServices/Volume` project, we create a `Volumizer` application that displays and modifies all five volume levels.

Attaching SeekBars to Volume Streams

The standard widget for allowing choice along a range of integer values is the `SeekBar`, a close cousin of the `ProgressBar`. `SeekBar` has a thumb that the user can slide to choose a value between `0` and some maximum that you set. So, we will use a set of five `SeekBar` widgets to control our five volume levels.

First, we need to create a layout with a `SeekBar` per stream:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/com.commonsware.android.syssvc.v
olume"
  android:stretchColumns="1"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
>
  <TableRow
    android:paddingTop="10px"
    android:paddingBottom="20px">
    <TextView android:text="Alarm:" />
    <SeekBar
```

```
      android:id="@+id/alarm"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="Music:" />
    <SeekBar
      android:id="@+id/music"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="Ring:" />
    <SeekBar
      android:id="@+id/ring"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="System:" />
    <SeekBar
      android:id="@+id/system"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
  <TableRow>
    <TextView android:text="Voice:" />
    <SeekBar
      android:id="@+id/voice"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
</TableLayout>
```

Then, we need to wire up each of those bars in the `onCreate()` for `Volumizer`, calling an `initBar()` method for each of the five bars:

```
public class Volumizer extends Activity {
  SeekBar alarm=null;
  SeekBar music=null;
  SeekBar ring=null;
  SeekBar system=null;
  SeekBar voice=null;
  AudioManager mgr=null;
```

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

    mgr=(AudioManager)getSystemService(Context.AUDIO_SERVICE);

    alarm=(SeekBar)findViewById(R.id.alarm);
    music=(SeekBar)findViewById(R.id.music);
    ring=(SeekBar)findViewById(R.id.ring);
    system=(SeekBar)findViewById(R.id.system);
    voice=(SeekBar)findViewById(R.id.voice);

    initBar(alarm, AudioManager.STREAM_ALARM);
    initBar(music, AudioManager.STREAM_MUSIC);
    initBar(ring, AudioManager.STREAM_RING);
    initBar(system, AudioManager.STREAM_SYSTEM);
    initBar(voice, AudioManager.STREAM_VOICE_CALL);
  }

  private void initBar(SeekBar bar, final int stream) {
    bar.setMax(mgr.getStreamMaxVolume(stream));
    bar.setProgress(mgr.getStreamVolume(stream));

    bar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
```

In `initBar()`, we set the appropriate size for the `Meter` bar via `setMax()`, set the initial value via `setProgress()`, and hook up an `OnSeekBarChangeListener` to find out when the user slides the bar, so we can set the volume on the stream via the `VolumeManager`:

```
                                boolean fromUser) {
        mgr.setStreamVolume(stream,  progress,
                            AudioManager.FLAG_PLAY_SOUND);
      }

      public void onStartTrackingTouch(SeekBar bar) {
        // no-op
      }

      public void onStopTrackingTouch(SeekBar bar) {
        // no-op
      }
    });
  }
}
```

The net result is that when the user slides a `SeekBar`, it adjusts the stream to match:

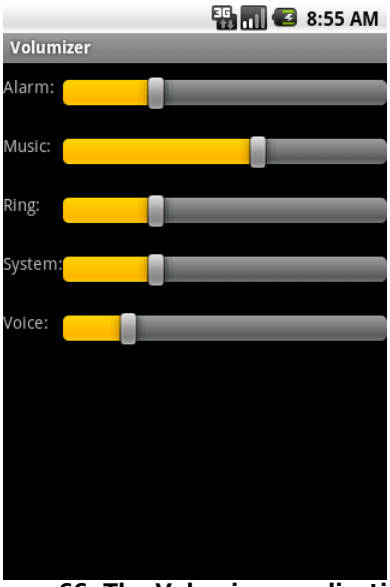


Figure 66. The Volumizer application

CHAPTER 15

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.

We start with an explanation of the inter-process communication (IPC) mechanism offered in Android for allowing services to work with clients in other applications. Then, we move onto the steps to allow a client to connect to a remote service, before describing how to turn an ordinary service into a remote one. We wrap by looking at how one can implement a callback system to allow services, through IPC, to pass information back to clients.

When IPC Attacks!

Services will tend to offer IPC as a means of interacting with activities or other Android components. Each service declares what methods it is making available over IPC; those methods are then available for other components to call, with Android handling all the messy details involved with making method calls across component or process boundaries.

The guts of this, from the standpoint of the developer, is expressed in AIDL: the Android Interface Description Language. If you have used IPC mechanisms like COM, CORBA, or the like, you will recognize the notion of

IDL. AIDL describes the public IPC interface, and Android supplies tools to build the client and server side of that interface.

With that in mind, let's take a look at AIDL and IPC.

Write the AIDL

IDLs are frequently written in a "language-neutral" syntax. AIDL, on the other hand, looks a lot like a Java interface. For example, here is some AIDL:

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

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

As with a Java interface, you declare a package at the top. As with a Java interface, the methods are wrapped in an interface declaration (`interface IScript { ... }`). And, as with a Java interface, you list the methods you are making available.

The differences, though, are critical.

First, not every Java type can be used as a parameter. Your choices are:

- Primitive values (`int`, `float`, `double`, `boolean`, etc.)
- `String` and `CharSequence`
- `List` and `Map` (from `java.util`)
- Any other AIDL-defined interfaces
- Any Java classes that implement the `Parcelable` interface, which is Android's flavor of serialization

In the case of the latter two categories, you need to include `import` statements referencing the names of the classes or interfaces that you are

using (e.g., `import com.commonsware.android.ISomething`). This is true even if these classes are in your own package – you have to import them anyway.

Next, parameters can be classified as `in`, `out`, or `inout`. Values that are `out` or `inout` can be changed by the service and those changes will be propagated back to the client. Primitives (e.g., `int`) can only be `in`; we included in for the AIDL for `enable()` just for illustration purposes.

Also, you cannot throw any exceptions. You will need to catch all exceptions in your code, deal with them, and return failure indications some other way (e.g., error code return values).

Name your AIDL files with the `.aidl` extension and place them in the proper directory based on the package name.

When you build your project, either via an IDE or via Ant, the `aidl` utility from the Android SDK will translate your AIDL into a server stub and a client proxy.

Implement the Interface

Given the AIDL-created server stub, now you need to implement the service, either directly in the stub, or by routing the stub implementation to other methods you have already written.

The mechanics of this are fairly straightforward:

- Create a private instance of the AIDL-generated `.Stub` class (e.g., `IScript.Stub`)
- Implement methods matching up with each of the methods you placed in the AIDL
- Return this private instance from your `onBind()` method in the `Service` subclass

Note that AIDL IPC calls are synchronous, and so the caller is blocked until the IPC method returns. Hence, your services need to be quick about their work.

We will see examples of service stubs later in this chapter.

A Consumer Economy

Of course, we need to have a client for AIDL-defined services, lest these services feel lonely.

Bound for Success

To use an AIDL-defined service, you first need to create an instance of your own `ServiceConnection` class. `ServiceConnection`, as the name suggests, represents your connection to the service for the purposes of making IPC calls.

Your `ServiceConnection` subclass needs to implement two methods:

1. `onServiceConnected()`, which is called once your activity is bound to the service
2. `onServiceDisconnected()`, which is called if your connection ends normally, such as you unbinding your activity from the service

Each of those methods receives a `ComponentName`, which simply identifies the service you connected to. More importantly, `onServiceConnected()` receives an `IBinder` instance, which is your gateway to the IPC interface. You will want to convert the `IBinder` into an instance of your AIDL interface class, so you can use IPC as if you were calling regular methods on a regular Java class (`IScript.Stub.asInterface(binder)`).

To actually hook your activity to the service, call `bindService()` on the activity:

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

The `bindService()` method takes three parameters:

1. An `Intent` representing the service you wish to invoke
2. Your `ServiceConnection` instance
3. A set of flags – most times, you will want to pass in `BIND_AUTO_CREATE`, which will start up the service if it is not already running

After your `bindService()` call, your `onServiceConnected()` callback in the `ServiceConnection` will eventually be invoked, at which time your connection is ready for use.

Request for Service

Once your service interface object is ready (`IScript.Stub.asInterface(binder)`), you can start calling methods on it as you need to. In fact, if you disabled some widgets awaiting the connection, now is a fine time to re-enable them.

However, you will want to trap two exceptions. One is `DeadObjectException` – if this is raised, your service connection terminated unexpectedly. In this case, you should unwind your use of the service, perhaps by calling `onServiceDisconnected()` manually, as shown above. The other is `RemoteException`, which is a more general-purpose exception indicating a cross-process communications problem. Again, you should probably cease your use of the service.

Getting Unbound

When you are done with the IPC interface, call `unbindService()`, passing in the `ServiceConnection`. Eventually, your connection's `onServiceDisconnected()` callback will be invoked, at which point you should

null out your interface object, disable relevant widgets, or otherwise flag yourself as no longer being able to use the service.

You can always reconnect to the service, via `bindService()`, if you need to use it again.

Service From Afar

Everything from the preceding two sections could be used by local services. In fact, that prose originally appeared in *The Busy Coder's Guide to Android Development* specifically in the context of local services. However, AIDL adds a fair bit of overhead, which is not necessary with local services. After all, 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.

So, given our AIDL description, let us examine some implementations, specifically for remote services.

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">
 <uses-sdk
      android:minSdkVersion="3"
      android:targetSdkVersion="6"
  />
  <supports-screens
    android:largeScreens="false"
    android:normalScreens="true"
    android:smallScreens="false"
  />
  <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 *The Busy Coder's Guide to Android Development* 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:

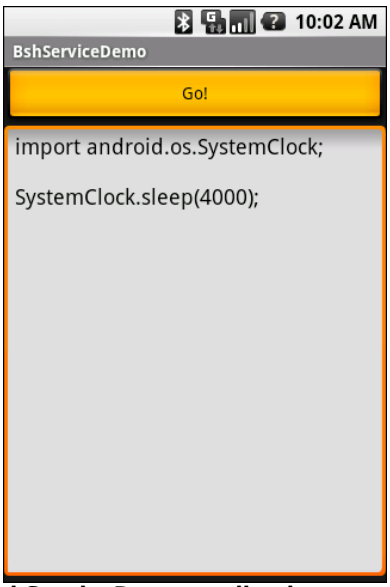


Figure 67. The BshServiceDemo application, running a long script

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:

```
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) {
    runOnUiThread(new Runnable() {
      public void run() {
        failureImpl(error);
      }
    });
  }
};

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);

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

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.

The Bind That Fails

Sometimes, a call to `bindService()` will fail for some reason. The most common cause will be an invalid `Intent` – for example, you might be trying to bind to a `Service` that you failed to register in the manifest. The `bindService()` method returns a `boolean` value indicating whether or not there was an immediate problem, so you can take appropriate steps.

For local services, this is usually just a coding problem. For remote services, though, it could be that the service you are trying to work with has not been installed on the device. You have two approaches for dealing with this:

1. You can watch for `bindService()` to return `false` and assume that means the service is not installed
2. You can use introspection to see if the service is indeed installed before you even try calling `bindService()`

We will look at introspection techniques later in this book.

CHAPTER 16

Introspection and Integration

Introspection, from a software development standpoint, refers to inspecting one's environment at runtime to figure out what is possible and how to integrate disparate components. In Android, this comes in two main flavors:

1. Sometimes, the introspection is based on a `Uri` – you get a `Uri` from someplace, and to you it is an opaque handle, and you do not necessarily know what to do with it
2. Sometimes, the introspection is more at the `Intent` or package level, where you are trying to figure out if such-and-so application is installed, or asking Android to give you choices for who can handle such-and-so `Intent`, etc.

Android has a fairly rich, somewhat disheveled, and frequently misunderstood collection of introspection techniques. This chapter outlines some of those, so you know how to make use of them to enhance your own applications.

We start with the ways to inject other activities into your own application's option menus and how, in theory, you could use that to get your activity in somebody else's option menu. We then cover ACTION_SEND and `createChooser()`, showing how you can hook into capabilities without knowing exactly what all the options are. We then spend a pair of sections examining `PackageManager` and how you can use it to peer inside the device and see what all is installed. We then see how you can implement

ACTION_SEND support in your own application, so you can appear as an option when some other application allows its users to "send" things. We wrap up with a discussion of how to create application shortcuts that can be dropped onto a user's home screen.

Would You Like to See the Menu?

Another way to give the user ways to take actions on a piece of content, without you knowing what actions are possible, is to inject a set of menu choices into the options menu via `addIntentOptions()`. This method, available on `Menu`, takes an `Intent` and other parameters and fills in a set of menu choices on the `Menu` instance, each representing one possible action. Choosing one of those menu choices spawns the associated activity.

The canonical example of using `addIntentOptions()` illustrates another flavor of having a piece of content and not knowing the actions that can be taken. Android applications are perfectly capable of adding new actions to existing content types, so even though you wrote your application and know what you expect to be done with your content, there may be other options you are unaware of that are available to users.

For example, imagine the tagging subsystem mentioned in the introduction to this chapter. It would be very annoying to users if, every time they wanted to tag a piece of content, they had to go to a separate tagging tool, then turn around and pick the content they just had been working on (if that is even technically possible) before associating tags with it. Instead, they would probably prefer a menu choice in the content's own “home” activity where they can indicate they want to tag it, which leads them to the set-a-tag activity and tells that activity what content should get tagged.

To accomplish this, the tagging subsystem should set up an intent filter, supporting any piece of content, with their own action (e.g., `ACTION_TAG`) and a category of `CATEGORY_ALTERNATIVE`. The category `CATEGORY_ALTERNATIVE` is the convention for one application adding actions to another application's content.

If you want to write activities that are aware of possible add-ons like tagging, you should use `addIntentOptions()` to add those add-ons' actions to your options menu, such as the following:

```
Intent intent = new Intent(null, myContentUri);

intent.addCategory(Intent.ALTERNATIVE_CATEGORY);
menu.addIntentOptions(Menu.ALTERNATIVE, 0,
                      new ComponentName(this,
                                        MyActivity.class),
                      null, intent, 0, null);
```

Here, `myContentUri` is the content `Uri` of whatever is being viewed by the user in this activity, `MyActivity` is the name of the activity class, and `menu` is the menu being modified.

In this case, the `Intent` we are using to pick actions from requires that appropriate intent receivers support the `CATEGORY_ALTERNATIVE`. Then, we add the options to the menu with `addIntentOptions()` and the following parameters:

- The sort position for this set of menu choices, typically set to `0` (appear in the order added to the menu) or `ALTERNATIVE` (appear after other menu choices)
- A unique number for this set of menu choices, or `0` if you do not need a number
- A `ComponentName` instance representing the activity that is populating its menu – this is used to filter out the activity's own actions, so the activity can handle its own actions as it sees fit
- An array of `Intent` instances that are the “specific” matches – any actions matching those intents are shown first in the menu before any other possible actions
- The `Intent` for which you want the available actions
- A set of flags. The only one of likely relevance is represented as `MATCH_DEFAULT_ONLY`, which means matching actions must also implement the `DEFAULT_CATEGORY` category. If you do not need this, use a value of `0` for the flags.

- An array of `Menu.Item`, which will hold the menu items matching the array of `Intent` instances supplied as the “specifics”, or `null` if you do not need those items (or are not using “specifics”)

Give Users a Choice

Let's suppose you want to send a message. There are many ways you can do that in standard Android: email (via the Email or Gmail apps) or a text message. Third-party apps may also have the notion of "sending", such as alternative email clients (e.g., K9) or Twitter clients (e.g., Twidroid).

You want to allow the user to choose both the means (i.e., the application) and the destination (i.e., the contact or address) for this message to be sent.

That can be handled very simply in Android:

```
void sendIt(String theMessage) {
  Intent i=new Intent(Intent.ACTION_SEND);

  i.setType("text/plain");
  i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
  i.putExtra(Intent.EXTRA_TEXT, theMessage);

  startActivity(Intent.createChooser(i,
                                     getString(R.string.share_title)));
}
```

The magic is in the `ACTION_SEND` protocol and `createChooser()`.

`ACTION_SEND` is an activity action that says, "Hey! I want to send...ummm...something! To...er...somebody! Yeah!". The documentation for `ACTION_SEND` describes a series of `Intent` extras you can attach to the `Intent` that provides the actual content of the message, from the message body (`EXTRA_TEXT` and `EXTRA_STREAM`) to the subject line (`EXTRA_SUBJECT`). You can even supply specific addresses (`EXTRA_EMAIL`, `EXTRA_CC`, `EXTRA_BCC`), if you know them already.

The `createChooser()` static method on `Intent` returns another `Intent`, one to a system-provided dialog-themed activity that gives the user a choice of

available activities that can support the desired action. This list is determined on the fly by introspection, seeing what capabilities exist on the device. So, one user might get just Email and Messaging, while another user might get K9, Gmail, Messaging, and Twidroid. Your code stays the same – Android provides the "glue" that connects your application to these arbitrary other applications that can handle your request to send the message.

Asking Around

The `addIntentOptions()` and `createChooser()` methods in turn use `queryIntentActivityOptions()` for the “heavy lifting” of finding possible actions. The `queryIntentActivityOptions()` method is implemented on `PackageManager`, which is available to your activity via `getPackageManager()`.

The `queryIntentActivityOptions()` method takes some of the same parameters as does `addIntentOptions()`, notably the caller `ComponentName`, the “specifics” array of `Intent` instances, the overall `Intent` representing the actions you are seeking, and the set of flags. It returns a `List` of `Intent` instances matching the stated criteria, with the “specifics” ones first.

If you would like to offer alternative actions to users, but by means other than `addIntentOptions()`, you could call `queryIntentActivityOptions()`, get the `Intent` instances, then use them to populate some other user interface (e.g., a toolbar).

A simpler version of this method, `queryIntentActivities()`, is used by the `Introspection/Launchalot` sample application. This presents a "launcher" – an activity that starts other activities – but uses a `ListView` rather than a grid like the Android default home screen uses.

Here is the Java code for `Launchalot` itself:

```
package com.commonsware.android.launchalot;

import android.app.ListActivity;
import android.content.ComponentName;
```

```
import android.content.Intent;
import android.content.pm.ActivityInfo;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ImageView;
import android.widget.ListView;
import android.widget.TextView;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Launchalot extends ListActivity {
  AppAdapter adapter=null;

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

    PackageManager pm=getPackageManager();
    Intent main=new Intent(Intent.ACTION_MAIN, null);

    main.addCategory(Intent.CATEGORY_LAUNCHER);

    List<ResolveInfo> launchables=pm.queryIntentActivities(main, 0);

    Collections.sort(launchables,
                     new ResolveInfo.DisplayNameComparator(pm));

    adapter=new AppAdapter(pm, launchables);
    setListAdapter(adapter);
  }

  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    ResolveInfo launchable=adapter.getItem(position);
    ActivityInfo activity=launchable.activityInfo;
    ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                         activity.name);
    Intent i=new Intent(Intent.ACTION_MAIN);

    i.addCategory(Intent.CATEGORY_LAUNCHER);
    i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
               Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);
    i.setComponent(name);

    startActivity(i);
  }
```

```
  class AppAdapter extends ArrayAdapter<ResolveInfo> {
    private PackageManager pm=null;

    AppAdapter(PackageManager pm, List<ResolveInfo> apps) {
      super(Launchalot.this, R.layout.row, apps);
      this.pm=pm;
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      if (convertView==null) {
        convertView=newView(parent);
      }

      bindView(position, convertView);

      return(convertView);
    }

    private View newView(ViewGroup parent) {
      return(getLayoutInflater().inflate(R.layout.row, parent, false));
    }

    private void bindView(int position, View row) {
      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(getItem(position).loadLabel(pm));

      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      icon.setImageDrawable(getItem(position).loadIcon(pm));
    }
  }
}
```

In `onCreate()`, we:

- Get a `PackageManager` object via `getPackageManager()`
- Create an `Intent` for `ACTION_MAIN` in `CATEGORY_LAUNCHER`, which identifies activities that wish to be considered "launchable"
- Call `queryIntentActivities()` to get a `List` of `ResolveInfo` objects, each one representing one launchable activity
- Sort those `ResolveInfo` objects via a `ResolveInfo.DisplayNameComparator` instance
- Pour them into a custom `AppAdapter` and set that to be the contents of our `ListView`

`AppAdapter` is an `ArrayAdapter` subclass that maps the icon and name of the launchable `Activity` to a row in the `ListView`, using a custom row layout.

Finally, in `onListItemClick()`, we construct an `Intent` that will launch the clicked-upon `Activity`, given the information from the corresponding `ResolveInfo` object. Not only do we need to populate the `Intent` with `ACTION_MAIN` and `CATEGORY_LAUNCHER`, but we also need to set the component to be the desired `Activity`. We also set `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED` flags, following Android's own launcher implementation from the Home sample project. Finally, we call `startActivity()` with that `Intent`, which opens up the activity selected by the user.

The result is a simple list of launchable activities:

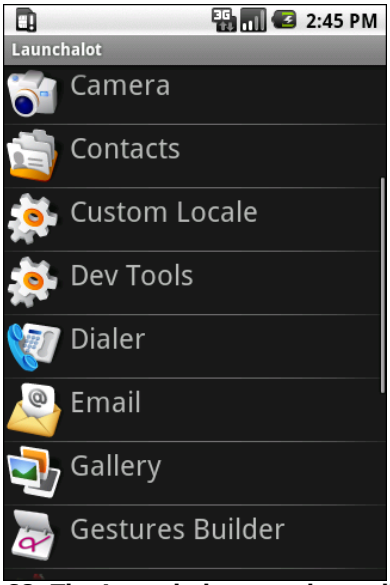


Figure 68. The Launchalot sample application

There is also a `resolveActivity()` method that takes a template Intent, as do `queryIntentActivities()` and `queryIntentActivityOptions()`. However, `resolveActivity()` returns the single best match, rather than a list.

Middle Management

The `PackageManager` class offers much more than merely `queryIntentActivities()` and `queryIntentActivityOptions()`. It is your gateway to all sorts of analysis of what is installed and available on the device where your application is installed and available. If you want to be able to intelligently connect to third-party applications based on whether or not they are around, `PackageManager` is what you will want.

Finding Applications and Packages

Packages are what get installed on the device – a package is the in-device representation of an APK. An application is defined within a package's manifest. Between the two, you can find out all sorts of things about existing software installed on the device.

Specifically, `getInstalledPackages()` returns a `List` of `PackageInfo` objects, each of which describes a single package. Here, you can find out:

- The version of the package, in terms of a monotonically increasing number (`versionCode`) and the display name (`versionName`)
- Details about all of the components – activities, services, etc. – offered by this package
- Details about the permissions the package requires

Similarly, `getInstalledApplications()` returns a `List` of `ApplicationInfo` objects, each providing data like:

- The user ID that the application will run as
- The path to the application's private data directory
- Whether or not the application is enabled

In addition to those methods, you can call:

- `getApplicationIcon()` and `getApplicationLabel()` to get the icon and display name for an application

- `getLaunchIntentForPackage()` to get an `Intent` for something launchable within a named package
- `setApplicationEnabledSetting()` to enable or disable an application

Finding Resources

You can access resources from another application, apparently without any security restrictions. This may be useful if you have multiple applications and wish to share resources for one reason or another.

The `getResourcesForActivity()` and `getResourcesForApplication()` methods on `PackageManager` return a `Resources` object. This is just like the one you get for your own application via `getResources()` on any `Context` (e.g., `Activity`). However, in this case, you identify what activity or application you wish to get the `Resources` from (e.g., supply the application's package name as a `String`).

There are also `getText()` and `getXml()` methods that dive into the `Resources` object for an application and pull out specific `String` or `XmlPullParser` objects. However, these require you to know the resource ID of the resource to be retrieved, and that may be difficult to manage between disparate applications.

Finding Components

Not only does Android offer "query" and "resolve" methods to find activities, but it offers similar methods to find other sorts of Android components:

- `queryBroadcastReceivers()`
- `queryContentProviders()`
- `queryIntentServices()`
- `resolveContentProvider()`
- `resolveService()`

For example, you could use `resolveService()` to determine if a certain remote service is available, so you can disable certain UI elements if the service is not on the device. You could achieve the same end by calling `bindService()` and watching for a failure, but that may be later in the application flow than you would like.

There is also a `setComponentEnabledSetting()` to toggle a component (activity, service, etc.) on and off. While this may seem esoteric, there are a number of possible uses for this method, such as:

- Flagging a launchable activity as disabled in your manifest, then enabling it programmatically after the user has entered a license key, achieved some level or standing in a game, or any other criteria
- Controlling whether a `BroadcastReceiver` registered in the manifest is hooked into the system or not, replicating the level of control you have with `registerReceiver()` while still taking advantage of the fact that a manifest-registered `BroadcastReceiver` can be started even if no other component of your application is running

Get In the Loop

Earlier in this chapter, we saw how to request to send a message somewhere via `ACTION_SEND`. If you have an application that has an intrinsic notion of "sending" or "sharing" things, you may wish to advertise that your application can respond to `ACTION_SEND`. Then, you automatically integrate with every Android application ever written that uses `ACTION_SEND`, without any additional work on their part.

The key is in the intent filter.

For example, take a look at `Introspection/TwitterSender`. This uses some of the guts of the `AppWidget/TwitterWidget` sample we saw in an earlier chapter. In that case, we fetched the Twitter timeline for a user. This time, we are going to support `ACTION_SEND`, routing a message to Twitter in the form of a status update. Some Android Twitter applications do this already, of course.

Our application will have three activities:

1. A `PreferenceActivity` to collect the screen name and password, largely cloned from the `TwitterWidget` sample
2. The main activity (`TwitterSender`) that supports `ACTION_SEND`
3. A test activity that sends a message via `ACTION_SEND` and `createChooser()`, so our `TwitterSender` will be an option

We will also make use of an `IntentService` to actually send the tweet in the background, freeing up the activity while avoiding having to deal with background threads ourselves.

The Manifest

First, let's take a peek at the project's `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.tsender"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-permission android:name="android.permission.INTERNET" />
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:name="TSPrefs"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="TwitterSenderTest"
                  android:label="@string/test_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="TwitterSender"
                  android:label="@string/sender_name"
                  android:theme="@android:style/Theme.NoDisplay">
            <intent-filter android:label="@string/sender_name">
                <action android:name="android.intent.action.SEND" />
                <data android:mimeType="text/plain" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
```

```
        <service android:name=".SenderService" />
    </application>
</manifest>
```

Two of the activities have normal "please show me in the Launcher" configurations. The service is simple as well.

The third activity – `TwitterSender` – has a somewhat more unusual `<intent-filter>` element. Here, we state that this activity should respond to any `Intent` used to start an activity that:

- References the `ACTION_SEND` action,
- Has content that is of type `text/plain`, and
- Appears in the `DEFAULT` category

That is the "secret sauce" that enables `TwitterSender` to work with `ACTION_SEND` Intent objects of the type we aim to support.

The PreferenceActivity

The `TSPrefs` activity is just an ordinary `PreferenceActivity`:

```
package com.commonsware.android.tsender;

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class TSPrefs extends PreferenceActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    addPreferencesFromResource(R.xml.preferences);
  }
}
```

It loads a pair of preferences from `res/xml/preferences.xml`, to collect the screen name and password:

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

```
  <PreferenceCategory android:title="User Account">
    <EditTextPreference
      android:key="user"
      android:title="User Name"
      android:summary="Your Twitter screen name"
      android:dialogTitle="Enter your Twitter user name" />
    <EditTextPreference
      android:key="password"
      android:title="Password"
      android:summary="Your Twitter account password"
      android:password="true"
      android:dialogTitle="Enter your Twitter password" />
  </PreferenceCategory>
</PreferenceScreen>
```

The Main Activity

`TwitterSender` itself spends most of its time making sure the `ACTION_SEND` request makes sense for Twitter:

```
package com.commonsware.android.tsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class TwitterSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String tweet=getIntent().getStringExtra(Intent.EXTRA_TEXT);

    if (TextUtils.isEmpty(tweet)) {
      tweet=getIntent().getStringExtra(Intent.EXTRA_SUBJECT);
    }

    if (TextUtils.isEmpty(tweet)) {
      Toast
        .makeText(this, "No message supplied!", Toast.LENGTH_LONG)
        .show();
    }
    else {
      if (tweet.length()>140) {
        tweet=TextUtils.substring(tweet, 0, 139);
      }

      Intent i=new Intent(this, SenderService.class);
```

```
      i.putExtra(Intent.EXTRA_TEXT, tweet);
      startService(i);

      Toast
        .makeText(this, "Your tweet is on its way!", Toast.LENGTH_LONG)
        .show();
    }

    finish();
  }
}
```

We check both `EXTRA_TEXT` and `EXTRA_SUBJECT` to see if there is a message to be sent. If not, we raise a `Toast` to tell the user that something is messed up. Assuming we have a message, we make sure it fits in 140 characters (the Twitter limit), wrap that in an `Intent` for our `SenderService` (see below), and start up that service. Then, we display a `Toast` to indicate the message is en route.

In either case – valid or invalid input – we `finish()` the activity, without showing any actual UI. That is because there is nothing really to show, having delegated all results to the `Toast` class. Because there is no UI to be shown, we use the `Theme.NoDisplay` them in our `AndroidManifest.xml` entry for this activity – this suppresses the otherwise-empty activity window from displaying.

The IntentService

`TwitterSender` uses an `IntentService` so that the work of sending the tweet is done on a background thread. An `IntentService` will also automatically shut down when the tweet is sent (assuming there are no more tweets queued up to be sent). Hence, for situations like this, where we want something done in the background but the originating activity will be gone, an `IntentService` is a great choice.

All the `SenderService` does is get the screen name and password from the default `SharedPreferences`, feed them into a `Twitter` object (using the same JTwitter JAR that `TwitterWidget` used), pull the message out of the `Intent` that `TwitterSender` used with `startService()`, and update the user's Twitter status with that message:

```
package com.commonsware.android.tsender;

import android.app.IntentService;
import android.content.Intent;
import android.content.SharedPreferences;
import android.preference.PreferenceManager;
import winterwell.jtwitter.Twitter;

public class SenderService extends IntentService {
  public SenderService() {
    super("SenderService");
  }

  @Override
  public void onHandleIntent(Intent intent) {
    SharedPreferences prefs=PreferenceManager.getDefaultSharedPreferences(this);
    Twitter client=new Twitter(prefs.getString("user", ""),
                               prefs.getString("password", ""));

    client.updateStatus(intent.getStringExtra(Intent.EXTRA_TEXT));
  }
}
```

The Test Activity

Our test activity – `TwitterSenderTest` – just fires off a pre-defined tweet using `ACTION_SEND`, using the `createChooser()` technique described earlier in this chapter:

```
package com.commonsware.android.tsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class TwitterSenderTest extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    sendIt("This is a test of TwitterSender");

    finish();
  }

  void sendIt(String theMessage) {
    Intent i=new Intent(Intent.ACTION_SEND);
```

```
    i.setType("text/plain");
    i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
    i.putExtra(Intent.EXTRA_TEXT, theMessage);

    startActivity(Intent.createChooser(i,
                                       getString(R.string.share_title)));
  }
}
```

The Results

Because we defined two `LAUNCHER` activities in our manifest, we have two icons in the Launcher – TS Prefs and TS Test.

Running TS Prefs brings up our preferences, which you will need to set before trying TS Test:

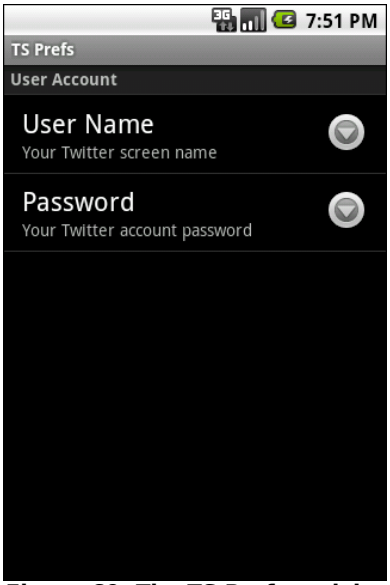


Figure 69. The TS Prefs activity

Running TS Test will bring up a chooser to pick which means you want to use to send the message:

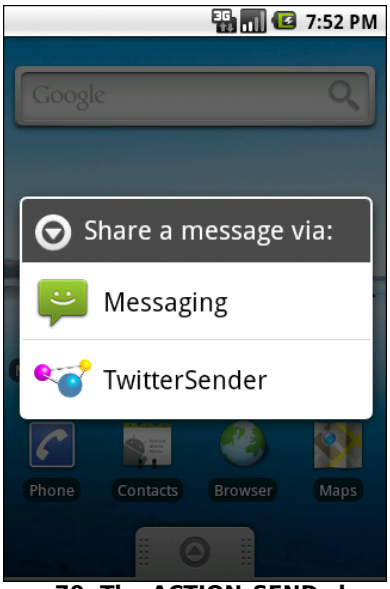


Figure 70. The ACTION_SEND chooser

If you choose `TwitterSender`, you will get a `Toast` indicating the tweet is being sent:

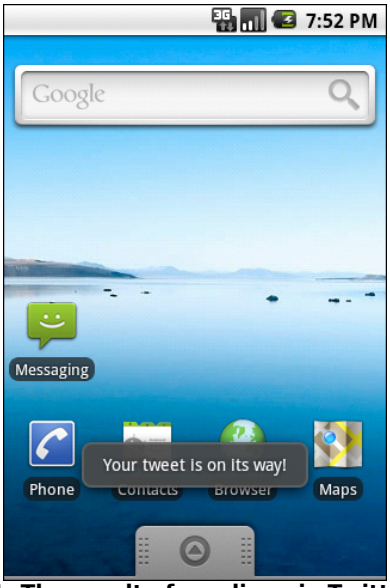


Figure 71. The result of sending via TwitterSender

And, of course, if you check the timeline of the user you specified in the preferences, you will see the message appear there.

Take the Shortcut

Another way to integrate with Android is to offer custom shortcuts. Shortcuts are available from the home screen. Whereas app widgets allow you to draw on the home screen, shortcuts allow you to wrap a custom Intent with an icon and caption and put that on the home screen. You can use this to drive users not just to your application's "front door", like the launcher icon, but to some specific capability within your application, like a bookmark.

In our case, in the `Introspection/QuickSender` sample, we will allow users to create shortcuts that use `ACTION_SEND` to send a pre-defined message, either to a specific address or anywhere, as we have seen before in this chapter.

Once again, the key is in the intent filter.

Registering a Shortcut Provider

Here is the manifest for `QuickSender`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.commonsware.android.qsender"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:name="QuickSender"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.CREATE_SHORTCUT" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Our single activity does not implement a traditional launcher `<intent-filter>`. Rather, it has one that watches for a `CREATE_SHORTCUT` action. This does two things:

1. It means that our activity will show up in the list of possible shortcuts a user can configure
2. It means this activity will be the recipient of a `CREATE_SHORTCUT` Intent if the user chooses this application from the shortcuts list

Implementing a Shortcut Provider

The job of a shortcut-providing activity is to:

- Create an `Intent` that will be what the shortcut launches
- Return that `Intent` and other data to the activity that started the shortcut provider
- Finally, `finish()`, so the caller gets control

You can see all of that in the `QuickSender` implementation:

```
package com.commonsware.android.qsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.widget.TextView;

public class QuickSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void save(View v) {
    Intent shortcut=new Intent(Intent.ACTION_SEND);
    TextView addr=(TextView)findViewById(R.id.addr);
    TextView subject=(TextView)findViewById(R.id.subject);
    TextView body=(TextView)findViewById(R.id.body);
    TextView name=(TextView)findViewById(R.id.name);

    if (!TextUtils.isEmpty(addr.getText())) {
```

```
      shortcut.putExtra(Intent.EXTRA_EMAIL, addr.getText().toString());
    }

    if (!TextUtils.isEmpty(subject.getText())) {
      shortcut.putExtra(Intent.EXTRA_SUBJECT, subject.getText().toString());
    }

    if (!TextUtils.isEmpty(body.getText())) {
      shortcut.putExtra(Intent.EXTRA_TEXT, body.getText().toString());
    }

    shortcut.setType("text/plain");

    Intent result=new Intent();

    result.putExtra(Intent.EXTRA_SHORTCUT_INTENT, shortcut);
    result.putExtra(Intent.EXTRA_SHORTCUT_NAME,
                    name.getText().toString());
    result.putExtra(Intent.EXTRA_SHORTCUT_ICON_RESOURCE,
                    Intent.ShortcutIconResource.fromContext(
                                        this,
                                        R.drawable.icon));

    setResult(RESULT_OK, result);
    finish();
  }
}
```

The `shortcut Intent` is the one that will be launched when the user taps the shortcut icon on the home screen. The `result Intent` packages up shortcut plus the icon and caption, where the icon is converted into an `Intent.ShortcutIconResource` object. That `result Intent` is then used with the `setResult()` call, to pass that back to whatever called `startActivityForResult()` to open up `QuickSender`. Then, we `finish()`.

At this point, all the information about the shortcut is in the hands of Android (or, more accurately, the home screen application), which can add the icon to the home screen.

Using the Shortcuts

To create a custom shortcut using `QuickSender`, long-tap on the background of the home screen to bring up the customization options:

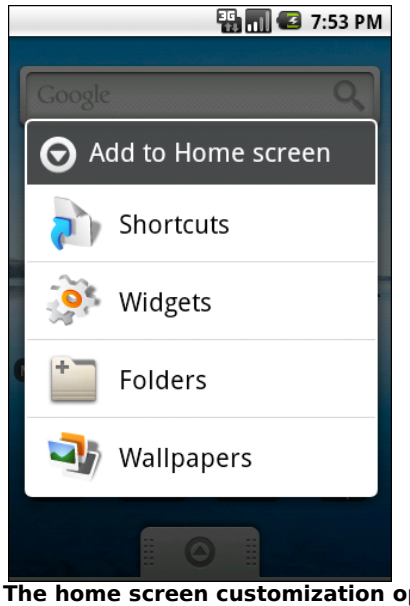


Figure 72. The home screen customization options list

Choose Shortcuts, and scroll down to find `QuickSender` in the list:

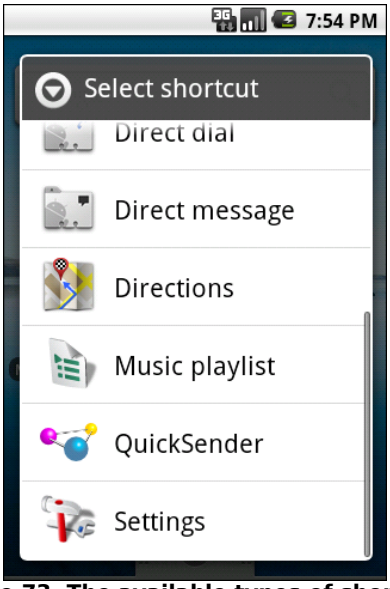


Figure 73. The available types of shortcuts

Click the `QuickSender` entry, which will bring up our activity with the form to define what to send:

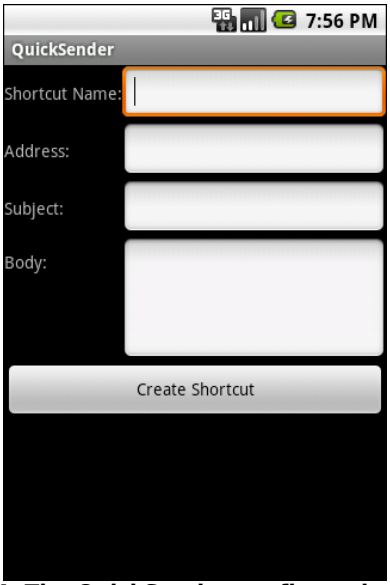


Figure 74. The QuickSender configuration activity

Fill in the name, either the subject or body, and optionally the address. Then, click the Create Shortcut button, and you will find your shortcut sitting on your home screen:

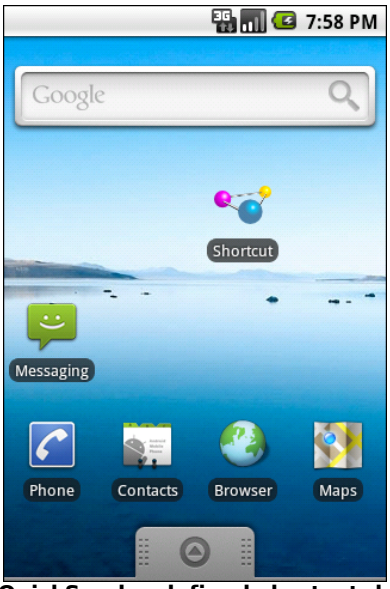


Figure 75. The QuickSender-defined shortcut, labeled Shortcut

If you launch that shortcut, Android will bring up a special chooser, to allow you to not only pick how to send the message, but optionally make that method the default for all future requests:

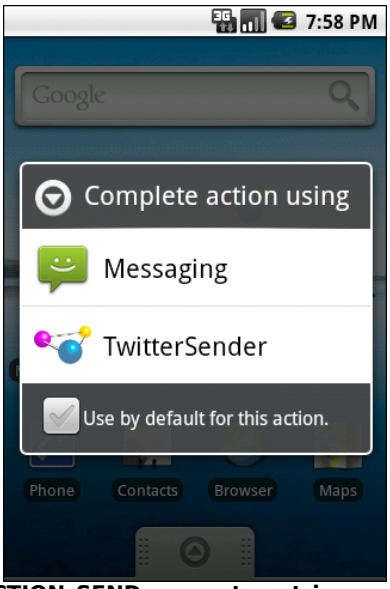


Figure 76. The ACTION_SEND request, as triggered by the shortcut

Depending on what you choose, of course, will dictate how the message actually gets sent.

Homing Beacons for Intents

If you are encountering problems with `Intent` resolution – you create an `Intent` for something and try starting an `Activity` or `Service` with it, and it does not work – you can add the `FLAG_DEBUG_LOG_RESOLUTION` flag to the `Intent`. This will dump information to LogCat about how the `Intent` resolution occurred, so you can better diagnose what might be going wrong.

CHAPTER 17

Device Configuration

This chapter is a bit esoteric for most developers. It covers various places where Android has configuration-style data and how one can modify it. This mostly should be of interest to teams responsible for configuring devices *en masse*, such as an enterprise or consulting teams supporting an enterprise.

Some of what is written here, notably the portions involving root access, have not been tested by the author. This chapter is the result of a research project, and so while the techniques have been described online as being used, the author is not presently able to confirm their accuracy.

Also, note that this material may change significantly between Android releases, and errors may result in permanent damage to a device (e.g., somehow leave it in a state where it cannot boot and cannot reset to factory settings). If you are concerned about these things, do not bother with this chapter.

However, the rumor that reading this chapter causes hair loss in men is completely unsubstantiated.

We start by enumerating various places where ordinary Android applications can modify the device configuration. Then, we look at some other places where configuration data gets stored that regular Android applications cannot access, but that device manufacturer-written apps

might. We wrap by a brief discussion of the issues involved in automating some of this configuration, such as an enterprise deployment of hundreds or thousands of Android devices.

The Happy Shiny Way

Certainly, some portions of the configuration of a device are available for applications to manipulate without special privileges or permissions. There are a few places in the Android SDK where you can modify settings; some of these are described below.

Settings.System

As described in the chapter on system services, there is a `Settings.System` class in the `android.provider` package that allows you to configure the device. These range from whether the password is shown when being typed in on a password-defined `EditText` to whether "airplane mode" is on (i.e., whether all radios are disabled, perhaps in compliance with an airline's regulations).

You can retrieve these settings via static getter methods on `Settings.System` (e.g., `getInt()`, `getFloat()`), keyed by public static data members on `Settings.System` itself. There are a similar set of setters to modify these settings. There are methods to get and set a `Configuration` object, which allows one to get or set several settings at once, but it does not appear that `Configuration` supports the full range of possible settings. Hence, in all likelihood, if you wish to automate manipulating these settings, you will want to create an application to do that...or apply some less public techniques described below.

WifiManager

It may be that you want to pre-populate a WiFi network on a device, so it is ready to use with an office wireless network without manual on-screen configuration. To do that, you can use the `WifiManager` class, obtained by

calling `Context.getSystemService(WIFI_SERVICE)` and downcasting the result to `WifiManager`.

`WifiManager` has an `addNetwork()` method that takes a `WifiConfiguration` object as a parameter. The `WifiConfiguration` object has numerous fields for describing the network, including the SSID, the pre-shared key for WPA-PSK, and so on. This method returns an integer ID for the network being added. Initially, the network is marked as disabled, so most likely you will want to immediately follow the `addNetwork()` call with an `enableNetwork()` call, supplying the network ID from `addNetwork()` and `true` to enable the network.

The Dark Arts

While you have control over a fair number of settings this way, there are still others that appear to be unreachable through the standard SDK. Modifying these areas of the device configuration require techniques that are not recommended if you can at all avoid them. Contributions to the Android open source project are welcome, so you might consider writing something that will allow for device configuration without having to use undocumented risky steps, while maintaining the security that the Android project has established.

Settings.Secure

As described in the chapter on system services, prior to Android 1.5, there were more settings in `Settings.System`. However, due to perceived abuses by third-party developers, a number of them were moved into `Settings.Secure`. While you can read these settings as before (via static getter methods), attempts to use the setter methods will result in errors.

If you wish to populate the `Settings.Secure` values, you have two choices:

1. Create an Android application that has the rights to use those setter methods. While the exact rules here are unclear, it is possible that an application signed with the same production signing key as the

firmware will have such rights. Hardware manufacturers, therefore, should be in position to create such an application.

2. Modify the underlying SQLite database that holds the data. That database, as of Android 1.5, is `/data/data/com.android.providers.settings/databases/settings.db`, and the secure settings are stored in a table named secure. However, to either execute SQL statements against this database, or to replace the database outright, you would need `root` permissions. Many devices can be "rooted", though for the publicly documented hacks, rooting is a permanent process. Hardware manufacturers may know if there is a way on their devices to temporarily have `root` privileges, long enough to run some scripts.

System Properties

At an even lower level are so-called system properties. You can see these by running `adb shell getprop` with a device or emulator attached. This contains everything from the URLs from which to obtain legal agreements to display on intial sign-on to the location where application-not-responding (ANR) traces are dumped.

The only known way to modify these settings is to actually modify the init script (`init.rc`) for Android itself, adding in setprop commands to override the system default values. For example, to hardwire in some DNS resolvers, rather than rely on DHCP, you could add statements like `setprop net.dns1 ...` and `setprop net.dns2 ...` (where the `...` are replaced with dot-notation IP addresses for the servers).

Bear in mind that `init.rc` might well be replaced when a device is upgraded to newer versions of Android, so making changes this way may not be reliable.

Automation, Both Shiny and Dark

If you are trying to modify a single device, and you can stick to SDK-supported changes, either just use the built-in Settings screens or write a standard Android application of your own to modify those settings.

Modifying settings on a bunch of devices this way, though, can be tedious. You would need to install the application, perhaps off of an internet office Web server, and that would require entering a URL in on the Browser application to download the APK. After a few installation screens, you could then run the application, then uninstall it. All of that cannot readily be automated, and it still does not cover the situations where you wish to modify settings beyond those supported by the SDK.

For bulk work, it may be simpler, albeit substantially more dangerous, to automate this process via `adb` commands. For example, you could create a SQL script that updates the `Settings.System` (`system` table) and `Settings.Secure` (`secure` table) and apply that script to the aforementioned settings.db via `adb shell sqlite3`. There should be some similar means to update the WiFi networks, though where that data is stored is not obvious. This, of course, requires root access.

PART IV – Advanced Development

CHAPTER 18

Testing

Presumably, you will want to test your code, beyond just playing around with it yourself by hand.

To that end, Android includes the JUnit test framework in the SDK, along with special test classes that will help you build test cases that exercise Android components, like activities and services. Even better, Android 1.5 has "gone the extra mile" and can pre-generate your test harness for you, to make it easier for you to add in your own tests.

This chapter assumes you have some familiarity with JUnit, though you certainly do not need to be an expert. You can learn more about JUnit at the JUnit site, from various books, and from the JUnit Yahoo forum.

You Get What They Give You

When you create a project in Android 1.5 using android create project, Android automatically creates a new `tests/` directory inside the project directory. If you look in there, you will see a complete set of Android project artifacts: manifest, source directories, resources, etc. This is actually a test project, designed to partner with the main project to create a complete testing solution.

In fact, that test project is all ready to go, other than not having any tests of significance. If you build and install your main project (onto an emulator or

device), then build and install the test project, you will be able to run unit tests.

Android ships with a very rudimentary JUnit runner, called `InstrumentationTestRunner`. Since this class resides in the Android environment (emulator or device), you need to invoke the runner to run your tests on the emulator or device itself. To do this, you can run the following command from a console:

```
adb shell am instrument -w
com.commonsware.android.contacts.spinners.tests/android.test.InstrumentationTest
Runner
```

In this case, we are instructing Android to run all the available test cases for the `com.commonsware.android.database` package, as this chapter uses some tests implemented on the `Contacts/Spinners` sample project.

If you were to run this on your own project, substituting in your package name, with just the auto-generated test files, you should see results akin to:

```
com.commonsware.android.database.ContactsDemoTest:.
Test results for InstrumentationTestRunner=.
Time: 0.61

OK (1 test)
```

The first line will differ, based upon your package and the name of your project's initial activity, but the rest should be the same, showing that a single test was run, successfully.

Of course, this is only the beginning.

Erecting More Scaffolding

Here is the source code for the test case that Android automatically generates for you:

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

```
import android.test.ActivityInstrumentationTestCase;

/**
 * This is a simple framework for a test of an Application. See
 * {@link android.test.ApplicationTestCase ApplicationTestCase} for more
information on
 * how to write and extend Application tests.
 * <p/>
 * To run this test, you can type:
 * adb shell am instrument -w \
 * -e class com.commonsware.android.database.ContactsDemoTest \
 * com.commonsware.android.database.tests/android.test.InstrumentationTestRunner
 */
public class ContactsDemoTest extends
ActivityInstrumentationTestCase<ContactsDemo> {

 public ContactsDemoTest() {
 super("com.commonsware.android.database", ContactsDemo.class);
 }

}
```

As you can see, there are no actual test methods. Instead, we have an `ActivityInstrumentationTestCase` implementation named `ContactsDemoTest`. The class name was generated by adding Test to the end of the main activity (`ContactsDemo`) of the project.

In the next section, we will examine `ActivityInstrumentationTestCase` more closely and see how you can use it to, as the name suggests, test your activities.

However, you are welcome to create ordinary JUnit test cases in Android – after all, this is just JUnit, merely augmented by Android. So, you can create classes like this:

```
package com.commonsware.android.contacts.spinners;

import junit.framework.TestCase;

public class SillyTest extends TestCase {
  protected void setUp() throws Exception {
    super.setUp();

    // do initialization here, run on every test method
  }

  protected void tearDown() throws Exception {
```

```
    // do termination here, run on every test method

    super.tearDown();
  }

  public void testNonsense() {
    assertTrue(1==1);
  }
}
```

There is nothing Android-specific in this test case. It is simply standard JUnit, albeit a bit silly.

You can also create test suites, to bundle up sets of tests for execution. Here, though, if you want, you can take advantage of a bit of Android magic: `TestSuiteBuilder`. `TestSuiteBuilder` uses reflection to find test cases that need to be run, as shown below:

```
package com.commonsware.android.contacts.spinners;

import android.test.suitebuilder.TestSuiteBuilder;
import junit.framework.Test;
import junit.framework.TestSuite;

public class FullSuite extends TestSuite {
  public static Test suite() {
    return(new TestSuiteBuilder(FullSuite.class)
                .includeAllPackagesUnderHere()
                .build());
  }
}
```

Here, we are telling Android to find all test cases located in `FullSuite`'s package (`com.commonsware.android.database`) and all sub-packages, and to build a `TestSuite` out of those contents.

A test suite may or may not be necessary for you. The command shown above to execute tests will execute any test cases it can find for the package specified on the command line. If you want to limit the scope of a test run, though, you can use the `-e` switch to specify a test case or suite to run:

```
adb shell am instrument -e class
com.commonsware.android.database.ContactsDemoTest -w
com.commonsware.android.database.tests/android.test.InstrumentationTestRunner
```

Here, we indicate we only want to run `ContactsDemoTest`, not all test cases found in the package.

Testing Real Stuff

While ordinary JUnit tests are certainly helpful, they are still fairly limited, since much of your application logic may be tied up in activities, services, and the like.

To that end, Android has a series of `TestCase` classes you can extend designed specifically to assist in testing these sorts of components.

ActivityInstrumentationTestCase

The test case created by Android's SDK tools, `ContactsDemoTest` in our example, is an `ActivityInstrumentationTestCase`. This class will run your activity for you, giving you access to the `Activity` object itself. You can then:

- Access your widgets
- Invoke public and package-private methods (more on this below)
- Simulate key events

Of course, the automatically-generated `ActivityInstrumentationTestCase` does none of that, since it does not know much about your activity. Below you will find an augmented version of `ContactsDemoTest` that does a little bit more:

```
package com.commonsware.android.contacts.spinners;

import android.test.ActivityInstrumentationTestCase;
import android.widget.ListView;
import android.widget.Spinner;

public class ContactsDemoTest
  extends ActivityInstrumentationTestCase<ContactSpinners> {
  private ListView list=null;
  private Spinner spinner=null;

  public ContactsDemoTest() {
```

```
    super("com.commonsware.android.contacts.spinners",
          ContactSpinners.class);
  }

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    ContactSpinners activity=getActivity();

    list=(ListView)activity.findViewById(android.R.id.list);
    spinner=(Spinner)activity.findViewById(R.id.spinner);
  }

  public void testSpinnerCount() {
    assertTrue(spinner.getAdapter().getCount()==3);
  }

  public void testListDefaultCount() {
    assertTrue(list.getAdapter().getCount()>0);
  }
}
```

Here are the steps to making use of `ActivityInstrumentationTestCase`:

1. Extend the class to create your own implementation. Since `ActivityInstrumentationTestCase` is a generic, you need to supply the name of the activity being tested (e.g., `ActivityInstrumentationTestCase<ContactsDemo>`).
2. In the constructor, when you chain to the superclass, supply the name of the package of the activity plus the activity class itself. You can optionally supply a third parameter, a boolean indicating if the activity should be launched in touch mode or not.
3. In `setUp()`, use `getActivity()` to get your hands on your `Activity` object, already typecast to the proper type (e.g., `ContactsDemo`) courtesy of our generic. You can also at this time access any widgets, since the activity is up and running by this point.
4. If needed, clean up stuff in `tearDown()`, no different than with any other JUnit test case.
5. Implement test methods to exercise your activity. In this case, we simply confirm that the `Spinner` has three items in its drop-down list and there is at least one contact loaded into the `ListView` by

default. You could, however, use `sendKeys()` and the like to simulate user input.

If you are looking at your emulator or device while this test is running, you will actually see the activity launched on-screen. `ActivityInstrumentationTestCase` creates a true running copy of the activity. This means you get access to everything you need; on the other hand, it does mean that the test case runs slowly, since the activity needs to be created and destroyed for each test method in the test case. If your activity does a lot on startup and/or shutdown, this may make running your tests a bit sluggish.

Note that your `ActivityInstrumentationTestCase` resides in the same package as the Activity it is testing – `ContactsDemoTest` and `ContactsDemo` are both in `com.commonsware.android.database`, for example. This allows `ContactsDemoTest` to access both public and package-private methods and data members. `ContactsDemoTest` still cannot access private methods, though. This allows `ActivityInstrumentationTestCase` to behave in a white-box (or at least gray-box) fashion, inspecting the insides of the tested activities in addition to testing the public API.

Now, despite the fact that Android's own tools create an `ActivityInstrumentationTestCase` subclass for you, that class is officially deprecated. They advise using `ActivityInstrumentationTestCase2` instead, which offers the same basic functionality, with a few extras, such as being able to specify the `Intent` that is used to launch the activity being tested. This is good for testing search providers, for example.

AndroidTestCase

For tests that only need access to your application resources, you can skip some of the overhead of `ActivityInstrumentationTestCase` and use `AndroidTestCase`. In `AndroidTestCase`, you are given a `Context` and not much more, so anything you can reach from a `Context` is testable, but individual activities or services are not.

While this may seem somewhat useless, bear in mind that a lot of the static testing of your activities will come in the form of testing the layout: are the widgets identified properly, are they positioned properly, does the focus work, etc. As it turns out, none of that actually needs an `Activity` object – so long as you can get the inflated `View` hierarchy, you can perform those sorts of tests.

For example, here is an `AndroidTestCase` implementation, `ContactsDemoBaseTest`:

```
package com.commonsware.android.contacts.spinners;

import android.test.AndroidTestCase;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ListView;
import android.widget.Spinner;

public class ContactsDemoBaseTest extends AndroidTestCase {
  private ListView list=null;
  private Spinner spinner=null;
  private ViewGroup root=null;

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    LayoutInflater inflater=LayoutInflater.from(getContext());

    root=(ViewGroup)inflater.inflate(R.layout.main, null);
    root.measure(480, 320);
    root.layout(0, 0, 480, 320);

    list=(ListView)root.findViewById(android.R.id.list);
    spinner=(Spinner)root.findViewById(R.id.spinner);
  }

  public void testExists() {
    assertNotNull(list);
    assertNotNull(spinner);
  }

  public void testRelativePosition() {
    assertTrue(list.getTop()>=spinner.getBottom());
    assertTrue(list.getLeft()==spinner.getLeft());
    assertTrue(list.getRight()==spinner.getRight());
  }
}
```

Most of the complicated work is performed in `setUp()`:

1. Inflate our layout using a `LayoutInflater` and the `Context` supplied by `getContext()`
2. Measure and lay out the widgets in the inflated `View` hierarchy – in this case, we lay them out on a 480x320 screen
3. Access the individual widgets to be tested

At that point, we can test static information on the widgets, but we cannot cause them to change very easily (e.g., we cannot simulate keypresses). In the case of `ContactsDemoBaseTest`, we simply confirm the widgets exist and are laid out as expected. We could use `FocusFinder` to test whether focus changes from one widget to the next should work as expected. We could ensure our resources exist under their desired names, test to see if our fonts exist in our assets, or anything else we can accomplish with just a `Context`.

Since we are not creating and destroying activities with each test case, these tests should run substantially faster.

Other Alternatives

Android also offers various other test case base classes designed to assist in testing Android components, such as:

- `ServiceTestCase`, used for testing services, as you might expect given the name
- `ActivityUnitTestCase`, a `TestCase` that creates the `Activity` (like `ActivityInstrumentationTestCase`), but does not fully connect it to the environment, so you can supply a mock `Context`, a mock `Application`, and other mock objects to test out various scenarios
- `ApplicationTestCase`, for testing custom `Application` subclasses

Monkeying Around

Independent from the JUnit system is the Monkey.

The Monkey is a test program that simulates random user input. It is designed for "bash testing", confirming that no matter what the user does, the application will not crash. The application may have odd results – random input entered into a Twitter client may, indeed, post that random input to Twitter. The Monkey does not test to make sure that results of random input make sense; it only tests to make sure random input does not blow up the program.

You can run the Monkey by setting up your initial starting point (e.g., the main activity in your application) on your device or emulator, then running a command like this:

```
adb shell monkey -p com.commonsware.android.database -v --throttle 100 600
```

Working from right to left, we are asking for 600 simulated events, throttled to run every 100 milliseconds. We want to see a list of the invoked events (`-v`) and we want to throw out any event that might cause the Monkey to leave our application, as determined by the application's package (`-p com.commonsware.android.database`).

The Monkey will simulate keypresses (both QWERTY and specialized hardware keys, like the volume controls), D-pad/trackball moves, and sliding the keyboard open or closed. Note that the latter may cause your emulator some confusion, as the emulator itself does not itself actually rotate, so you may end up with your screen appearing in landscape while the emulator is still, itself, portrait. Just rotate the emulator a couple of times (e.g., `<Ctrl>-<F12>`) to clear up the problem.

For playing with a Monkey, the above command works fine. However, if you want to regularly test your application this way, you may need some measure of repeatability. After all, the particular set of input events that trigger your crash may not come up all that often, and without that repeatable scenario, it will be difficult to repair the bug, let alone test that the repair worked.

To deal with this, the Monkey offers the `-s` switch, where you provide a seed for the random number generator. By default, the Monkey creates its own

seed, giving totally random results. If you supply the seed, while the sequence of events is random, it is random for that seed – repeatedly using the same seed will give you the same events. If you can arrange to detect a crash and know what seed was used to create that crash, you may well be able to reproduce the crash.

There are many more Monkey options, to control the mix of event types, to generate profiling reports as tests are run, and so on. The Monkey documentation in the SDK's Developer's Guide covers all of that and more.

CHAPTER 19

Production Applications

Of course, all of this programming you have done will be a bit silly if you only have debug applications running on an emulator, or perhaps your own phone. Somewhere along the line, you may want others to run your applications as well, perhaps by buying them from you.

This chapter focuses on the steps you will need to take to have your application be distributed in a production form, through the Android Market and elsewhere. Much of the focus is on the Android Market because it is the largest and the one people tend to think about. However, along the way, we will cover other markets, other forms of distribution, and things you need to do that are relevant for any form of production distribution.

Market Theory

As noted, the Android Market is the largest and most visible marketplace for Android applications. It, therefore, will tend to set the tone that other markets either follow or specifically position themselves against.

The biggest of these is the Android Market "lifestream" model.

When somebody buys an application off of the Android Market, they are not just buying one edition of one application for one device. Rather, they are buying all editions of that application for any device they purchase and register to their Google account.

For example, the Android Market supports all of the following scenarios and more:

- The user buys a phone, buys some applications off of the Market, and then replaces their phone with another Android device
- The user buys two Android devices (a phone and a larger-screen media player) and registers both devices to the same Google account, and therefore can download applications purchased from one device on both devices
- The user buys some applications off of the Market, uninstalls them to free up space, then reinstalls them later from the Market without additional fees
- The user buys some applications off of the Market and gets all of those applications' updates as free downloads, without additional fees

This "lifestream" model is great for the user. Whether it is great or bad for you as a developer depends on your revenue model and how you view your relationship with your customers. Regardless, it is what it is, and this "lifestream" concept permeates much of the way you will use the Android Market.

Making Your Mark

Perhaps the most important step in preparing your application for production distribution is signing it with a production signing key. While mistakes here may not be immediately apparent, they can have significant long-term impacts, particularly when it comes time for you to distribute an update.

Role of Code Signing

There are many reasons why Android wants you to sign your application with a production key. Here are perhaps the top three:

1. It will help distinguish your production applications from debug versions of the same applications
2. Multiple applications signed with the same key can access each other's private files, if they are set up to use a shared user ID in their manifests
3. You can only update an application if it has a signature from the same digital certificate

The latter one is the most important for you, if you plan on offering updates of your application. If you sign version 1.0 of your application with one key, and you sign version 2.0 of your application with another key, version 2.0 will not install over top version 1.0 – it will fail with a certificate-match error.

What Happens In Debug Mode

Of course, you may be wondering how you got this far in life without worrying about keys and certificates and signatures (unless you are using Google Maps, in which case you experienced a bit of this when you got your API key).

The Android build process, whether through Ant or Eclipse, creates a debug key for you automatically. That key is automatically applied when you create a debug version of your application (e.g., `ant debug` or `ant install`). This all happens behind the scenes, so it is very possible for you to go through weeks and months of development and not encounter this problem.

In fact, the most likely place where you might encounter this problem is in a distributed development environment, such as an open source project. There, you might have encountered problem #3 from the previous section, where a debug application compiled by one team member cannot install over the debug application from another team member, since they do not share a common debug key. You may have run into similar problems just on your own if you use multiple development machines (e.g., a desktop in the

home office and a notebook for when you are on the road delivering Android developer training).

So, developing in debug mode is easy. It is mostly when you move to production that things get a bit more interesting.

Creating a Production Signing Key

To create a production signing key, you will need to use `keytool`. This comes with the Java SDK, and so it should be available to you already.

The `keytool` utility manages the contents of a "keystore", which can contain one or more keys. Each "keystore" has a password for the store itself, and keys can also have their own individual passwords. You will need to supply these passwords later on when signing an application with the key.

Here is an example of running keytool:

```
mmurphy@opti755:~$ keytool -genkey -v -keystore cw-release.keystore -alias cw-re
lease -keyalg RSA -validity 10000
```

Figure 77. Running keytool

The parameters used here are:

- `-genkey`, to indicate we want to create a new key
- `-v`, to be verbose about the key creation process
- `-keystore`, to indicate what keystore we are manipulating (`cw-release.keystore`), which will be created if it does not already exist
- `-alias`, to indicate what human-readable name we want to give the key (`cw-release`)
- `-keyalg`, to indicate what public-key encryption algorithm to be using for this key (`RSA`)
- `-validity`, to indicate how long this key should be valid, where 10,000 days or more is recommended

The length of the validity is important. Once your key expires, you can no longer use it for signing new applications, which means once the key expires, you cannot update existing Android applications. 10,000 days, presumably, is beyond the expected lifespan of this signing mechanism. Also, the Android Market requires your key to be valid beyond October 22, 2033.

If you run the above command, you will be prompted for a number of pieces of information. If you have ever created an SSL certificate, the prompts will be familiar:

```
mmurphy@opti755:~$ keytool -genkey -v -keystore cw-release.keystore -alias cw-re
lease -keyalg RSA -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Mark Murphy
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:  CommonsWare, LLC
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:  PA
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US corr
ect?
  [no]:  yes

Generating 1,024 bit RSA key pair and self-signed certificate (SHA1withRSA) with
 a validity of 10,000 days
        for: CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA,
 C=US
Enter key password for <cw-release>
        (RETURN if same as keystore password):
Re-enter new password:
[Storing cw-release.keystore]
mmurphy@opti755:~$
```

Figure 78. Results of running keytool

You will note that this is a self-signed certificate – you do not have to purchase a certificate from Verisign or anyone. These keys are for creating immutable identity, but are not for creating confirmed identity. In other

words, these certificates do not prove you are such-and-so person, but can prove that the same key signed two different APKs.

In theory, you only need to do the above steps once per business.

Signing with the Production Key

To sign an application with a production key, you must first create an unsigned version of the APK. By default (e.g., `ant debug`), you get an APK signed with the debug key. Instead, specifically build a release version (e.g., `ant release`), which should give you an `-unsigned.apk` file in your project's `bin/` directory.

Next, to apply the key, you will use the `jarsigner` tool. Like `keytool`, `jarsigner` comes with the Java SDK, and so you should already have it on your development machine.

Here is an example of running `jarsigner`:

```
mmurphy@opti755:~/stuff/CommonsWare/projects/vidtry$ jarsigner -verbose -keystor
e ~/cw-release.keystore bin/vidtry-unsigned.apk cw-release
```

Figure 79. Running jarsigner

In this case, the parameters supplied are:

- `-verbose`, to explain what is going on as the program runs
- `-keystore`, to indicate where the keystore that contains the production key resides (`~/cw-release.keystore`)
- the path to the APK to sign (`bin/vidtry-unsigned.apk`)
- the alias of the key in the keystore to apply (`cw-release`)

At this point, `jarsigner` will prompt you for the keystore's password (and the key's password if you supplied a distinct password for it to `keytool`), then it will apply the signature:

```
mmurphy@opti755:~/stuff/CommonsWare/projects/vidtry$ jarsigner -verbose -keystor
e ~/cw-release.keystore bin/vidtry-unsigned.apk cw-release
Enter Passphrase for keystore:
   adding: META-INF/MANIFEST.MF
   adding: META-INF/CW-RELEA.SF
   adding: META-INF/CW-RELEA.RSA
  signing: res/drawable/btn_media_player.9.png
  signing: res/drawable/btn_media_player_disabled.9.png
  signing: res/drawable/btn_media_player_disabled_selected.9.png
  signing: res/drawable/btn_media_player_pressed.9.png
  signing: res/drawable/btn_media_player_selected.9.png
  signing: res/drawable/ic_media_pause.png
  signing: res/drawable/ic_media_play.png
  signing: res/drawable/media_button_background.xml
  signing: res/layout/main.xml
  signing: AndroidManifest.xml
  signing: resources.arsc
  signing: classes.dex
mmurphy@opti755:~/stuff/CommonsWare/projects/vidtry$
```

Figure 80. Results of running jarsigner

Next, you should test the signature by `jarsigner -verify -verbose -certs` on the same APK file, which now has a signature. You will get output akin to:

```
 1090 Sat Aug 08 13:56:38 EDT 2009 META-INF/MANIFEST.MF
 1211 Sat Aug 08 13:56:38 EDT 2009 META-INF/CW-RELEA.SF
 946 Sat Aug 08 13:56:38 EDT 2009 META-INF/CW-RELEA.RSA
sm 1683 Sat Aug 08 13:54:46 EDT 2009 res/drawable/btn_media_player.9.png

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 743 Sat Aug 08 13:54:46 EDT 2009 res/drawable/btn_media_player_disabled.9.png

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 1030 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player_disabled_selected.9.png

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 1220 Sat Aug 08 13:54:46 EDT 2009 res/drawable/btn_media_player_pressed.9.png

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 1471 Sat Aug 08 13:54:46 EDT 2009
res/drawable/btn_media_player_selected.9.png
```

```
 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 576 Sat Aug 08 13:54:46 EDT 2009 res/drawable/ic_media_pause.png

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 938 Sat Aug 08 13:54:46 EDT 2009 res/drawable/ic_media_play.png

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 1176 Sat Aug 08 13:54:46 EDT 2009 res/drawable/media_button_background.xml

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 2668 Sat Aug 08 13:54:46 EDT 2009 res/layout/main.xml

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 1368 Sat Aug 08 13:54:46 EDT 2009 AndroidManifest.xml

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 2888 Sat Aug 08 13:54:46 EDT 2009 resources.arsc

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]

sm 16860 Sat Aug 08 13:54:46 EDT 2009 classes.dex

 X.509, CN=Mark Murphy, OU=Unknown, O="CommonsWare, LLC", L=Unknown, ST=PA, C=US
 [certificate is valid from 8/8/09 1:49 PM to 12/24/36 12:49 PM]


 s = signature was verified
 m = entry is listed in manifest
 k = at least one certificate was found in keystore
 i = at least one certificate was found in identity scope

jar verified.
```

In particular, you want to make sure that the name of the key is what you expect and is not "Android Debug", which would indicate the APK was signed with the debug key instead of the production key.

At this point, you should also rename the APK, at least to remove the now-erroneous `-unsigned` portion of the filename.

Now, you have a production-signed APK, ready for distribution...or, hopefully, ready for more testing, *then* distribution.

Two Types of Key Security

There are two facets to securing your production key that you need to think about:

1. You need to make sure nobody steals your production keystore and its password. If somebody does, they could publish replacement versions of your applications – since they are signed with the same key, Android will assume the replacements are legitimate.
2. You need to make sure you do not lose your production keystore and its password. Otherwise, even *you* will be unable to publish replacement versions of your applications.

For solo developers, the latter scenario is more probable. There already have been cases where developers had to rebuild their development machine and wound up with new keys, locking themselves out from updating their own applications. As with everything involving computers, having a solid backup regimen is highly recommended.

For teams, the former scenario may be more likely. If more than one person needs to be able to sign the application, the production keystore will need to be shared, possibly even stored in the revision control system for the project. The more people who have access to the keystore, the more likely it is somebody will wind up doing something evil with it. This is particularly true for projects with public revision control systems, such as open source projects – developers might not think of the implications of putting the production keystore out for people to access.

Related Keys

Switching from debug to production keys may have additional ramifications for your application.

For example, if you are integrating Google Maps, you no doubt obtained a Maps API key to use with your application. As it turns out, you most likely got an API that corresponds to your debug signing key. For production, you will need a different Maps API key, one that corresponds to your production signing key.

This will likely be a significant pain for you, because the Maps API key goes in the source code, meaning the source code is now dependent upon how it is being signed. You may wish to apply some automation to this, such as building custom Ant tasks that switches between debug and production Maps API keys in your source code depending on how you are building the project.

In principle, the same concept may extend to other keys for other Android development add-ons, though none are known at this time.

Get Ready To Go To Market

While being able to sign your application reliably with a production key is necessary for publishing a production application, it is not sufficient. Particularly for the Android Market, there are other things you must do, or should do, as part of getting ready to release your application.

Versioning

As was described in *The Busy Coder's Guide to Android Development*, you need to supply `android:versionCode` and `android:versionName` attributes in your `<manifest>` element in your `AndroidManifest.xml` file. The value of `android:versionName` is what users and prospective users will see in terms of the label associated with your application version (e.g., "1.0.1", "System V",

"Loquacious Llama"). More important, though, is the value of `android:versionCode`, which needs to be an integer increasing with each release – that is how Android tells whether some edition of your APK is an upgrade over what the user currently has.

You also need to specify the `minSdkVersion` of your application:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.search">
  <uses-sdk minSdkVersion="2" />
  ...
</manifest>
```

If you are using Android 1.5 features, or think you might be, the safest value to choose is `3`.

Package Name

You also need to make sure that your package name – as denoted by the `package` attribute of the root `<manifest>` element – is going to be unique. If somebody tries downloading your application onto their device, and some other application is already installed with that same package name, your application will fail to install.

Since the manifest's package name also provides the base Java package for your project, and since you hopefully named your Java packages with something based off of a domain name you own or something else demonstrably unique, this should not cause a huge problem.

Icon and Label

Your `<application>` element needs to specify `android:icon` and `android:name` attributes, to supply the name and icon that will be associated with the application in the My Applications list on the device and related screens. Your activities will inherit the icon if they do not specify icons of their own.

If you have graphic design skills, the Android developer site has guidelines for creating icons that will match other icons in the system. If not, just supply a 48x48 pixel image with appropriate transparency (if applicable).

Logging

In production, try to minimize unnecessary logging, particularly at low logging levels (e.g., debug). Remember that even if Android does not actually log the information, whatever processing is involved in making the `Log.d()` call will still be done, unless you arrange to skip the processing somehow. You could outright delete the extraneous logging calls, or wrap them in an `if()` test:

```
if (!SomeClass.IS_DEVELOPMENT) {
  Log.d(TAG, "This is what happened");
}
```

Here, `IS_DEVELOPMENT` is a `public static final boolean` value, `true` during development, `false` as you work your way to production. Whether you adjust the definition by hand or by automating the build process is up to you. But, when `IS_DEVELOPMENT` is `false`, any work that would have been done to build up the actual `Log` invocation will be skipped, saving CPU cycles and battery life.

Conversely, error logs become even more important in production. Sometimes, you have difficult reproducing bugs "in the lab" and only encounter them on customer devices. Being able to get stack traces from those devices could make a major difference in your ability to get the bug fixed rapidly.

First, in addition to your regular exception handlers, consider catching everything those handlers miss, notably runtime exceptions:

```
Thread.setDefaultUncaughtExceptionHandler(onBlooey);
```

This will route all uncaught exceptions to an `onBlooey` handler:

```
private Thread.UncaughtExceptionHandler onBlooey=
  new Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread thread, Throwable ex) {
    Log.e(TAG, "Uncaught exception", ex);
  }
};
```

There, you can log it, raise a dialog if appropriate, etc.

Then, offer some means to get your logs off the device and to you, via email or a Web service. Some Android analytics firms, like Flurry, offer exception stack trace collection as part of their service. There are also open source projects that support this feature, such as android-remote-stacktrace.

Testing

As always, testing, particularly acceptance testing, is important.

Bear in mind that the act of creating the production signed version of your application could introduce errors, such as having the wrong Google Maps API key. Hence, it is important to do user-level testing of your application after you sign, not just before you sign, in case the act of signing messed things up. After all, what you are shipping to those users is the production signed edition – you do not want your users tripping over obvious flaws.

As you head towards production, also consider testing in as many distinct environments as possible, such as:

- Trying more than one device, particularly if you can get devices with different display sizes
- If you rely on the Internet, try your application with WiFi, with 3G, with EDGE/2G, and with the Internet unavailable
- If you rely on GPS, try your application with GPS disabled, GPS enabled and working, and GPS enabled but not available (e.g., underground)

EULA

End-user license agreements – EULAs – are those long bits of legal prose you are supposed to read and accept before using an application, Web site, or other protected item. Whether EULAs are enforcible in your jurisdiction is between you and your qualified legal counsel to determine.

In fact, many developers, particularly of free or open source applications, specifically elect not to put a EULA in their applications, considering them annoying, pointless, or otherwise bad.

However, the Android Market developer distribution agreement has one particular clause that might steer you towards having a EULA:

> *You agree that if you use the Market to distribute Products, you will protect the privacy and legal rights of users. If the users provide you with, or your Product accesses or uses, user names, passwords, or other login information or personal information, you must make the users aware that the information will be available to your Product, and you must provide legally adequate privacy notice and protection for those users...**But if the user has opted into a separate agreement with you that allows you or your Product to store or use personal or sensitive information directly related to your Product (not including other products or applications) then the terms of that separate agreement will govern your use of such information.***

(emphasis added)

Hence, if you are concerned about being bound by what Google thinks appropriate privacy is, you may wish to consider a EULA just to replace their terms with your own.

Unfortunately, having a EULA on a mobile device is particularly annoying to users, because EULAs tend to be long and screens tend to be short.

Again, please seek professional legal assistance on issues regarding EULAs.

To Market, To Market

And now, the moment you have been waiting for: putting your application on the Android Market!

Of course, you may have to wait a little bit longer, depending on where you live, how much you like reading legal agreements, and so on.

Here is what you need to do to get your application on the Market.

Google Checkout

Google Checkout is Google's answer to PayPal. More importantly for Android developers, as of the time of this writing, it is the sole option for charging users from the Android Market. Of course, if you are distributing your applications for free, this is not an issue.

If you do intend to charge, though, you need to go through the process to get a merchant account with Google Checkout. Basically, this integrates Google Checkout with a checking account of yours, so purchases can be deposited in your account as they occur. It also helps to validate you as a business.

Note that Google Checkout is only available in certain countries. Both you (as the developer) and your customer needs to be in Checkout-capable countries for payment to work. This means, among other things, that if you do not reside in a country supported by Google Checkout, you cannot charge for applications in the Android Market.

All of this is not free. 30% is taken off the top as a fee to the mobile carriers who distribute and support the Android Market. Google Checkout may also charge additional fees, particularly for cross-border purchases, though it is unclear if that is happening at present.

Terms and Conditions

As the author is fond of saying: "I am not a lawyer, nor do I play one on TV".

That being said, there are a number of aspects of the Android Market terms and conditions that you should examine closely to see if they will pose a problem for you, such as:

- "All fees received by Developers for Products distributed via the Market must be processed by the Market's Payment Processor."
- "Products that cannot be previewed by the buyer (such as applications): You authorize Google to give the buyer a full refund of the Product price if the buyer requests the refund within 48 hours after purchase."
- "Except in cases when multiple disputes are initiated by a user with abnormal dispute history, billing disputes received by Payment Processor for Products sold for less than $10 may be automatically charged back to the Developer, in addition to any handling fees charged by the Payment Processor."
- "Users are allowed unlimited reinstalls of each application distributed via the Market."
- "You agree that you will not engage in any activity with the Market, including the development or distribution of Products, that interferes with, disrupts, damages, or accesses in an unauthorized manner the devices, servers, networks, or other properties or services of any third party including, but not limited to, Android Users, Google or any mobile network operator."
- "You may not use customer information obtained from the Market to sell or distribute Products outside of the Market."
- "You may not use the Market to distribute or make available any Product whose primary purpose is to facilitate the distribution of Products outside of the Market."

Some of those terms are in support of the "lifestream" model that the Android Market employs. Others are not. Whether any of them will cause you difficulty is for you and qualified legal counsel to determine.

Data Collection

Putting your application on the Market is a matter of signing up for the Market (and incurring a $25 fee to do so), then uploading and describing the APKs you wish to distribute. It helps, though, if you determine how you are going to describe those applications before you find yourself confronted with the Android Market upload form.

Here is what you will need to provide to Google as part of uploading an Android Market application:

Images

You can supply up to two screenshots of your application, in HVGA (480x320) or WVGA854 (854x480) portrait size, as PNG or JPEG files. These will be scaled down by the Market for display on the device as thumbnails, and the user can tap the thumbnail to see a larger (but not quite 1:1 in terms of pixel size) rendition of the screenshot.

You can also supply up to two "promotional graphics", as 180x120 PNG or JPEG files. What exactly a "promotional graphic" is supposed to be is unclear, but you could use this for a logo, or a zoomed-in fragment of another screenshot, etc.

Your goal behind the screenshots and graphics should be to demonstrate to the user the polish of your user interface, and perhaps a bit on how they can use it – though that is probably only going to be obvious if they tap on the thumbnail to view the full screenshot.

Title

Here, you have 30 characters to name your application. This will be what prospective customers see in the Android Market application when they browse for applications, either in a category or via a search.

Since this is your first chance to grab the prospect's attention, try to maximize your use of these 30 characters. If your application name is short, consider using a subtitle. For example, if you created an application that tries to use the vibration motor to create a massager, rather than just having a title of, say, "Relaxon", consider "Relaxon: Custom Massager" or "Relaxon: Ease Muscle Tension". Here, you provide the prospect with a tidbit of additional information beyond a probably opaque name, making it a bit more likely that they will click your entry and read your description.

Note that you should not put the version number in the title in most cases. For one, it takes up space you might use for more useful information for prospects. For another, the application description screen – where the prospect goes after clicking your title – will already show your version number as pulled from your application's manifest. It is possible that for a major release, it would be worth the space to add the version number, but probably not for incremental updates.

You have the option of supporting multiple languages for your Market entry. If you go this route, the user will see your entry in their own language, if you and the Market both support it. If you specified that you wanted to support multiple languages in the Market, you need to provide a title in each language. Be careful with this, though – you may not want to try adding listings in languages that your application itself does not support, as users may consider that a "bait and switch" tactic (sell it in German, ship English).

Description

In addition to the really short title, you can provide a really short description. Specifically, you have 325 characters to explain to the prospect what it is that makes your application worth getting.

Your description has two missions:

1. Help people find your application, by containing likely keywords that users might search on that pertain to your application. For example, if you are distributing a "calorie counter" application, you might try to work in words like "weight", "diet", "exercise", "health", and such into your application description, so it will come up in search results for those terms.
2. Help convince those who find your application that it is worth buying.

While search will be limited to the 355 combined title and description characters, you can also leverage a Web site to help convince people to consider your application. Hence, all else being equal, consider emphasizing keywords more, and include the shortest possible URL that can point to a Web page about your product. That Web page, of course, is not subject to Android Market limitations, so you can include screenshots, demos, testimonials, and the like. However, since users cannot click on the URL in the description, it needs to be short and simple for them to type into a Web browser.

Of course, just having a description that is a list of keywords will look silly and probably cut into sales, so you need to strike a balance between emphasizing keywords and having a professional-looking description.

One convention that has emerged is to use the tail end of the description to explain what has been added in an updated version of your application.

As with the title, if you specified that you wanted to support multiple languages in the Market, you need to provide a description in each language.

There is also a separate field for "promo text", up to 80 characters. It is unclear where this prose will appear inside the Android Market.

Application Type and Category

You may choose where you want your program to be listed among the available Android Market categories. As with the on-device Android Market application, you first choose "Applications" or "Games", then choose the specific category. Note that you can only be in one category...at a time.

It appears that you can elect to re-categorize your application after initial publication. Hence, you might consider running what marketers call "A/B testing" – after a month or so, start switching your application to other likely categories, and track sales on each for a while. Eventually, you will find the category that yields the most sales. This, of course, assumes your application could reasonably appear in more than one category.

Pricing

You can either stipulate a price for your application, or have it be free. To have a price, you need to have a Google Checkout merchant account, as described above, and that may preclude you from setting a price if merchant accounts are not available in your country.

Note that you cannot switch from free to non-free, so be sure you want a free application before choosing to make it be free.

Also note that the price you provide will be in your own national currency and will show up in that currency for prospective buyers. This causes a bit of confusion, as many people are not used to dealing with other currencies.

Copy Protection

The Android Market offers copy protection, of at least a minor sort. It attempts to make it difficult for somebody to copy the game off of one

device and onto another one owned by somebody else. Its effectiveness is questionable, considering that "rooting" Android devices is commonplace and "rooted" Android devices can bypass the copy protection with ease.

If that were not bad enough, the Android Market copy protection:

- Limits distribution, as users of the ADP1 developer phone cannot obtain copy protected applications off of the Market
- Had a round of problems early on where upgrades of copy-protected applications seemed to be unreliable, though complaints about this seem to have subsided
- Is irreversible: once you make the selection for your application (copy-protected or not), you are stuck with that decision for all updates

Locations for Distribution

Independent from your selection of languages is a selection of locations where your application should be distributed. You can either check specific countries, or check "All Current and Future Locations" for maximum coverage. By clicking the name of a country, you may also be able to filter based upon specific carrier (e.g., in the US, you could distribute to T-Mobile customers but not Verizon customers).

Many developers are safe with the "All Current and Future Locations" selection. Here are some reasons you may elect to constrain distribution to only a subset of locations:

- You are concerned that your application may violate some nations' laws, and so you restrict distribution to known safe venues
- You do not wish to pay the additional fee for international sales that Google Checkout imposes
- You are concerned about your ability to provide the desired level of technical support on a global basis

- You are concerned that your application may violate some carriers' terms of service, so you restrict your distribution to carriers you have relationships with or otherwise deem safe

Just remember that if you do not check "All Current and Future Locations" that you should check back in your Developer Console periodically to see if there are new location or carrier options available to you.

Contact Information

You will need to provide a contact name, email address, and phone number. As the Android Market agreement indicates, this information is made publicly available. Hence, giving out your home phone number or your personal email address may not be the best option.

For email addresses, consider getting a dedicated account for your Android application uses, or at least consider using an alias if your mail provider offers it.

For phone numbers, consider setting up an inexpensive alternative number, such as Skype, Google Voice, onSIP, etc. You can either elect to try answering calls or have them all roll to voicemail, finding out about messages via email and returning the calls as needed.

Pulling Distribution

If you decide you do not want to have your application published on the Market, you can unpublish it at any time. Just go to your Android Market Developer Console, click on the application in question, and click the Unpublish link at the bottom of the page. Your application will be removed from distribution within minutes.

Note that your application will still remain in the system, just not in the public Market. So, if the reason for pulling distribution was temporary (e.g., major bug needing a fix), you can republish again later, with a new APK as needed.

Market Filters

Not every application will be visible on every device, even if that device has the Android Market. Here are some known, expected filters that are in place:

- Applications written using Android 1.5 or earlier SDKs will not appear in the Market for QVGA devices, such as the HTC Tattoo.
- Applications that require certain hardware (e.g., camera) will not appear in the Market for devices that lack that hardware
- Paid appilcations will not appear in the Market running on a developer phone (e.g., ADP1, ADP2)

Going Wide

The Android Market is not the only answer for distributing your applications. For some people, it may not even be the best one. After all:

1. Not all Android devices will have Android Market, particularly those whose manufacturers are simply using the Android open source tree rather than signing any sort of deal with Google to get proprietary applications like Android Market
2. Not all users can use Android Market. For example, owners of the ADP1 cannot obtain copy-protected applications from the Android Market
3. Not all developers can sell via the Android Market, only those in select locations, in part due to the dependence upon Google Checkout
4. Android Market's only current payment option is Google Checkout, which some consumers will not wish to use, or cannot use because they lack the payment mechanisms (e.g., credit card) that Google Checkout requires
5. Android Market's terms and conditions may contain terms that developers are unwilling to accept

6. Android Market takes 30% off the top from the developer's take, which some developers may prefer to avoid
7. Some carriers and/or device manufacturers may elect to run their own markets for control purposes, or to capture more revenue (e.g., more than 30%), or to support other languages, or other reasons
8. Considering all the complaints that other firms have gotten with their one-app-store-to-rule-the-world tactics, it seems to be in Android's best interests to have a vibrant ecosystem of competing markets
9. Carriers have already exerted control over the Android Market's contents, banning tethering applications and such, while independent markets may not have similar restrictions

As of the time of this writing, the leading alternative Android application markets – AndAppStore, SlideME, Handango, etc. – are all fairly small and under-marketed compared to the Android Market. That situation is likely to shift, as devices start to ship with other markets in addition to, or perhaps instead, the Android Market.

Keep tabs on Android news, and as new markets arise, consider the likelihood that they will get visibility. For example, any market application distributed on devices is liable to get some attention from purchasers of those devices. You can get a bit of a first-mover advantage if you put a quality application on a new market early, in hopes of holding onto top popularity ratings on that market quicker and for longer.

Click Here To Download

Of course, there is nothing to say you have to use any of these markets. You are welcome to distribute Android applications yourself, through your Web site. That might be useful for:

- Free applications, in addition to listing them in markets
- Internal distribution within a business, via a company intranet, for applications not destined for public use

- Implementing your own purchasing system that does not line up with existing models, such as a subscription-based library of applications

The minimum you need is to have the Android APK MIME type configured on your Web server. Then, if somebody clicks on a link to your APK on your site, Android will know to route the download to the installation engine. The Android application MIME type is `application/vnd.android.package-archive`, and you will need to set that up as appropriate for your Web server. For example, for `nginx`, you simply need to add the following as another entry in your `mime.types` file:

```
application/vnd.android.package-archive  apk;
```

Keyword Index

Class.....

Command.....

Constant.....

Method.....

Property.....