The Busy Coder's Guide to
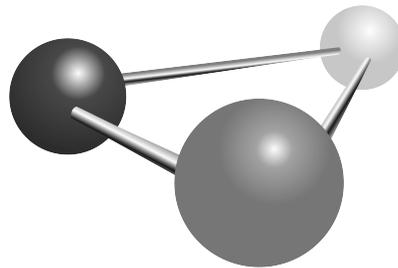
Advanced Android™ Development

Mark L. Murphy

CommonsWare

The Busy Coder's Guide to Advanced Android Development

by Mark L. Murphy

CommonsWare

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

Table of Contents

---

v

Welcome to the Warescription!

We hope you enjoy this ebook and its updates – subscribe to the Warescription newsletter on the Warescription site to learn when new editions of this book, or other books, are available.

All editions of CommonsWare titles, print and ebook, follow a software-style numbering system. Major releases (1.0, 2.0, etc.) are available in both print and ebook; minor releases (0.1, 0.9, etc.) are available in ebook form for Warescription subscribers only. Releases ending in .9 are "release candidates" for the next major release, lacking perhaps an index but otherwise being complete.

Personalized materials that you purchase as part of a Warescription are for the your use only. That means that while you are welcome to make copies as needed (e.g., home use, office use, mobile device use) for your own use, you are not welcome to make copies available to other people or organizations.

If CommonsWare determines that your personalized materials have been illicitly copied, your account will be immediately suspended. 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, or follow the links contained in the book's preface.

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`.

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 commonsguy Twitter feed

- The CommonsBlog

- The Warescription newsletter, which you can subscribe to off of your Warescription page

Getting Help

If you have questions about the book examples, visit StackOverflow and ask a question, tagged with **android** and **commonsware**.

If you have general Android developer questions, visit StackOverflow and ask a question, tagged with **android** (and any other relevant tags, such as **java**).

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.

Errata and the Book Bug Bounty

Books updated as frequently as CommonsWare's inevitably have bugs. Flaws. Errors. Even the occasional gaffe, just to keep things interesting. You

will find a list of the known bugs on the errata page on the CommonsWare Web site.

But, there are probably even more problems. If you find one, please 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 book's GitHub repository. 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. Some of 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. For Ant-based builds, 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.

The book sometimes shows entire source files, and occasionally shows only fragments of source files that are relevant to the current discussion. The book rarely shows each and every file for the sample projects. Please refer to the source code repository for the full source to any of the book samples.

Some samples will be from other Android projects, such as the CommonsWare Android Components. Those chapters will include links to their respective source code repositories.

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

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

This edition of this book will be available under the aforementioned Creative Commons license on December 1, **2015**. 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-ShareAlike 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

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

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

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

```
package com.commonsware.android.geoweb;
```

```java
import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.webkit.WebView;
import org.json.JSONException;
import org.json.JSONObject;

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,
                                             onLocation);
  }

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

  LocationListener onLocation=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) {
```

Special Creative Commons BY-NC-SA 3.0 License Edition

```
      // required for interface, not used
    }

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

  public class Locater {
    public String getLocation() throws JSONException {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

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

      JSONObject json=new JSONObject();

      json.put("lat", loc.getLatitude());
      json.put("lon", loc.getLongitude());

      return(json.toString());
    }
  }
}
```

This looks a bit like some of the `WebView` examples in *The Busy Coder's Guide to Android Development*'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 encoded in JSON

3. It uses `addJavascriptInterface()` to expose a `Locater` instance under the name `locater` to the Web content loaded in the `WebView`

The Locater API uses JSON to return both a latitude and a longitude at the same time. We are limited to using data types that are in common between Javascript and Java, so we cannot pass back the `Location` object we get from the `LocationManager`. Hence, we convert the key `Location` data into a simple JSON structure that the Javascript on the Web page can parse.

4

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
<html>
<head>
<title>Android GeoWebOne Demo</title>
<script language="javascript">
  function whereami() {
    var location=JSON.parse(locater.getLocation());

    document.getElementById("lat").innerHTML=location.lat;
    document.getElementById("lon").innerHTML=location.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()">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:
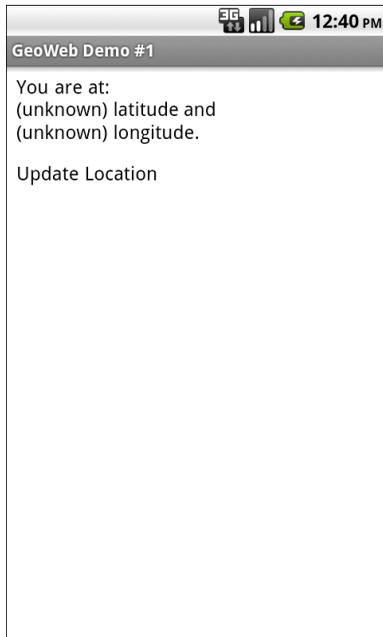
Figure 1. The GeoWebOne sample application, as initially launched

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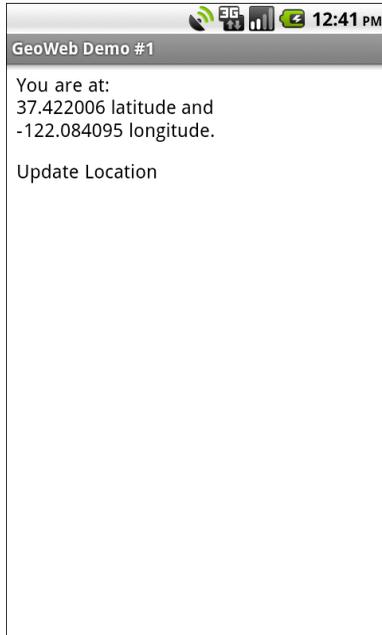
Figure 2. The GeoWebOne sample application, after clicking the Update Location link

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:

```
package com.commonsware.android.geoweb2;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.webkit.WebView;
import org.json.JSONException;
import org.json.JSONObject;

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,
                                           onLocation);
}

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

LocationListener onLocation=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 String getLocation() throws JSONException {
    Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

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

    JSONObject json=new JSONObject();

    json.put("lat", loc.getLatitude());
```

```
        json.put("lon", loc.getLongitude());

        return(json.toString());
      }
    }
}
```

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
<html>
<head>
<title>Android GeoWebTwo Demo</title>
<script language="javascript">
  function whereami(lat, lon) {
    document.getElementById("lat").innerHTML=lat;
    document.getElementById("lon").innerHTML=lon;
  }

  function pull() {
    var location=JSON.parse(locater.getLocation());

    whereami(location.lat, location.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="pull()">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.

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.

This chapter starts with a discussion of the various ways you can go about creating custom View classes. It then moves into an examination of `ColorMixer`, a composite widget, made up of several other widgets within a layout.

Note that the material in this chapter is focused on creating custom `View` classes for use within a single Android project. If your goal is to truly create reusable custom widgets, you will also need to learn how to package them so they can be reused – that is covered in a later chapter.

Pick Your Poison

You have five major options for creating a custom `View` class.

First, your "custom `View` class" might really only be custom `Drawable` resources. Many widgets can adopt a radically different look and feel just

with replacement graphics. For example, you might think that these toggle buttons from the Android 2.1 Google Maps application are some fancy custom widget:

Figure 3. Google Maps navigation toggle buttons

In reality, those are just radio buttons with replacement images.

Second, your custom `View` class might be a simple subclass of an existing widget, where you override some behaviors or otherwise inject your own logic. Unfortunately, most of the built-in Android widgets are not really designed for this sort of simple subclassing, so you may be disappointed in how well this particular technique works.

Third, your custom `View` class might be a composite widget – akin to an activity's contents, complete with layout and such, but encapsulated in its own class. This allows you to create something more elaborate than you will just by tweaking resources. We will see this later in the chapter with `ColorMixer`.

Fourth, you might want to implement your own layout manager, if your GUI rules do not fit well with `RelativeLayout`, `TableLayout`, or other built-in containers. For example, you might want to create a layout manager that more closely mirrors the "box model" approach taken by XUL and Flex, or you might want to create one that mirrors Swing's `FlowLayout` (laying widgets out horizontally until there is no more room on the current row, then start a new row).

Finally, you might want to do something totally different, where you need to draw the widget yourself. For example, the `ColorMixer` widget uses `SeekBar` widgets to control the mix of red, blue, and green. But, you might create a `ColorWheel` widget that draws a spectrum gradient, detects touch events, and lets the user pick a color that way.

Some of these techniques are fairly simple; others are fairly complex. All share some common traits, such as widget-defined attributes, that we will see throughout the remainder of this chapter.

Colors, Mixed How You Like Them

The classic way for a user to pick a color in a GUI is to use a color wheel like this one:

Figure 4. A color wheel from the API samples

There is even code to make one in the API samples.

However, a color wheel like that is difficult to manipulate on a touch screen, particularly a capacitive touchscreen designed for finger input. Fingers are great for gross touch events and lousy for selecting a particular color pixel.

Another approach is to use a mixer, with sliders to control the red, green, and blue values:
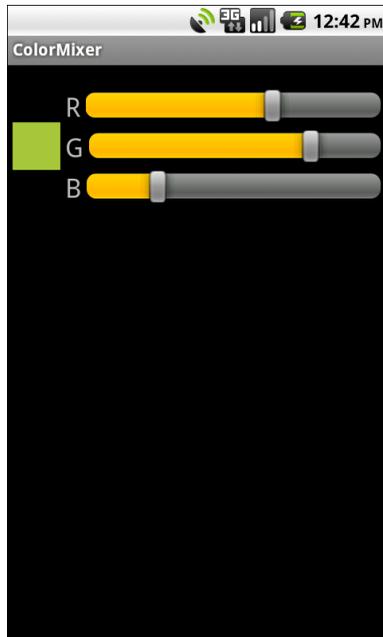
Figure 5. The ColorMixer widget, inside an activity

That is the custom widget you will see in this section, based on the code in the Views/ColorMixer project.

The Layout

ColorMixer is a composite widget, meaning that its contents are created from other widgets and containers. Hence, we can use a layout file to describe what the widget should look like.

The layout to be used for the widget is not that much: three SeekBar widgets (to control the colors), three TextView widgets (to label the colors), and one plain View (the "swatch" on the left that shows what the currently selected color is). Here is the file, found in res/layout/mixer.xml in the Views/ColorMixer project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <View android:id="@+id/swatch"
```

```
    android:layout_width="40dip"
    android:layout_height="40dip"
    android:layout_alignParentLeft="true"
    android:layout_centerVertical="true"
    android:layout_marginLeft="4dip"
  />
  <TextView android:id="@+id/redLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/swatch"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:text="@string/red"
    android:textSize="10pt"
  />
  <SeekBar android:id="@+id/red"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/redLabel"
    android:layout_toRightOf="@id/redLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
  <TextView android:id="@+id/greenLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/redLabel"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/green"
    android:textSize="10pt"
  />
  <SeekBar android:id="@+id/green"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/greenLabel"
    android:layout_toRightOf="@id/greenLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
  <TextView android:id="@+id/blueLabel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/greenLabel"
    android:layout_toRightOf="@id/swatch"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:text="@string/blue"
    android:textSize="10pt"
  />
  <SeekBar android:id="@+id/blue"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
```

```
    android:layout_alignTop="@id/blueLabel"
    android:layout_toRightOf="@id/blueLabel"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
</merge>
```

One thing that is a bit interesting about this layout, though, is the root element: `<merge>`. A `<merge>` layout is a bag of widgets that can be poured into some other container. The layout rules on the children of `<merge>` are then used in conjunction with whatever container they are added to. As we will see shortly, `ColorMixer` itself inherits from `RelativeLayout`, and the children of the `<merge>` element will become children of `ColorMixer` in Java. Basically, the `<merge>` element is only there because XML files need a single root – otherwise, the `<merge>` element itself is ignored in the layout.

The Attributes

Widgets usually have attributes that you can set in the XML file, such as the `android:src` attribute you can specify on an `ImageButton` widget. You can create your own custom attributes that can be used in your custom widget, by creating a `res/values/attrs.xml` file containing `declare-styleable` resources to specify them.

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

```
<resources>
  <declare-styleable name="ColorMixer">
    <attr name="initialColor" format="color" />
  </declare-styleable>
</resources>
```

The `declare-styleable` element describes what attributes are available on the widget class specified in the name attribute – in our case, `ColorMixer`. Inside `declare-styleable` you can have one or more `attr` elements, each indicating the `name` of an attribute (e.g., `initialColor`) and what data `format` the attribute has (e.g., `color`). 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 only one attribute: `initialColor`, which will hold the initial color we want the mixer set to when it first appears.

There are many possible values for the `format` attribute in an `attr` element, including:

- `boolean`
- `color`
- `dimension`
- `float`
- `fraction`
- `integer`
- `reference` (which means a reference to another resource, such as a `Drawable`)
- `string`

You can even support multiple formats for an attribute, by separating the values with a pipe (e.g., `reference|color`).

The Class

Our `ColorMixer` class, a subclass of `RelativeLayout`, will take those attributes and provide the actual custom widget implementation, for use in activities.

Constructor Flavors

A `View` has three possible constructors:

- One takes just a `Context`, which usually will be an `Activity`
- One takes a `Context` and an `AttributeSet`, the latter of which represents the attributes supplied via layout XML
- One takes a `Context`, an `AttributeSet`, and the default style to apply to the attributes

If you are expecting to use your custom widget in layout XML files, you will need to implement the second constructor and chain to the superclass. If you want to use styles with your custom widget when declared in layout XML files, you will need to implement the third constructor and chain to the superclass. If you want developers to create instances of your View class in Java code directly, you probably should implement the first constructor and, again, chain to the superclass.

In the case of ColorMixer, all three constructors are implemented, eventually routing to the three-parameter edition, which initializes our widget. Below, you will see the first two of those constructors, with the third coming up in the next section:

```
public ColorMixer(Context context) {
  this(context, null);
}

public ColorMixer(Context context, AttributeSet attrs) {
  this(context, attrs, 0);
}
```

Using the Attributes

The ColorMixer has a starting color – after all, the SeekBar widgets and swatch View have to show something. Developers can, if they wish, set that color via a setColor() method:

```
public void setColor(int color) {
  red.setProgress(Color.red(color));
  green.setProgress(Color.green(color));
  blue.setProgress(Color.blue(color));
  swatch.setBackgroundColor(color);
}
```

If, however, we want developers to be able to use layout XML, we need to get the value of initialColor out of the supplied AttributeSet. In ColorMixer, this is handled in the three-parameter constructor:

```
public ColorMixer(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);

    ((Activity)getContext())
```

```
      .getLayoutInflater()
      .inflate(R.layout.mixer, this, true);

  swatch=findViewById(R.id.swatch);

  red=(SeekBar)findViewById(R.id.red);
  red.setMax(0xFF);
  red.setOnSeekBarChangeListener(onMix);

  green=(SeekBar)findViewById(R.id.green);
  green.setMax(0xFF);
  green.setOnSeekBarChangeListener(onMix);

  blue=(SeekBar)findViewById(R.id.blue);
  blue.setMax(0xFF);
  blue.setOnSeekBarChangeListener(onMix);

  if (attrs!=null) {
    TypedArray a=getContext()
              .obtainStyledAttributes(attrs,
                                R.styleable.ColorMixer,
                                0, 0);

    setColor(a.getInt(R.styleable.ColorMixer_initialColor,
                  0xFFA4C639));
    a.recycle();
  }
}
```

There are three steps for getting attribute values:

1. Get a TypedArray conversion of the AttributeSet by calling obtainStyledAttributes() on our Context, supplying it the AttributeSet and the ID of our styleable resource (in this case, R.styleable.ColorMixer, since we set the name of the declare-styleable element to be ColorMixer)

2. Use the TypedArray to access specific attributes of interest, by calling an appropriate getter (e.g., getInt()) with the ID of the specific attribute to fetch (R.styleable.ColorMixer_initialColor)

3. Recycle the TypedArray when done, via a call to recycle(), to make the object available to Android for use with other widgets via an object pool (versus creating new instances every time)

Note that the name of any given attribute, from the standpoint of TypedArray, is the name of the styleable resource (R.styleable.ColorMixer)

concatenated with an underscore and the name of the attribute itself (`_initialColor`).

In `ColorMixer`, we get the attribute and pass it to `setColor()`. Since `getColor()` on `AttributeSet` takes a default value, we supply some stock color that will be used if the developer declined to supply an `initialColor` attribute.

Also note that our `ColorMixer` constructor inflates the widget's layout. In particular, it supplies `true` as the third parameter to `inflate()`, meaning that the contents of the layout should be added as children to the `ColorMixer` itself. When the layout is inflated, the `<merge>` element is ignored, and the `<merge>` element's children are added as children to the `ColorMixer`.

Saving the State

Similar to activities, a custom `View` overrides `onSaveInstanceState()` and `onRestoreInstanceState()` to persist data as needed, such as to handle a screen orientation change. The biggest difference is that rather than receive a `Bundle` as a parameter, `onSaveInstanceState()` must return a `Parcelable` with its state...including whatever state comes from the parent `View`.

The simplest way to do that is to return a `Bundle`, in which we have filled in our state (the chosen color) and the parent class' state (whatever that may be).

So, for example, here are implementations of `onSaveInstanceState()` and `onRestoreInstanceState()` from `ColorMixer`:

```
@Override
public Parcelable onSaveInstanceState() {
  Bundle state=new Bundle();

  state.putParcelable(SUPERSTATE, super.onSaveInstanceState());
  state.putInt(COLOR, getColor());

  return(state);
}
```

```
@Override
public void onRestoreInstanceState(Parcelable ss) {
  Bundle state=(Bundle)ss;

  super.onRestoreInstanceState(state.getParcelable(SUPERSTATE));

  setColor(state.getInt(COLOR));
}
```

The Rest of the Functionality

ColorMixer defines a callback interface, named OnColorChangedListener:

```
public interface OnColorChangedListener {
  public void onColorChange(int argb);
}
```

ColorMixer also provides getters and setters for an OnColorChangedListener object:

```
public OnColorChangedListener getOnColorChangedListener() {
  return(listener);
}

public void setOnColorChangedListener(OnColorChangedListener listener) {
  this.listener=listener;
}
```

The rest of the logic is mostly tied up in the SeekBar handler, which will adjust the swatch based on the new color and invoke the OnColorChangedListener object, if there is one:

```
private SeekBar.OnSeekBarChangeListener onMix=new
SeekBar.OnSeekBarChangeListener() {
  public void onProgressChanged(SeekBar seekBar, int progress,
                                boolean fromUser) {
    int color=getColor();

    swatch.setBackgroundColor(color);

    if (listener!=null) {
      listener.onColorChange(color);
    }
  }
```

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

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

Seeing It In Use

The project contains a sample activity, ColorMixerDemo, that shows the use of the ColorMixer widget.

The layout for that activity, shown below, can be found in res/layout/main.xml of the Views/ColorMixer project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:mixer="http://schemas.android.com/apk/res/com.commonsware.android.colorm
ixer"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
>
  <TextView android:id="@+id/color"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
  />
  <com.commonsware.android.colormixer.ColorMixer
    android:id="@+id/mixer"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    mixer:initialColor="#FFA4C639"
  />
</LinearLayout>
```

Notice that the root LinearLayout element defines two namespaces, the standard android namespace, and a separate one named mixer. The URL associated with that namespace indicates that we are looking to reference styleable attributes from the com.commonsware.android.colormixer package.

Our ColorMixer widget is in the layout, with a fully-qualified class name (com.commonsware.android.colormixer.ColorMixer), since ColorMixer is not in

the `android.widget` package. Notice that we can treat our custom widget like any other, giving it a width and height and so on.

The one attribute of our `ColorMixer` widget that is unusual is `mixer:initialColor`. `initialColor`, you may recall, was the name of the attribute we declared in `res/values/attrs.xml` and retrieve in Java code, to represent the color to start with. The `mixer` namespace is needed to identify where Android should be pulling the rules for what sort of values an `initialColor` attribute can hold. Since our `<attr>` element indicated that the format of `initialColor` was `color`, Android will expect to see a color value here, rather than a string or dimension.

The `ColorMixerDemo` activity is not very elaborate:

```
package com.commonsware.android.colormixer;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class ColorMixerDemo extends Activity {
  private TextView color=null;

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

    color=(TextView)findViewById(R.id.color);

    ColorMixer mixer=(ColorMixer)findViewById(R.id.mixer);

    mixer.setOnColorChangedListener(onColorChange);
  }

  private ColorMixer.OnColorChangedListener onColorChange=
    new ColorMixer.OnColorChangedListener() {
    public void onColorChange(int argb) {
      color.setText(Integer.toHexString(argb));
    }
  };
}
```

It gets access to both the `ColorMixer` and the `TextView` in the main layout, then registers an `OnColorChangedListener` with the `ColorMixer`. That listener,

in turn, puts the value of the color in the `TextView`, so the user can see the hex value of the color along with the shade itself in the swatch.

More Fun With ListViews

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

We start with a look at how to have a `ListView` with more than one distinct type of row, section headers in this case. We then move ahead to look at how to have header and footer rows that are in the `ListView` but are not part of your actual adapter. We then spend a pair of sections discussing the list selector – that orange bar that appears as you navigate a `ListView` with the D-pad or trackball – and how to control its behavior.

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:
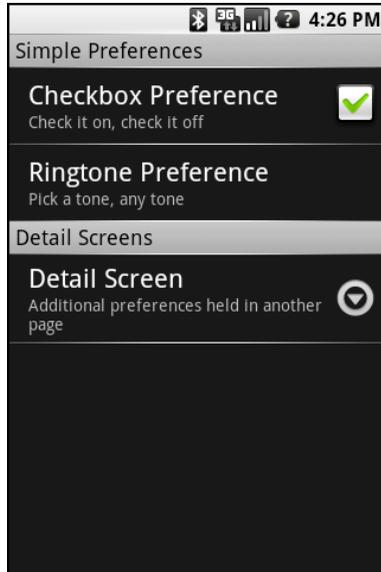
Figure 6. A PreferenceScreen UI

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`. You can get the source code to `MergeAdapter` from its GitHub repository.

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).

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 MergeAdapter 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 – MergeAdapterDemo – is just a ListActivity:

```
<?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" />
  <ListView android:id="@android:id/list" android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent" android:drawSelectorOnTop="false" />
</LinearLayout>
```

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` with heading views in between (one a `Button`, one a `TextView`):

```java
package com.commonsware.cwac.merge.demo;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import android.app.ListActivity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.Spinner;
import android.widget.TextView;
import com.commonsware.cwac.merge.MergeAdapter;

public class MergeAdapterDemo extends ListActivity {
  private static final 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 MergeAdapter adapter=null;
  private ArrayAdapter<String> arrayAdapter=null;

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

    adapter=new MergeAdapter();
    arrayAdapter=buildFirstList();
    adapter.addAdapter(arrayAdapter);
    adapter.addView(buildButton(), true);
    adapter.addAdapter(buildSecondList());
    adapter.addView(buildLabel());
    adapter.addAdapter(buildSecondList());

    setListAdapter(adapter);

    MergeAdapter spinnerAdapter=new MergeAdapter();
```

```java
    spinnerAdapter.addAdapter(buildFirstSpinnerList());
    spinnerAdapter.addAdapter(buildSecondSpinnerList());

    ((Spinner)findViewById(R.id.spinner)).setAdapter(spinnerAdapter);
  }

  @Override
  public void onListItemClick(ListView parent, View v, int position,
                              long id) {
    Log.d("MergeAdapterDemo", String.valueOf(position));
  }

  private ArrayAdapter<String> buildFirstList() {
    return(new ArrayAdapter<String>(
                                this,
                                android.R.layout.simple_list_item_1,
                                new ArrayList<String>(
                                                Arrays
                                                  .asList(items))));
  }

  private View buildButton() {
    Button result=new Button(this);

    result.setText("Add Capitalized Words");
    result.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        for (String item : items) {
          arrayAdapter.add(item.toUpperCase());
        }
      }
    });

    return(result);
  }

  private View buildLabel() {
    TextView result=new TextView(this);

    result.setText("Hello, world!");

    return(result);
  }

  private ListAdapter buildSecondList() {
    ArrayList<String> list=new ArrayList<String>(Arrays.asList(items));

    Collections.shuffle(list);

    return(new ArrayAdapter<String>(
                                this,
                                android.R.layout.simple_list_item_1,
                                list));
```

```
  }

  private ArrayAdapter<String> buildFirstSpinnerList() {
    ArrayAdapter<String> result=
        new ArrayAdapter<String>(
                                  this,
                                  android.R.layout.simple_spinner_item,
                                  new ArrayList<String>(
                                                        Arrays
                                                              .asList(items)));

    result
          .setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item
);

    return(result);
  }
  private ListAdapter buildSecondSpinnerList() {
    ArrayList<String> list=new ArrayList<String>(Arrays.asList(items));

    Collections.shuffle(list);

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

    result
          .setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item
);

    return(result);
  }
}
```

The result is much as you might expect:
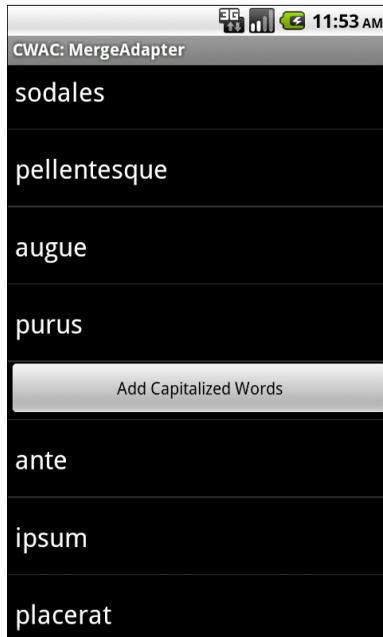
Figure 7. A ListView using a MergeAdapter, showing two lists and an intervening header

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.

How MergeAdapter Does It

MergeAdapter is a surprisingly lengthy class, given its simple premise: stitch together several ListAdapter objects to appear to the outside world as a single contiguous ListAdapter. This section will not cover all aspects of MergeAdapter, but here are some of the highlights.

Managing Adapters

MergeAdapter maintains an ArrayList of ListAdapter objects that represent the contents, and there is an addAdapter() method that adds a ListAdapter to that ArrayList.

`MergeAdapter` also supports adding an individual `View`, via a few flavors of `addView()` and `addViews()` methods. It does this via the simple expedient of creating a `ListAdapter` out of those widget(s) and adding that `ListAdapter` to the `ArrayList`. Specifically, `MergeAdapter` uses another CWAC component, `SackOfViewsAdapter`, to handle that chore. `SackOfViewsAdapter` is a `ListAdapter` that holds onto an `ArrayList` of `View` objects, one per row. This allows `MergeAdapter` to deal with everything in terms of `ListAdapter` objects.

You can take a look at `SackOfViewsAdapter` through its GitHub repository.

Supporting the Adapter Interface

`MergeAdapter` supports the `ListAdapter` interface, by extending `BaseAdapter`. That being said, there are still about a dozen methods that `MergeAdapter` needs to implement to supply its custom `ListAdapter` logic.

Most of those are a simple matter of iteration. For example, here is the implementation of `getViewTypeCount()`:

```
@Override
public int getViewTypeCount() {
  int total=0;

  for (ListAdapter piece : pieces) {
    total+=piece.getViewTypeCount();
  }

  return(Math.max(total, 1));  // needed for setListAdapter() before content
add'
}
```

The number of `View` types from the `MergeAdapter` is the sum of the number of `View` types from each of its member `ListAdapter` objects, or 1, whichever is higher.

Some of these require a bit more smarts in the iteration, such as `getItemViewType()`:

```
@Override
  public int getItemViewType(int position) {
```

```
    int typeOffset=0;
    int result=-1;

    for (ListAdapter piece : pieces) {
      int size=piece.getCount();

      if (position<size) {
        result=typeOffset+piece.getItemViewType(position);
        break;
      }

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

    return(result);
}
```

Here, we need to walk through the ListAdapter objects in sequence until we come to the one that has the desired position. Along the way, we add up how many View type "slots" were used by previous adapters. The View type slot for our desired position is the View type of that row within its ListAdapter, offset by the number of slots already consumed by previous adapters. This keeps us in the range of 0 to getViewTypeCount().

To deal with things being disabled, such as some headers, MergeAdapter returns false from areAllItemsEnabled(). It then delegates isItemEnabled() to its adapter:

```
@Override
 public boolean isEnabled(int position) {
   for (ListAdapter piece : pieces) {
     int size=piece.getCount();

     if (position<size) {
       return(piece.isEnabled(position));
     }

     position-=size;
   }

   return(false);
}
```

The rest of the methods on ListAdapter are implemented in similar fashion.

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.header;

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

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 boolean areWeDeadYet=false;

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

```java
    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=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) {
      getListView().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:
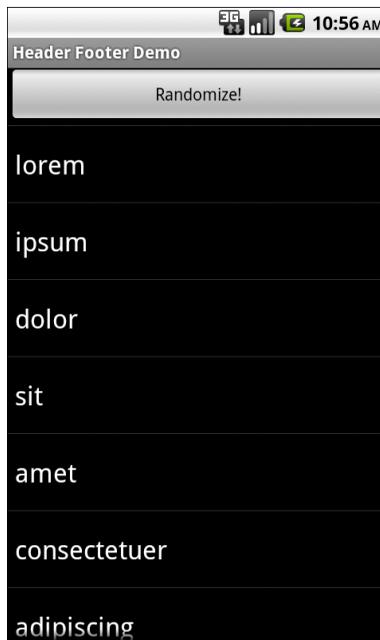
Figure 8. A ListView with a header view shown

If you scroll downward, the header will slide off the top, and eventually the footer will scroll into view:
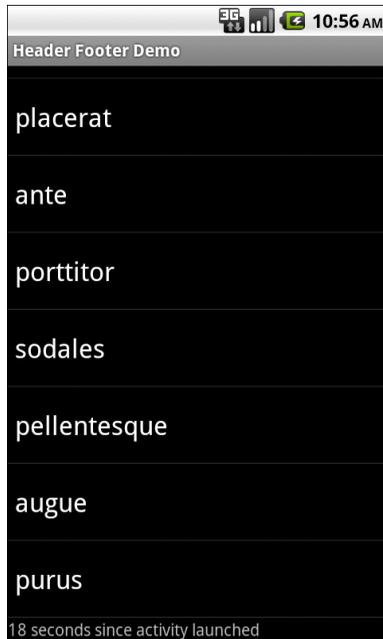
Figure 9. A ListView with a footer view shown

Note that the same effect can be achieved with a `MergeAdapter`. `MergeAdapter` offers somewhat greater flexibility, at the cost of requiring an external library.

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
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <View
    android:id="@+id/bar"
    android:background="#FFFFFF00"
    android:layout_width="5px"
    android:layout_height="match_parent"
    android:visibility="invisible"
  />
  <TextView
    android:id="@+id/label"
    android:layout_width="match_parent"
    android:layout_height="match_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.

44

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
<?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="match_parent"
  android:layout_height="match_parent"
  android:listSelector="#00000000"
/>
```

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

Figure 10. 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
<?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

47

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:
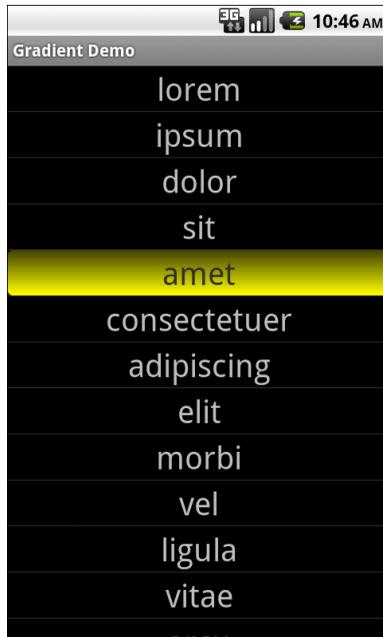
Figure 11. The GradientDemo sample application

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
<?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 .

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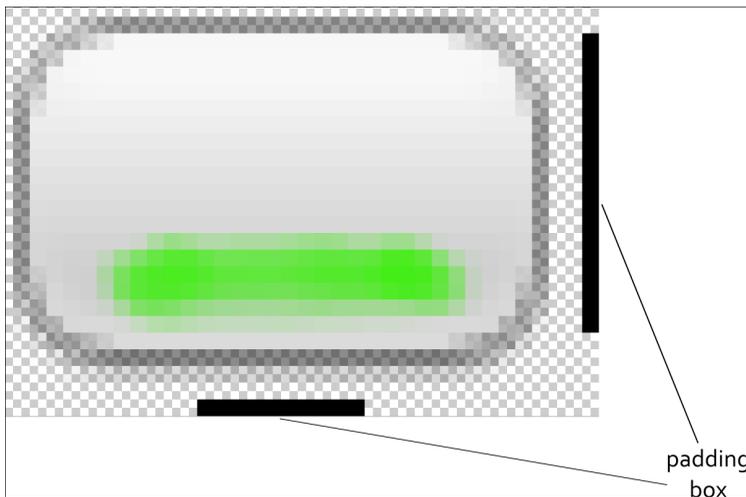
Figure 12. The padding box, as shown by a set of control lines to the right and bottom of the stretchable image

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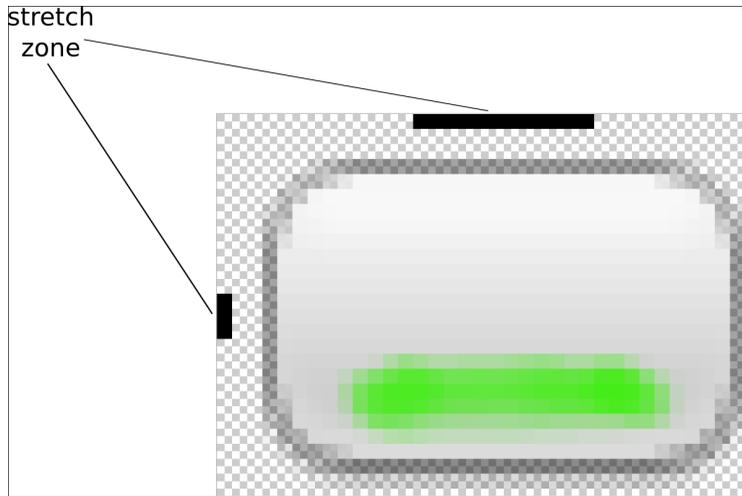
Figure 13. The stretch zones, as shown by a set of control lines to the left and top of the stretchable image

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:
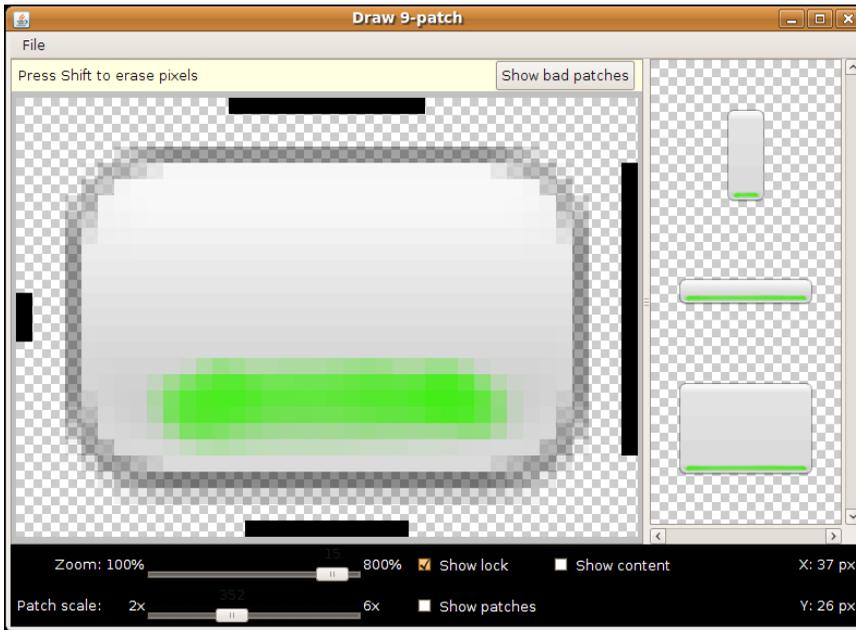
Figure 14. The draw9patch tool

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

59

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
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
  >
    <TableRow
      android:layout_width="match_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="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow
      android:layout_width="match_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="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
  </TableLayout>
```

```
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <Button android:id="@+id/resize"
      android:layout_width="64px"
      android:layout_height="64px"
      android:text="Hi!"
      android:textSize="5pt"
      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 64px square:

```
package com.commonsware.android.ninepatch;

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(176); // 240 less 64 starting size
    vertical.setMax(176);   // 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(64+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,
                                                                   64+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:
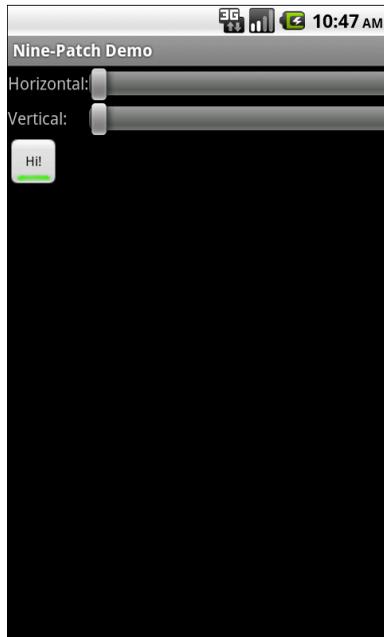
Figure 15. The NinePatch sample project, in its initial state
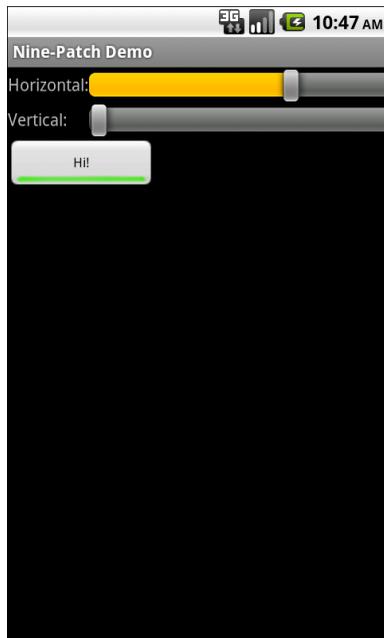
Figure 16. The NinePatch sample project, after making it bigger horizontally
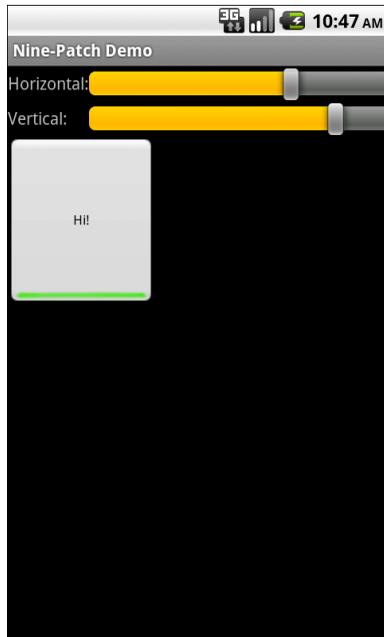
Figure 17. The NinePatch sample application, after making it bigger in both dimensions

Home Screen App Widgets

One of the oft-requested features added in Android 1.5 was 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/PairOfDice` project, you will find an app widget that displays a roll of a pair of dice. Clicking on the app widget re-rolls, in case you want a better result.

The Manifest

First, we need to register our `BroadcastReceiver` implementation in our `AndroidManifest.xml` file, along with a few extra features:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.appwidget.dice"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-sdk android:minSdkVersion="5"
            android:targetSdkVersion="8" />
  <supports-screens android:largeScreens="true"
                     android:normalScreens="true"
                     android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <receiver android:icon="@drawable/cw"
              android:label="@string/app_name"
              android:name=".AppWidget">
      <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>
    <activity android:name="PairOfDiceActivity"
              android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Here we have just a `<receiver>`. 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.

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="144dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  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. Also, ideally, you use `dip` instead of `px` for the

Special Creative Commons BY-NC-SA 3.0 License Edition

dimensions, so the number of cells will remain constant regardless of screen density.

- 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 "rounded up" to 30 minutes. Hence our 15-minute (`900000` millisecond) request will actually result in an update every 30 minutes.

- The initial layout to use for the app widget, for the time between when the user requests the app widget and when `onUpdate()` of our `AppWidgetProvider` gets control.

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 `PairOfDice` app widget:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/background"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/widget_frame"
    >
  <ImageView android:id="@+id/left_die"
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/die_5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
  />
  <ImageView android:id="@+id/right_die"
    android:layout_centerVertical="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/die_2"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="7dip"
  />
</RelativeLayout>
```

All we have is a pair of `ImageView` widgets (one for each die), inside of a `RelativeLayout`. The `RelativeLayout` has a background, specified as a nine-patch PNG file. This allows the `RelativeLayout` to have guaranteed contrast with whatever wallpaper is behind it, so the user can tell the actual app widget bounds.

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 `PairOfDice`:

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

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.widget.RemoteViews;

public class AppWidget extends AppWidgetProvider {
  private static final int[] IMAGES={R.drawable.die_1,R.drawable.die_2,
                                     R.drawable.die_3,R.drawable.die_4,
                                     R.drawable.die_5,R.drawable.die_6};
```

```
  @Override
  public void onUpdate(Context ctxt, AppWidgetManager mgr,
                       int[] appWidgetIds) {
    ComponentName me=new ComponentName(ctxt, AppWidget.class);

    mgr.updateAppWidget(me, buildUpdate(ctxt, appWidgetIds));
  }

  private RemoteViews buildUpdate(Context ctxt, int[] appWidgetIds) {
    RemoteViews updateViews=new RemoteViews(ctxt.getPackageName(),
                                            R.layout.widget);

    Intent i=new Intent(ctxt, AppWidget.class);

    i.setAction(AppWidgetManager.ACTION_APPWIDGET_UPDATE);
    i.putExtra(AppWidgetManager.EXTRA_APPWIDGET_IDS, appWidgetIds);

    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0 , i,
                                        PendingIntent.FLAG_UPDATE_CURRENT)
;

    updateViews.setImageViewResource(R.id.left_die,
                                IMAGES[(int)(Math.random()*6)]);
    updateViews.setOnClickPendingIntent(R.id.left_die, pi);
    updateViews.setImageViewResource(R.id.right_die,
                                IMAGES[(int)(Math.random()*6)]);
    updateViews.setOnClickPendingIntent(R.id.right_die, pi);
    updateViews.setOnClickPendingIntent(R.id.background, pi);

    return(updateViews);
  }
}
```

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`.

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:

- `setImageViewResource()` to set the image for each of our `ImageView` widgets, in this case a randomly chosen die face (using graphics created from a set of SVG files from the OpenClipArt site)
- `setOnClickPendingIntent()` to provide a `PendingIntent` that should get fired off when a die, or the overall app widget background, is clicked

We then supply that `RemoteViews` to the `AppWidgetManager`, which pushes the `RemoteViews` structure to the home screen, which renders our new app widget UI.

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:
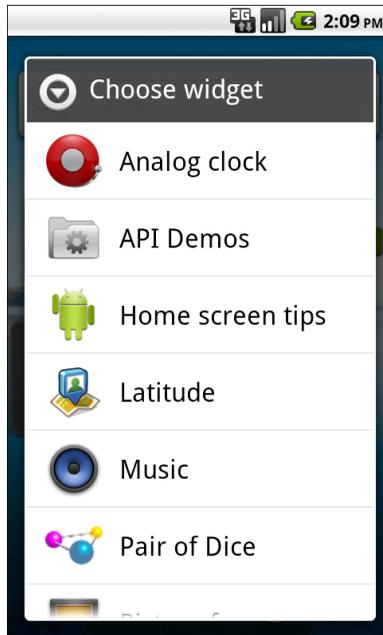
Figure 18. The roster of available widgets

When you choose Pair of Dice, the app widget will appear on the home screen:
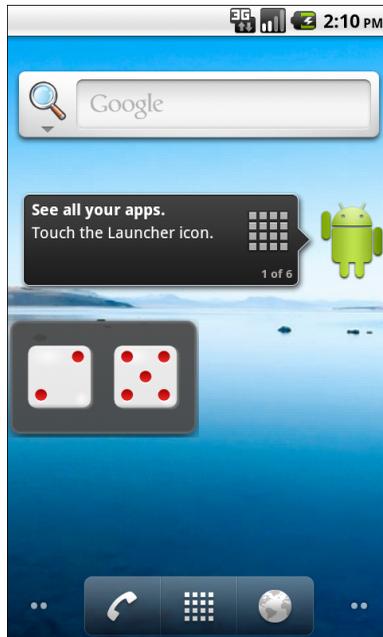
Figure 19. The Pair of Dice app widget, in action

To re-roll, just tap anywhere on the app widget.

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. 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 `PairOfDice` app widget to your home screen, nothing blows up – they just show the same roll. That is because `PairOfDice` uses a version of `updateAppWidget()` that does not take any app widget IDs, and therefore updates all app widgets simultaneously.

App Widgets: Their Life and Times

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 has since been fixed, and if you are not supporting Android 1.5, you will not need to worry about this problem.

Controlling Your (App Widget's) Destiny

As `PairOfDice` 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 `PairOfDice` will let you use `AlarmManager` (described in another 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`)

Also, note that the `updatePeriodMillis` 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 `updatePeriodMillis` to `0` and use `AlarmManager` to control the timing and behavior of your widget updates.

Note that if there are multiple instances of your app widget on the user's home screen, they will all update approximately simultaneously if you are using `updatePeriodMillis`. If you elect to set up your own update schedule, you can control which app widgets get updated when, if you choose.

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, like Sony Ericsson, Samsung, and LG, have followed suit, as will others in the future. 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
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_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:

```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/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. 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.

One Size May Not Fit All

It may be that you want to offer multiple app widget sizes to your users. Some might only want a small app widget. Some might really like what you have to offer and want to give you more home screen space to work in.

The good news: this is easy to do.

The bad news: it requires you, in effect, to have one app widget per size.

The size of an app widget is determined by the app widget metadata XML file. That XML file is tied to a `<receiver>` element in the manifest representing one app widget. Hence, to have multiple sizes, you need multiple metadata files and multiple `<receiver>` elements.

This also means your app widgets will show up multiple times in the app widget selection list, when the user goes to add an app widget to their home screen. Hence, supporting many sizes will become annoying to the user, if they perceive you are "spamming" the app widget list. Try to keep the number of app widget sizes to a reasonable number (say, one or two sizes).

For certain types of app widgets, you can now make them resizeable, as of API Level 11 – this will be discussed in the next section.

Advanced App Widgets on Android 3.x

Android 3.0 introduced a few new capabilities for app widgets, to make them more interactive and more powerful than before. The documentation lags a bit, though, so determining how to use these features takes a bit of exploring. Fortunately for you, the author did some of that exploring on your behalf, to save you some trouble.

New Widgets for App Widgets

In addition to the classic widgets available for use in app widgets and RemoteViews, five more were added for Android 3.0:

- GridView
- ListView
- StackView
- ViewFlipper
- AdapterViewFlipper

Three of these (GridView, ListView, ViewFlipper) are widgets that existed in Android since the outset. StackView is a new widget to provide a "stack of cards" UI:

Figure 20. The Google Books app widget, showing a StackView

`AdapterViewFlipper` works like a `ViewFlipper`, allowing you to toggle between various children with only one visible at a time. However, whereas with `ViewFlipper` all children are fully-instantiated `View` objects held by the `ViewFlipper` parent, `AdapterViewFlipper` uses the `Adapter` model, so only a small number of actual `View` objects are held in memory, no matter how many potential children there are.

With the exception of `ViewFlipper`, the other four all require the use of an `Adapter`. This might seem odd, as there is no way to provide an `Adapter` to a `RemoteViews`. That is true, but Android 3.0 added new ways for `Adapter`-like communication between the app widget host (e.g., home screen) and your application. We will take an in-depth look at that in an upcoming section.

Preview Images

App widgets can now have preview images attached. Preview images are drawable resources representing a preview of what the app widget might look like on the screen. On tablets, this will be used as part of an app widget gallery, replacing the simple context menu presentation you see on Android 1.x and 2.x phones:
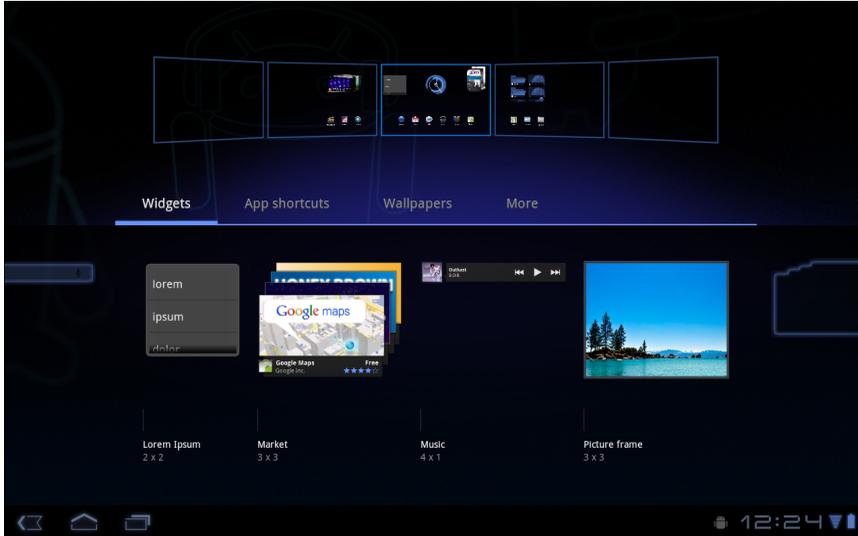
Figure 21. The XOOM tablet's app widget gallery

To create the preview image itself, the Android 3.0 emulator contains a Widget Preview application that lets you run an app widget in its own container, outside of the home screen:
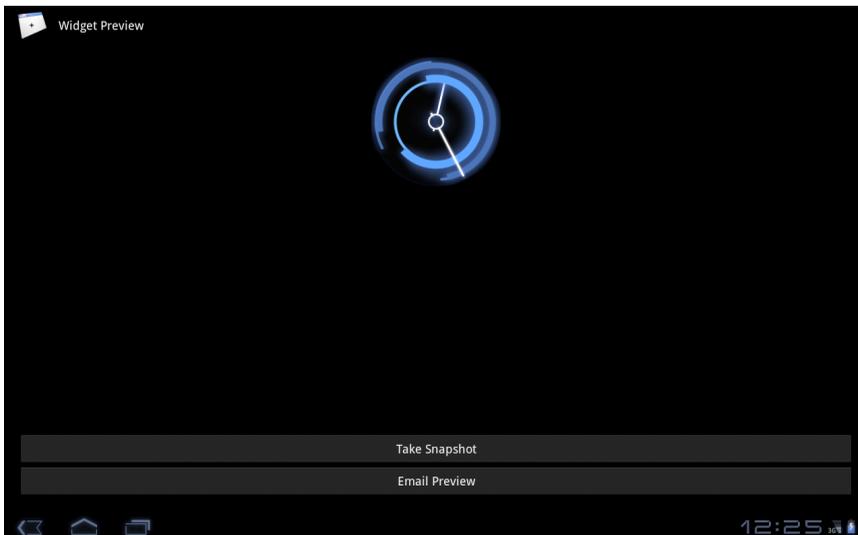
Figure 22. The Widget Preview application, showing a preview of the Analog Clock app widget

From here, you can take a snapshot and save it to external storage, copy it to your project's `res/drawable-nodpi/` directory (indicating that there is no intrinsic density assumed for this image), and reference it in your app widget metadata via an `android:previewImage` attribute. We will see an example of such an attribute in .

Adapter-Based App Widgets

In an activity, if you put a `ListView` or `GridView` into your layout, you will also need to hand it an `Adapter`, providing the actual row or cell `View` objects that make up the contents of those selection widgets.

In an app widget, this becomes a bit more complicated. The host of the app widget does not have any `Adapter` class of yours. Hence, just as we have to send the contents of the app widget's UI via a `RemoteViews`, we will need to provide the rows or cells via `RemoteViews` as well. Android, starting with API Level 11, has a `RemoteViewsService` and `RemoteViewsFactory` that you can use for this purpose. Let's take a look, in the form of the `AppWidget/LoremWidget` sample project, which will put a `ListView` of 25 nonsense words into an app widget.

The AppWidgetProvider

At its core, our `AppWidgetProvider` (named `WidgetProvider`, in a stunning display of creativity) still needs to create and configure a `RemoteViews` object with the app widget UI, then use `updateAppWidget()` to push that `RemoteViews` to the host via the `AppWidgetManager`. However, for an app widget that involves an `AdapterView`, like `ListView`, there are two more key steps:

1. You have to tell the `RemoteViews` the identity of a `RemoteViewsService` that will help fill the role that the `Adapter` would in an activity

2. You have to provide the `RemoteViews` with a "template" `PendingIntent` to be used when the user taps on a row or cell in the `AdapterView`, to replace the `onListItemClick()` or similar method you might have used in an activity

For example, here is WidgetProvider for our nonsense-word app widget:

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

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.Context;
import android.content.Intent;
import android.content.ComponentName;
import android.net.Uri;
import android.widget.RemoteViews;

public class WidgetProvider extends AppWidgetProvider {
  public static String EXTRA_WORD=
    "com.commonsware.android.appwidget.lorem.WORD";

  @Override
  public void onUpdate(Context ctxt, AppWidgetManager appWidgetManager,
                       int[] appWidgetIds) {
    for (int i=0; i<appWidgetIds.length; i++) {
      Intent svcIntent=new Intent(ctxt, WidgetService.class);

      svcIntent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);
      svcIntent.setData(Uri.parse(svcIntent.toUri(Intent.URI_INTENT_SCHEME)));

      RemoteViews widget=new RemoteViews(ctxt.getPackageName(),
                                         R.layout.widget);

      widget.setRemoteAdapter(appWidgetIds[i], R.id.words,
                              svcIntent);

      Intent clickIntent=new Intent(ctxt, LoremActivity.class);
      PendingIntent clickPI=PendingIntent
                            .getActivity(ctxt, 0,
                                         clickIntent,
                                         PendingIntent.FLAG_UPDATE_CURRENT);

      widget.setPendingIntentTemplate(R.id.words, clickPI);

      appWidgetManager.updateAppWidget(appWidgetIds[i], widget);
    }

    super.onUpdate(ctxt, appWidgetManager, appWidgetIds);
  }
}
```

The call to setRemoteAdapter() is where we point the RemoteViews to our RemoteViewsService for our AdapterView widget. The main rules for the Intent used to identify the RemoteViewsService are:

- The service must be identified by its data (`Uri`), so even if you create the `Intent` via the `Context-and-Class` constructor, you will need to convert that into a `Uri` via `toUri(Intent.URI_INTENT_SCHEME)` and set that as the `Uri` for the `Intent`. Why? While your application has access to your `RemoteViewService Class` object, the app widget host will not, and so we need something that will work across process boundaries. You could elect to add your own `<intent-filter>` to the `RemoteViewsService` and use an `Intent` based on that, but that would make your service more publicly visible than you might want.

- Any extras that you package on the `Intent` – such as the app widget ID in this case – will be on the Intent that is delivered to the `RemoteViewsService` when it is invoked by the app widget host.

Note that this project uses the original form of `setRemoteAdapter()`, taking the app widget ID as the first parameter. That method signature was deprecated as of API Level 14 (Android 4.0 / Ice Cream Sandwich), as supplying the app widget ID was superfluous. Once Honeycomb tablets are mostly upgraded to Ice Cream Sandwich, you may wish to consider switching to the new two-parameter flavor of `setRemoteAdapter()`.

The call to `setPendingIntentTemplate()` is where we provide a `PendingIntent` that will be used as the template for all row or cell clicks. As we will see in a bit, the underlying `Intent` in the `PendingIntent` will have more data added to it by our `RemoteViewsFactory`.

In all other respects, our `WidgetProvider` is unremarkable compared to other app widgets. It will need to be registered in the manifest as a `<provider>`, as with any other app widget.

The RemoteViewsService

Android supplies a `RemoteViewsService` class that you will need to extend, and this class is the one you must register with the `RemoteViews` for an `AdapterView` widget. For example, here is `WidgetService` (once again, a highly creative name) from the `LoremWidget` project:

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

import android.content.Intent;
import android.widget.RemoteViewsService;

public class WidgetService extends RemoteViewsService {
  @Override
  public RemoteViewsFactory onGetViewFactory(Intent intent) {
    return(new LoremViewsFactory(this.getApplicationContext(),
                                 intent));
  }
}
```

As you can see, this service is practically trivial. You have to override one method, onGetViewFactory(), which will return the RemoteViewsFactory to use for supplying rows or cells for the AdapterView. You are passed in an Intent, the one used in the setRemoteAdapter() call. Hence, if you have more than one AdapterView widget in your app widget, you could elect to have two RemoteViewsService implementations, or one that discriminates between the two widgets via something in the Intent (e.g., custom action string). In our case, we only have one AdapterView, so we create an instance of a LoremViewFactory and return it. Google demonstrates using getApplicationContext() here to supply the Context object to RemoteViewsFactory, instead of using the Service as a Context – it is unclear at this time why this is.

Another thing different about the RemoteViewsService is how it is registered in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.appwidget.lorem"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="LoremActivity"
              android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <receiver android:icon="@drawable/cw"
```

```
                android:label="@string/app_name"
                android:name="WidgetProvider">
        <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="WidgetService"
               android:permission="android.permission.BIND_REMOTEVIEWS" />
    </application>
</manifest>
```

Note the use of android:permission, specifying that whoever sends an Intent to WidgetService must hold the BIND_REMOTEVIEWS permission. This can only be held by the operating system. This is a security measure, so arbitrary applications cannot find out about your service and attempt to spoof being the OS and cause you to supply them with RemoteViews for the rows, as this might leak private data.

The RemoteViewsFactory

A RemoteViewsFactory interface implementation looks and feels a lot like an Adapter. In fact, one could imagine that the Android developer community might create CursorRemoteViewsFactory and ArrayRemoteViewsFactory and such to further simplify writing these classes.

For example, here is LoremViewsFactory, the one used by the LoremWidget project:

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

import android.appwidget.AppWidgetManager;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.widget.RemoteViews;
import android.widget.RemoteViewsService;

public class LoremViewsFactory implements RemoteViewsService.RemoteViewsFactory
{
  private static final 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 Context ctxt=null;
  private int appWidgetId;

  public LoremViewsFactory(Context ctxt, Intent intent) {
      this.ctxt=ctxt;
      appWidgetId=intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                              AppWidgetManager.INVALID_APPWIDGET_ID);
  }

  @Override
  public void onCreate() {
    // no-op
  }

  @Override
  public void onDestroy() {
    // no-op
  }

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

  @Override
  public RemoteViews getViewAt(int position) {
    RemoteViews row=new RemoteViews(ctxt.getPackageName(),
                          R.layout.row);

    row.setTextViewText(android.R.id.text1, items[position]);

    Intent i=new Intent();
    Bundle extras=new Bundle();

    extras.putString(WidgetProvider.EXTRA_WORD, items[position]);
    i.putExtras(extras);
    row.setOnClickFillInIntent(android.R.id.text1, i);

    return(row);
  }

  @Override
  public RemoteViews getLoadingView() {
    return(null);
  }

  @Override
  public int getViewTypeCount() {
    return(1);
```

```
  }

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

  @Override
  public boolean hasStableIds() {
    return(true);
  }

  @Override
  public void onDataSetChanged() {
    // no-op
  }
}
```

You need to implement a handful of methods that have the same roles in a RemoteViewsFactory as they do in an Adapter, including:

- getCount()
- getViewTypeCount()
- getItemId()
- hasStableIds()

In addition, you have onCreate() and onDestroy() methods that you must implement, even if they do nothing, to satisfy the interface.

You will need to implement getLoadingView(), which will return a RemoteViews to use as a placeholder while the app widget host is getting the real contents for the app widget. If you return null, Android will use a default placeholder.

The bulk of your work will go in getViewAt(). This serves the same role as getView() does for an Adapter, in that it returns the row or cell View for a given position in your data set. However:

- You have to return a RemoteViews, instead of a View, just as you have to use RemoteViews for the main content of the app widget in your AppWidgetProvider

- There is no recycling, so you do not get a `View` (or `RemoteViews`) back to somehow repopulate, meaning you will create a new `RemoteViews` every time

The impact of the latter is that you do not want to put large data sets into an app widget, as scrolling may get sluggish, just as you do not want to implement an `Adapter` without recycling unused `View` objects.

In `LoremViewsFactory`, the `getViewAt()` implementation creates a `RemoteViews` for a custom row layout, cribbed from one in the Android SDK:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_vertical"
    android:paddingLeft="6dip"
    android:minHeight="?android:attr/listPreferredItemHeight"
/>
```

Then, `getViewAt()` pours in a word from the static `String[]` of nonsense words into that `RemoteViews` for the `TextView` inside it. It also creates an `Intent` and puts the nonsense word in as an `EXTRA_WORD` extra, then provides that `Intent` to `setOnClickFillInIntent()`. The contents of the "fill-in" `Intent` are merged into the "template" `PendingIntent` from `setPendingIntentTemplate()`, and the resulting `PendingIntent` is what is invoked when the user taps on an item in the `AdapterView`. The fully-configured `RemoteViews` is then returned.

The Rest of the Story

The app widget metadata needs no changes related to `Adapter`-based app widget contents. However, `LoremWidget` does add the `android:previewImage` attribute:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="146dip"
  android:minHeight="146dip"
  android:updatePeriodMillis="0"
  android:initialLayout="@layout/widget"
```

```
    android:autoAdvanceViewId="@+id/words"
    android:previewImage="@drawable/preview"
    android:resizeMode="vertical"
/>
```

This points to the `res/drawable-nodpi/preview.png` file that represents a "widgetshot" of the app widget in isolation, obtained from the Widget Preview application:

Figure 23. The preview of LoremWidget

Also, the metadata specifies `android:resizeMode="vertical"`. This attribute is new to Android 3.1, and allows the app widget to be resized by the user (in this case, only in the vertical direction, to show more rows). Older versions of Android will ignore this attribute, and the app widget will remain in your requested size. You can use `vertical`, `horizontal`, or `both` (via the pipe operator) as values for `android:resizeMode`.

When the user taps on an item in the list, our `PendingIntent` is set to bring up `LoremActivity`. This activity has `android:theme="@android:style/Theme.NoDisplay"` set in the manifest, meaning that it will not have its own user interface. Rather, it will extra our `EXTRA_WORD` out of the `Intent` used to launch the activity and display it in a `Toast` before finishing:

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

import android.app.Activity;
import android.os.Bundle;
```

91

```
import android.widget.Toast;

public class LoremActivity extends Activity {
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);

    String word=getIntent().getStringExtra(WidgetProvider.EXTRA_WORD);

    if (word==null) {
      word="We did not get a word!";
    }

    Toast.makeText(this, word, Toast.LENGTH_LONG).show();

    finish();
  }
}
```

The Results

When you compile and install the application, nothing new shows up in the home screen launcher, because we have no activity defined to respond to ACTION_MAIN and CATEGORY_HOME. This would be unusual for an application distributed through the Android Market, as users often get confused if they install something and then do not know how to start it. However, for the purposes of this example, we should be fine, as readers of programming books never get confused about such things.

However, if you bring up the app widget gallery (e.g., long-tap on the home screen of a Motorola XOOM), you will see LoremWidget there, complete with preview image. You can drag it into one of the home screen panes and position it. When done, the app widget appears as expected:
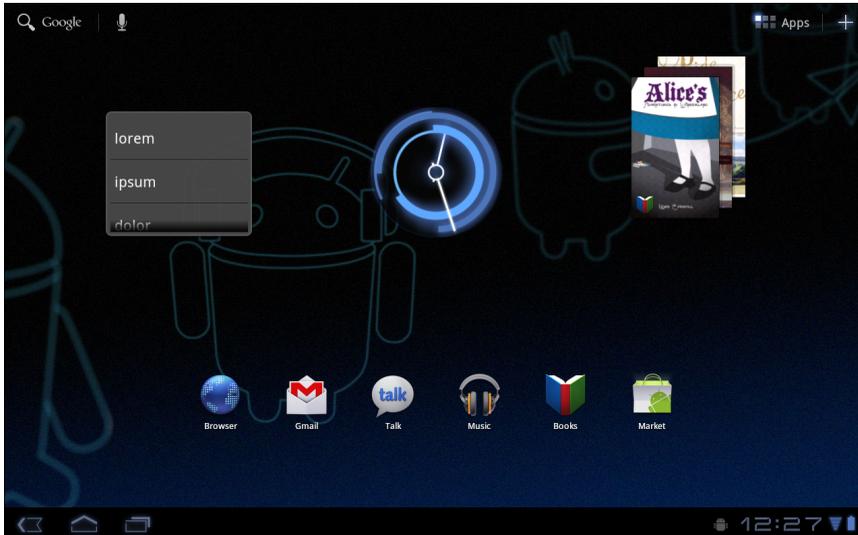
Figure 24. A XOOM home screen, showing the LoremWidget on the left

The `ListView` is live and can be scrolled. Tapping an entry brings up the corresponding `Toast`:
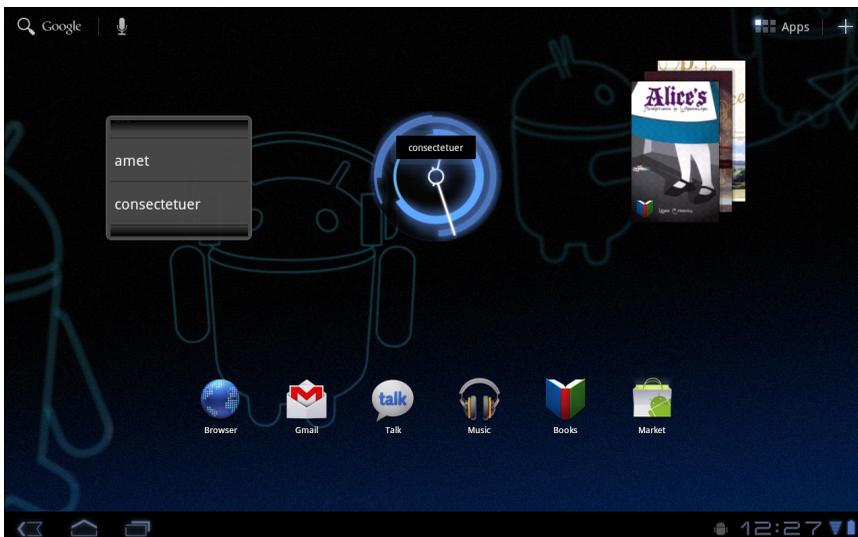
Figure 25. A XOOM home screen, showing the LoremWidget on the left

---

93

The above image illustrates that a Toast is not a great UI choice on a tablet, given the relative size of the Toast compared to the screen. Users will be far more likely to miss the Toast than ever before.

If the user long-taps on the app widget, they will be able to reposition it. On Android 3.1 and beyond, when they lift their finger after the long-tap, the app widget will show resize handles on the sides designated by your android:resizeMode attribute:
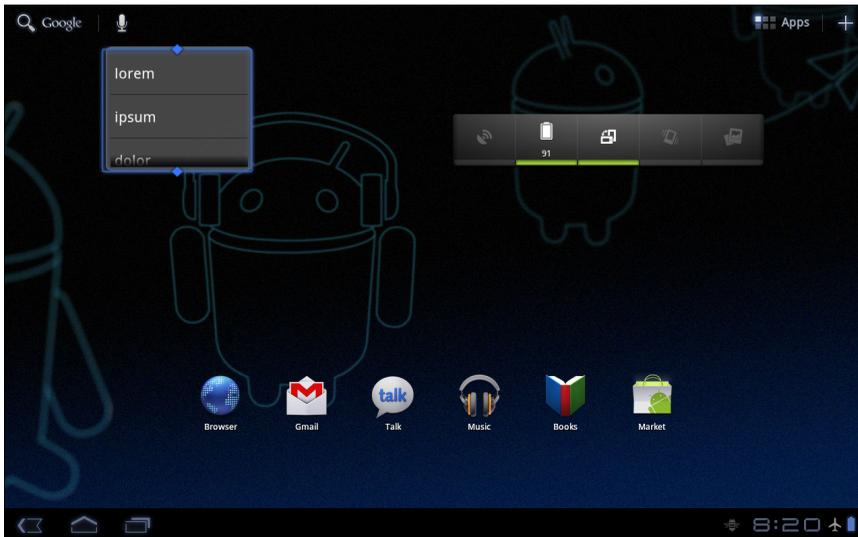
Figure 26. A XOOM home screen, showing the LoremWidget on the left, with resize handles

The user can then drag those handles to expand or shrink the app widget in the specified dimensions:
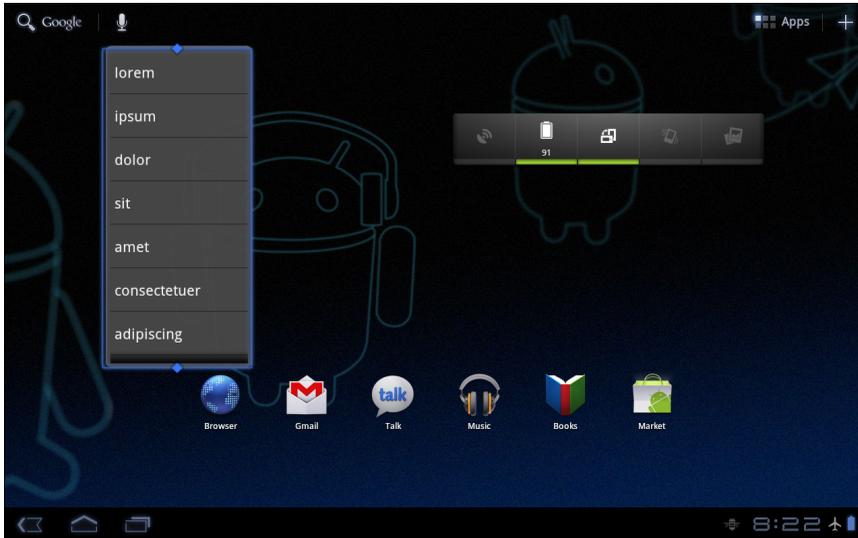
Figure 27. The resized LoremWidget

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.

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.

We start by demonstrating how you can convert from latitude and longitude to screen coordinates on the current map. We then investigate what it takes to layer things on top of the map, such as a persistent pop-up

panel instead of using a transient Toast to display something in response to a tap. Next, we look at how to have custom icons per item in an ItemizedOverlay, rather than having everything the overlay look the same. We wrap up with coverage of how to load up the contents of an ItemizedOverlay asynchronously, in case that might take a while and should not be done on the main application thread.

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 28. 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
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_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;
}
```

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:

```java
@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:

```java
package com.commonsware.android.nooer;

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

105

```java
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
```

```java
  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 29. The EvenNooerYawk application, showing the PopupPanel towards the bottom

...and like this when it is towards the top:
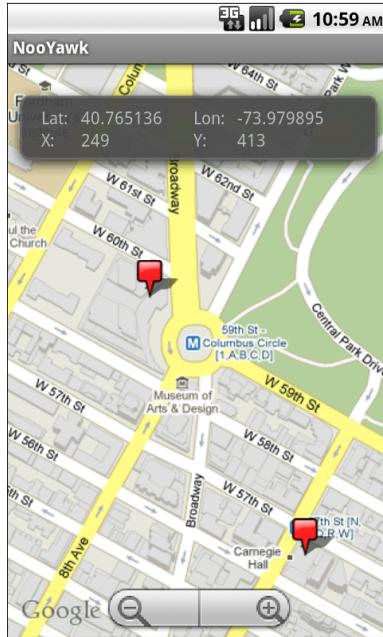
Figure 30. The EvenNooerYawk application, showing the PopupPanel towards the top

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.

111

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:

```xml
<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:

```java
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 31. The ILuvNooYawk application, showing custom icons per item

...we can change those by clicking on an item and pressing the H key:
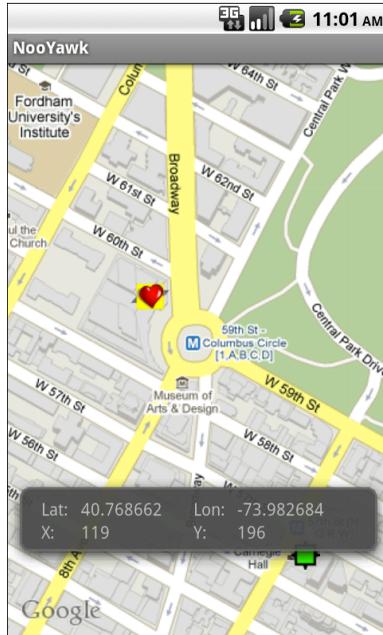
Figure 32. The ILuvNooYawk application, showing one item's icon toggled to a heart (and selected)

Note that getMarker() on an OverlayItem gets called very frequently – every time the map is panned or zoomed, the markers are re-requested. As such, it is important that getMarker() be as efficient as possible, particularly if you have a lot of items in your overlay.

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

117

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));
}
```

A Little Touch of Noo Yawk

As all of these examples have demonstrated, users can tap on maps, particularly on `OverlayItem` icons, to indicate something of interest.

Sometimes, though, what they really want to do is move one of those items.

For example:

- They might want to reposition an endpoint for a route for which you are providing turn-by-turn directions
- They might want to fine-tune a waypoint on a set of walking or cycling tour stops they are designing using your app, adjusting its location by a bit
- They might want to change the corner points on a polygon they are creating on your map, to designate postal zones or township boundaries or whatever

Courtesy of an assist from Greg Milette, we can show you how this is done, via the `Maps/NooYawkTouch` project.

Touch Events

Simple touch events are...well...fairly simple.

In an `ItemizedOverlay`, you can override the `onTouchEvent()` method, to be notified of touch operations. Any event you pass to the superclass will be handled as normal, such as item taps or pan-and-zoom operations. However, you can intercept events that you would prefer to handle yourself. Your `onTouchEvent()` method will be passed a `MotionEvent` object (the actual event) and the `MapView`.

There are three touch events of relevance for repositioning items on a map, distinguished by their action (getAction() on the MotionEvent):

1. MotionEvent.ACTION_DOWN, when a finger is placed onto the touchscreen

2. MotionEvent.ACTION_MOVE, when the finger is slid across the touchscreen

3. MotionEvent.ACTION_UP, when the finger is lifted off of the touchscreen

The MotionEvent also gives you the screen coordinates of where the touch event occurred, via getX() and getY().

To manage a drag operation, therefore, we need to:

- Watch for an ACTION_DOWN event, identify the item that was touched, and kick off the drag

- Watch for ACTION_MOVE events while we are in "drag mode" and move the item to the new position

- Watch for an ACTION_UP event and stop the drag operation, positioning the item in its final resting place

Here is the implementation of onTouchEvent() for the NooYawkTouch version of SitesOverlay:

```java
@Override
public boolean onTouchEvent(MotionEvent event, MapView mapView) {
  final int action=event.getAction();
  final int x=(int)event.getX();
  final int y=(int)event.getY();
  boolean result=false;

  if (action==MotionEvent.ACTION_DOWN) {
    for (OverlayItem item : items) {
      Point p=new Point(0,0);

      map.getProjection().toPixels(item.getPoint(), p);

      if (hitTest(item, marker, x-p.x, y-p.y)) {
        result=true;
        inDrag=item;
```

```
        items.remove(inDrag);
        populate();

        xDragTouchOffset=0;
        yDragTouchOffset=0;

        setDragImagePosition(p.x, p.y);
        dragImage.setVisibility(View.VISIBLE);

        xDragTouchOffset=x-p.x;
        yDragTouchOffset=y-p.y;

        break;
      }
    }
  }
  else if (action==MotionEvent.ACTION_MOVE && inDrag!=null) {
    setDragImagePosition(x, y);
    result=true;
  }
  else if (action==MotionEvent.ACTION_UP && inDrag!=null) {
    dragImage.setVisibility(View.GONE);

    GeoPoint pt=map.getProjection().fromPixels(x-xDragTouchOffset,
                                               y-yDragTouchOffset);
    OverlayItem toDrop=new OverlayItem(pt, inDrag.getTitle(),
                                   inDrag.getSnippet());

    items.add(toDrop);
    populate();

    inDrag=null;
    result=true;
  }

  return(result || super.onTouchEvent(event, mapView));
}
```

We will look at the three major branches of this code in the sections that follow.

Finding an Item

`ItemizedOverlay` offers a convenient `hitTest()` method, to determine if a touch event (or anything else with a screen coordinate) is "close" to a specific `OverlayItem`. The `hitTest()` method returns a simple `boolean` indicating if the touch event was a hit on the item. Hence, to find out if a given `ACTION_DOWN` event was on an item, we can simply iterate over all

items, passing each to `hitTest()`, and breaking out of the loop if we get a hit. If we make it through the whole loop with `hitTest()` returning `false` each time, the user tapped someplace away from any items.

The only catch is that `hitTest()` works in the item's frame of reference. Rather than passing a screen coordinate relative to the corner of the screen (as is returned by `getX()` and `getY()` on `MotionEvent`), we have to pass a coordinate relative to the item's on-screen location. Fortunately, Android provides some utility methods to assist with this as well.

So, let's take a closer look at our `ACTION_DOWN` handling in `onTouchEvent()`:

```java
if (action==MotionEvent.ACTION_DOWN) {
  for (OverlayItem item : items) {
    Point p=new Point(0,0);

    map.getProjection().toPixels(item.getPoint(), p);

    if (hitTest(item, marker, x-p.x, y-p.y)) {
      result=true;
      inDrag=item;
      items.remove(inDrag);
      populate();

      xDragTouchOffset=0;
      yDragTouchOffset=0;

      setDragImagePosition(p.x, p.y);
      dragImage.setVisibility(View.VISIBLE);

      xDragTouchOffset=x-p.x;
      yDragTouchOffset=y-p.y;

      break;
    }
  }
}
```

When we get an `ACTION_DOWN` event, we iterate over the items in our `ItemizedOverlay`. For each, we determine the item's screen coordinates using the `toPixels()` method on a `Projection`, converting the latitude and longitude of the item.

To convert our touch event (x, y) coordinates to be relative to the item, we simply have to subtract the coordinates of the item from our event's

coordinates. That can then be fed into the `hitTest()` method, which will return `true` or `false` depending on whether this item is near the touch location.

Of course, identifying the item the user chose to drag is only the first step.

Dragging the Item

A drag-and-drop operation usually involves whatever the user is dragging to appear to move across the screen in concert with the user's finger, mouse, or other pointing device. In the case of our `ItemizedOverlay`, this means we want to show the steady progression of the item across the screen, so long as the user has their finger continuously sliding on the screen.

To do that, we will:

- Hide the item in the overlay when the user touches it (`ACTION_DOWN`)

- Draw the icon for the item above the map while the user is dragging it (`ACTION_MOVE`)

- Put the item back in the overlay – at the right geographic coordinates – when the user lifts their finger (`ACTION_UP`)

Hiding an overlay item is simply a matter of removing it from the `ItemizedOverlay` and calling `populate()` again.

To render our icon during the drag operation, we can add an `ImageView` to our layout, as a later child of the `RelativeLayout` holding the `MapView`, so the image appears to float over the map:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <com.google.android.maps.MapView android:id="@+id/map"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:apiKey="0mjl6OufrY-tHs6WFurtL7rsYyEMpdEqBCbyjXg"
```

```
    android:clickable="true"
  />
  <ImageView android:id="@+id/drag"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/marker"
    android:visibility="gone"
  />
</RelativeLayout>
```

Then, after we remove the item from the overlay, we take the formerly-hidden `ImageView`, make it visible, and position it based on where the item had been a moment ago on the screen. This requires a pair of offset values:

- We need to know where in the image the point of our push-pin is (xDragImageOffset, yDragImageOffset)

- We need to know where, relative to the image, the user put their finger (xDragTouchOffset, yDragTouchOffset)

The values for `xDragImageOffset` and `yDragImageOffset` do not change, so long as we are using the same icon. Hence, we can calculate these once, up in our `SitesOverlay` constructor:

```
xDragImageOffset=dragImage.getDrawable().getIntrinsicWidth()/2;
yDragImageOffset=dragImage.getDrawable().getIntrinsicHeight();
```

The values for `xDragTouchOffset` and `yDragTouchOffset` are based on where the item is and where the finger touched the screen. We wind up with:

```
        setDragImagePosition(p.x, p.y);
         dragImage.setVisibility(View.VISIBLE);

         xDragTouchOffset=x-p.x;
         yDragTouchOffset=y-p.y;
```

This relies on some calculations in a `setDragImagePosition()` method on `SitesOverlay`:

```
private void setDragImagePosition(int x, int y) {
  RelativeLayout.LayoutParams lp=
    (RelativeLayout.LayoutParams)dragImage.getLayoutParams();

  lp.setMargins(x-xDragImageOffset-xDragTouchOffset,
```

```
                    y-yDragImageOffset-yDragTouchOffset, 0, 0);
    dragImage.setLayoutParams(lp);
}
```

Whenever we receive an ACTION_MOVE while we are dragging an item, we simply reposition our ImageView to the new location, using the pre-computed offsets:

```
else if (action==MotionEvent.ACTION_MOVE && inDrag!=null) {
  setDragImagePosition(x, y);
  result=true;
}
```

Finally, when the user lifts their finger and we get an ACTION_UP (while we are dragging an item), we can hide the ImageView, convert the final screen coordinate back into latitude and longitude, and put our item back in the ItemizedOverlay at that position:

```
else if (action==MotionEvent.ACTION_UP && inDrag!=null) {
  dragImage.setVisibility(View.GONE);

  GeoPoint pt=map.getProjection().fromPixels(x-xDragTouchOffset,
                                  y-yDragTouchOffset);
  OverlayItem toDrop=new OverlayItem(pt, inDrag.getTitle(),
                              inDrag.getSnippet());

  items.add(toDrop);
  populate();

  inDrag=null;
  result=true;
}
```

Note that this sample only supports dragging via a single finger – in other words, it does not support multi-touch operations.

Creating Custom Dialogs and Preferences

Android ships with a number of dialog classes for specific circumstances, like `DatePickerDialog` and `ProgressDialog`. Similarly, Android comes with a smattering of `Preference` classes for your `PreferenceActivity`, to accept text or selections from lists and so on.

However, there is plenty of room for improvement in both areas. As such, you may find the need to create your own custom dialog or preference class. This chapter will show you how that is done.

We start off by looking at creating a custom AlertDialog, not by using `AlertDialog.Builder` (as shown in *The Busy Coder's Guide to Android Development*), but via a custom subclass. Then, we show how to create your own dialog-style Preference, where tapping on the preference pops up a dialog to allow the user to customize the preference value.

Your Dialog, Chocolate-Covered

For your own application, the simplest way to create a custom `AlertDialog` is to use `AlertDialog.Builder`. You do not need to create any special subclass – just call methods on the `Builder`, then `show()` the resulting dialog.

However, if you want to create a reusable `AlertDialog`, this may become problematic. For example, where would this code to create the custom `AlertDialog` reside?

So, in some cases, you may wish to extend `AlertDialog` and supply the dialog's contents that way, which is how `TimePickerDialog` and others are implemented. Unfortunately, this technique is not well documented. This section will illustrate how to create such an `AlertDialog` subclass, as determined by looking at how the core Android team did it for their own dialogs.

The sample code is `ColorMixerDialog`, a dialog wrapping around the `ColorMixer` widget shown in a previous chapter. The implementation of `ColorMixerDialog` can be found in the CWAC-ColorMixer GitHub repository, as it is part of the CommonsWare Android Components.

Using this dialog works much like using `DatePickerDialog` or `TimePickerDialog`. You create an instance of `ColorMixerDialog`, supplying the initial color to show and a listener object to be notified of color changes. Then, call `show()` on the dialog. If the user makes a change and accepts the dialog, your listener will be informed.
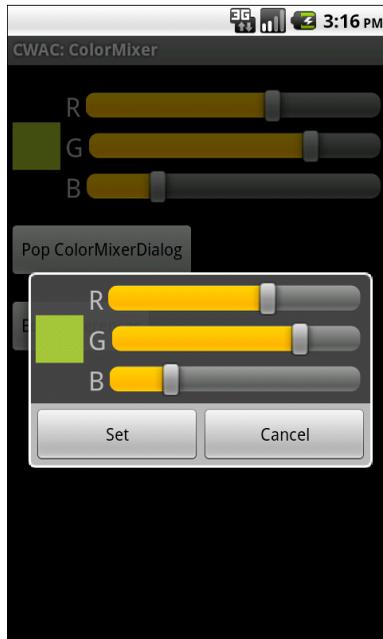
Figure 33. The ColorMixerDialog

Basic AlertDialog Setup

The `ColorMixerDialog` class is actually delightfully short, since all of the actual color mixing is handled by the `ColorMixer` widget:

```
package com.commonsware.cwac.colormixer;

import android.app.AlertDialog;
import android.content.Context;
import android.content.DialogInterface;
import android.os.Bundle;
import com.commonsware.cwac.parcel.ParcelHelper;


public class ColorMixerDialog extends AlertDialog
  implements DialogInterface.OnClickListener {
  static private final String COLOR="c";
  private ColorMixer mixer=null;
  private int initialColor;
  private ColorMixer.OnColorChangedListener onSet=null;

  public ColorMixerDialog(Context ctxt,
                          int initialColor,
```

```
                          ColorMixer.OnColorChangedListener onSet) {
    super(ctxt);

    this.initialColor=initialColor;
    this.onSet=onSet;

    ParcelHelper parcel=new ParcelHelper("cwac-colormixer", ctxt);

    mixer=new ColorMixer(ctxt);
    mixer.setColor(initialColor);

    setView(mixer);
    setButton(ctxt.getText(parcel.getIdentifier("set", "string")),
            this);
    setButton2(ctxt.getText(parcel.getIdentifier("cancel", "string")),
            (DialogInterface.OnClickListener)null);
  }

  @Override
  public void onClick(DialogInterface dialog, int which) {
    if (initialColor!=mixer.getColor()) {
      onSet.onColorChange(mixer.getColor());
    }
  }

  @Override
  public Bundle onSaveInstanceState() {
    Bundle state=super.onSaveInstanceState();

    state.putInt(COLOR, mixer.getColor());

    return(state);
  }

  @Override
  public void onRestoreInstanceState(Bundle state) {
    super.onRestoreInstanceState(state);

    mixer.setColor(state.getInt(COLOR));
  }
}
```

We simply extend the `AlertDialog` class and implement a constructor of our own design. In this case, we take in three parameters:

- A `Context` (typically an `Activity`), needed for the superclass

- The initial color to use for the dialog, such as if the user is editing a color they chose before

- A `ColorMixer.OnColorChangedListener` object, just like `ColorMixer` uses, to notify the dialog creator when the color is changed

We then create a `ColorMixer` and call `setView()` to make that be the main content of the dialog. We also call `setButton()` and `setButton2()` to specify a "Set" and "Cancel" button for the dialog. The latter just dismisses the dialog, so we need no event handler. The former we route back to the `ColorMixerDialog` itself, which implements the `DialogInterface.OnClickListener` interface.

This class is part of a parcel, designed to be reused by many projects. Hence, we cannot simply reference standard resources via the `R.` syntax – rather, we use a `ParcelHelper` to find out the right resource IDs on the fly at runtime. More information on why this is needed can be found in the chapter on reusable components.

Handling Color Changes

When the user clicks the "Set" button, we want to notify the application about the color change...if the color actually changed. This is akin to `DatePickerDialog` and `TimePickerDialog` only notifying you of date or times if the user clicks Set and actually changed the values.

The `ColorMixerDialog` tracks the initial color via the `initialColor` data member. In the `onClick()` method – required by `DialogInterface.OnClickListener` – we see if the mixer has a different color than the `initialColor`, and if so, we call the supplied `ColorMixer.OnColorChangedListener` callback object:

```
@Override
public void onClick(DialogInterface dialog, int which) {
  if (initialColor!=mixer.getColor()) {
    onSet.onColorChange(mixer.getColor());
  }
}
```

State Management

Dialogs use `onSaveInstanceState()` and `onRestoreInstanceState()`, just like activities do. That way, if the screen is rotated, or if the hosting activity is

being evicted from RAM when it is not in the foreground, the dialog can save its state, then get it back later as needed.

The biggest difference with `onSaveInstanceState()` for a dialog is that the `Bundle` of state data is not passed into the method. Rather, you get the `Bundle` by chaining to the superclass, then adding your data to the `Bundle` it returned, before returning it yourself:

```
@Override
public Bundle onSaveInstanceState() {
  Bundle state=super.onSaveInstanceState();

  state.putInt(COLOR, mixer.getColor());

  return(state);
}
```

The `onRestoreInstanceState()` pattern is much closer to the implementation you would find in an `Activity`, where the `Bundle` with the state data to restore is passed in as a parameter:

```
@Override
public void onRestoreInstanceState(Bundle state) {
  super.onRestoreInstanceState(state);

  mixer.setColor(state.getInt(COLOR));
}
```

Preferring Your Own Preferences, Preferably

The Android Settings application, built using the `Preference` system, has lots of custom `Preference` classes. You too can create your own `Preference` classes, to collect things like dates, numbers, or colors. Once again, though, the process of creating such classes is not well documented. This section reviews one recipe for making a `Preference` – specifically, a subclass of `DialogPreference` – based on the implementation of other `Preference` classes in Android.

The result is `ColorPreference`, a `Preference` that uses the `ColorMixer` widget. As with the `ColorMixerDialog` from the previous section, the `ColorPreference`

is from the CommonsWare Android Components, and its source code can be found in the CWAC-ColorMixer GitHub repository.

One might think that `ColorPreference`, as a subclass of `DialogPreference`, might use `ColorMixerDialog`. However, that is not the way it works, as you will see.

The Constructor

A Preference is much like a custom View, in that there are a variety of constructors, some taking an `AttributeSet` (for the preference properties), and some taking a default style. In the case of `ColorPreference`, we need to get the string resources to use for the names of the buttons in the dialog box, providing them to `DialogPreference` via `setPositiveButtonText()` and `setNegativeButtonText()`. Since `ColorPreference` is part of a parcel, it uses the parcel system to look up the string resource – a custom `Preference` that would be just part of a project could just use `getString()` directly.

Here, we just implement the standard two-parameter constructor, since that is the one that is used when this preference is inflated from a preference XML file:

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

  ParcelHelper parcel=new ParcelHelper("cwac-colormixer", ctxt);

  setPositiveButtonText(ctxt.getText(parcel.getIdentifier("set", "string")));
  setNegativeButtonText(ctxt.getText(parcel.getIdentifier("cancel", "string")));
}
```

Creating the View

The `DialogPreference` class handles the pop-up dialog that appears when the preference is clicked upon by the user. Subclasses get to provide the `View` that goes inside the dialog. This is handled a bit reminiscent of a `CursorAdapter`, in that there are two separate methods to be overridden:

1. `onCreateDialogView()` works like `newView()` of `CursorAdapter`, returning a `View` that should go in the dialog

2. `onBindDialogView()` works like `bindView()` of `CursorAdapter`, where the custom `Preference` is supposed to configure the `View` for the current preference value

In the case of `ColorPreference`, we use a `ColorMixer` for the `View`:

```
@Override
protected View onCreateDialogView() {
  mixer=new ColorMixer(getContext());

  return(mixer);
}
```

Then, in `onBindDialogView()`, we set the mixer's color to be `lastColor`, a private data member:

```
@Override
protected void onBindDialogView(View v) {
  super.onBindDialogView(v);

  mixer.setColor(lastColor);
}
```

We will see later in this section where `lastColor` comes from – for the moment, take it on faith that it holds the user's chosen color, or a default value.

Dealing with Preference Values

Of course, the whole point behind a `Preference` is to allow the user to set some value that the application will then use later on. Dealing with values is a bit tricky with `DialogPreference`, but not too bad.

Getting the Default Value

The preference XML format has an `android:defaultValue` attribute, which holds the default value to be used by the preference. Of course, the actual

data type of the value will differ widely – an `EditTextPreference` might expect a `String`, while `ColorPreference` needs a color value.

Hence, you need to implement `onGetDefaultValue()`. This is passed a `TypedArray` – similar to how a custom `View` uses a `TypedArray` for getting at its custom attributes in an XML layout file. It is also passed an index number into the array representing `android:defaultValue`. The custom `Preference` needs to return an `Object` representing its interpretation of the default value.

In the case of `ColorPreference`, we simply get an integer out of the `TypedArray`, representing the color value, with an overall default value of `0xFFA4C639` (a.k.a., Android green):

```
@Override
protected Object onGetDefaultValue(TypedArray a, int index) {
  return(a.getInt(index, 0xFFA4C639));
}
```

Setting the Initial Value

When the user clicks on the preference, the `DialogPreference` supplies the last-known preference value to its subclass, or the default value if this preference has not been set by the user to date.

The way this works is that the custom `Preference` needs to override `onSetInitialValue()`. This is passed in a boolean flag (`restoreValue`) indicating whether or not the user set the value of the preference before. It is also passed the `Object` returned by `onGetDefaultValue()`. Typically, a custom `Preference` will look at the flag and choose to either use the default value or load the already-set preference value.

To get the existing value, `Preference` defines a set of type-specific getter methods – `getPersistedInt()`, `getPersistedString()`, etc. So, `ColorPreference` uses `getPersistedInt()` to get the saved color value:

```
@Override
protected void onSetInitialValue(boolean restoreValue, Object defaultValue) {
```

```
  lastColor=(restoreValue ? getPersistedInt(lastColor) : (Integer)defaultValue);
}
```

Here, onSetInitialValue() stores that value in lastColor – which then winds up being used by onBindDialogView() to tell the ColorMixer what color to show.

Closing the Dialog

When the user closes the dialog, it is time to persist the chosen color from the ColorMixer. This is handled by the onDialogClosed() callback method on your custom Preference:

```
@Override
  protected void onDialogClosed(boolean positiveResult) {
    super.onDialogClosed(positiveResult);

    if (positiveResult) {
      if (callChangeListener(mixer.getColor())) {
        lastColor=mixer.getColor();
        persistInt(lastColor);
      }
    }
  }
```

The passed-in boolean indicates if the user accepted or dismissed the dialog, so you can elect to skip saving anything if the user dismissed the dialog. The other DialogPreference implementations also call callChangeListener(), which is somewhat ill-documented. Assuming both the flag and callChangeListener() are true, the Preference should save its value to the persistent store via persistInt(), persistString(), or kin.

Using the Preference

Given all of that, using the custom Preference class in an application is almost anti-climactic. You simply add it to your preference XML, with a fully-qualified class name:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <com.commonsware.cwac.colormixer.ColorPreference
```

```
    android:key="favoriteColor"
    android:defaultValue="0xFFA4C639"
    android:title="Your Favorite Color"
    android:summary="Blue.  No yel--  Auuuuuuuugh!" />
</PreferenceScreen>
```

At this point, it behaves no differently than does any other `Preference` type. Since `ColorPreference` stores the value as an integer, your code would use `getInt()` on the `SharedPreferences` to retrieve the value when needed.

The user sees an ordinary preference entry in the `PreferenceActivity`:
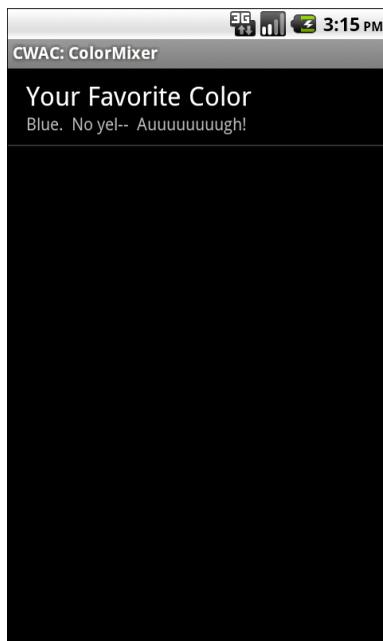
Figure 34. A PreferenceActivity, showing the ColorPreference

When tapped, it brings up the mixer:

137
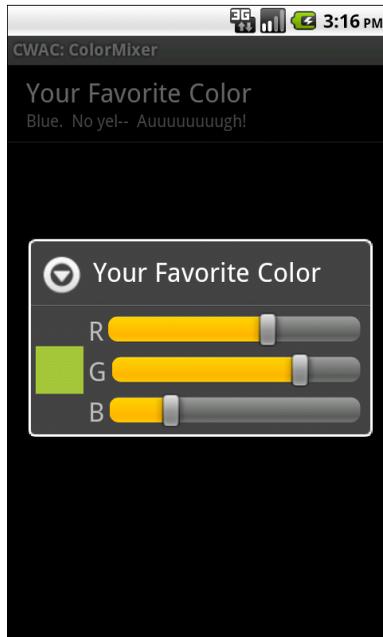
Figure 35. The ColorMixer in a custom DialogPreference

Choosing a color and pressing the BACK button persists the color value as a preference.

Advanced Fragments and the Action Bar

The Android UI framework introduced in the "Honeycomb" release (Android 3.0) is a significant departure from the programming practices we used with Android 1.x and 2.x devices. While those apps still work, and in many cases need no changes, other apps will benefit from using the action bar and fragments, particularly with an eye towards supporting multiple screen sizes.

The assumption is that you know the basics of how the action bar and fragments work, perhaps from reading *The Busy Coder's Guide to Android Development*. This chapter, therefore, examines more interesting uses of these technologies, particularly using the Android Support package to allow for backwards compatibility to Android 1.x and 2.x devices.

About the Sample App

The sample application – found in the `Honeycomb/FeedFragments` project – is somewhat more elaborate than are most samples found in this book. Hence, it is probably a good idea to get a feeling for what this application does before diving into how it leverages fragments and the action bar to achieve its ends.

Note that the explanation of the code that forms the bulk of this chapter does not go into all of the details, such as how the RSS feeds are parsed. The focus is on fragments and the action bar, less on the actual "business logic", as it were.

What the User Sees

On a Honeycomb tablet, such as a Motorola XOOM, the user is first greeted with an activity showing a list of RSS feeds down the left side of the screen, underneath a Honeycomb action bar:
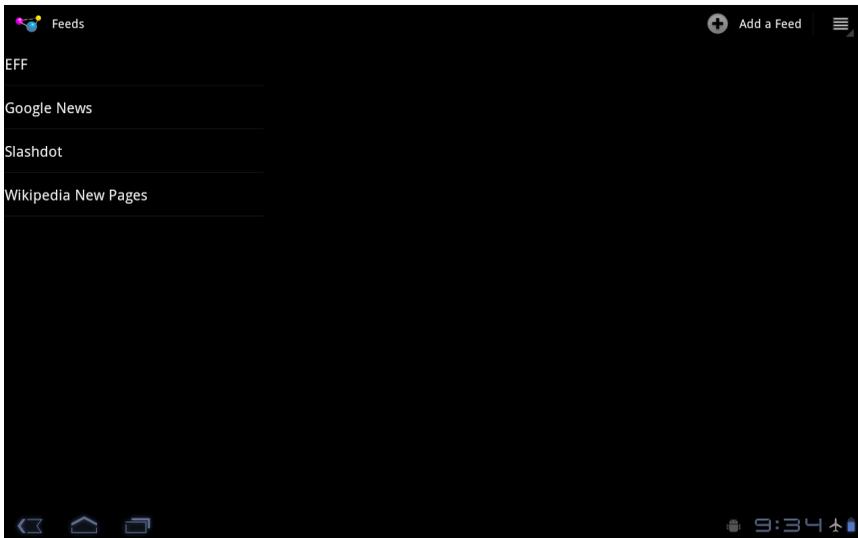
Figure 36. FeedFragments initial state on a XOOM

Tapping a feed brings up another list, showing the items in that feed, with a persistent highlight on the tapped-upon feed:

Figure 37. FeedFragments with the list of items for a tapped-upon feed

Tapping on an item displays the associated Web page on the right:
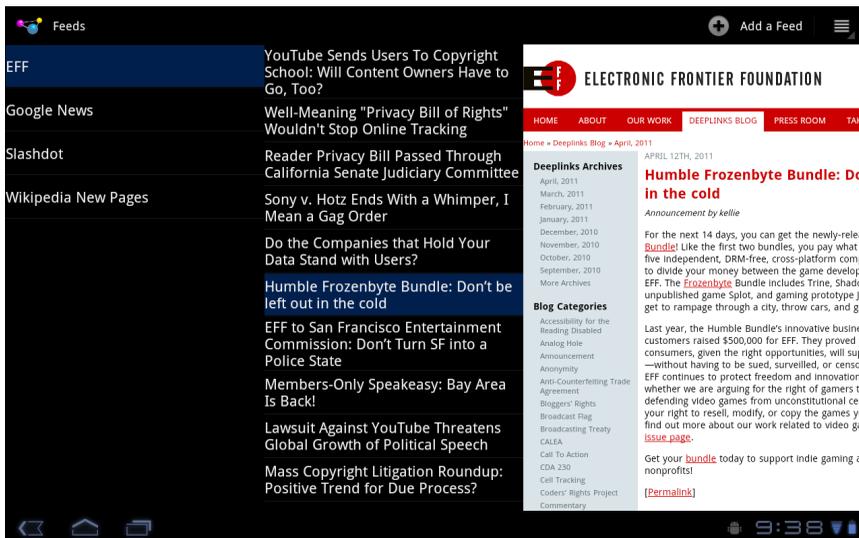
Figure 38. FeedFragments showing the page for an item

Pressing the BACK button closes each added bit of UI in reverse order. So, if you press BACK from the state shown in the above screenshot, the Web

content vanishes; pressing BACK a second time gets rid of the list of items. Pressing BACK from just the list of feeds will close up the application.

In portrait mode, the same device shows the two lists side-by-side at the top and the Web content beneath it:
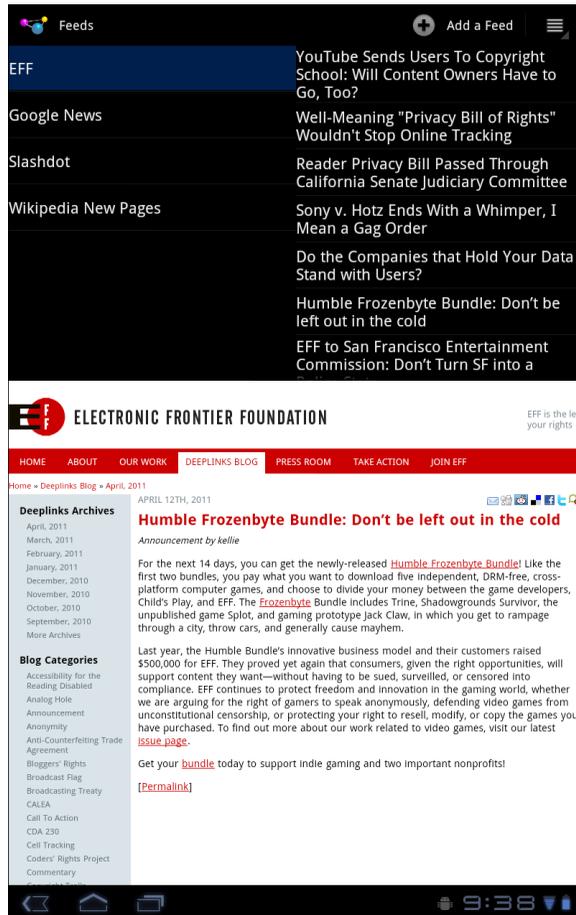
Figure 39. FeedFragments in portrait mode

The "Add a Feed" button on the action bar pops up a dialog for the user to enter a new feed name and URL:
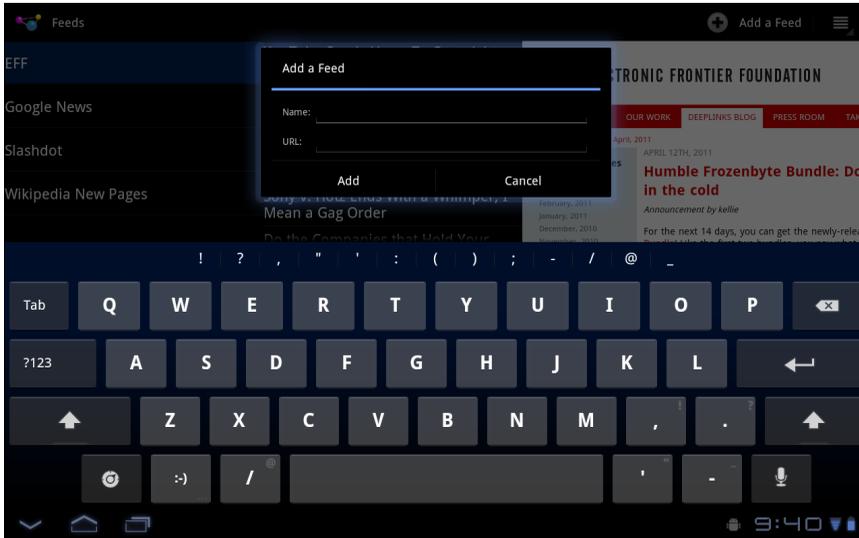
Figure 40. Adding a feed to FeedFragments

The newly-added feed shows up then in the list on the left:
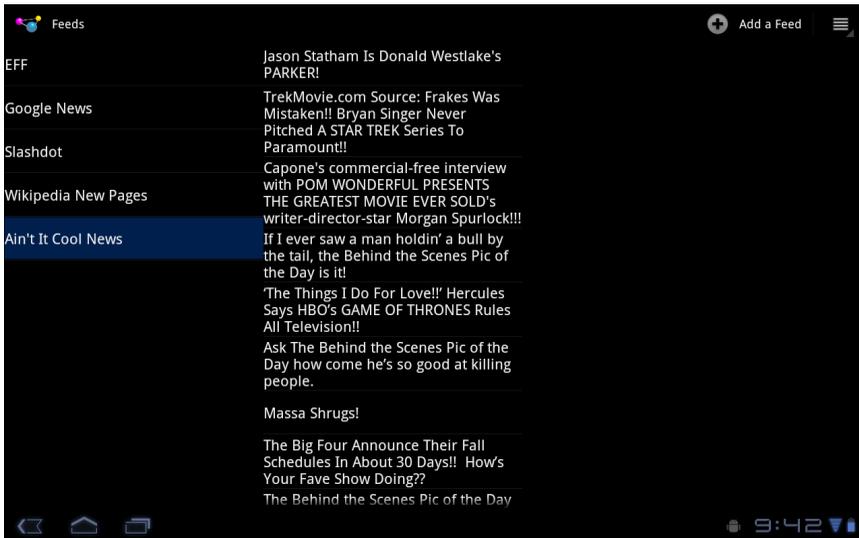
Figure 41. FeedFragments with a new feed

Tapping on the menu button in the upper-right corner shows a handful of other options:
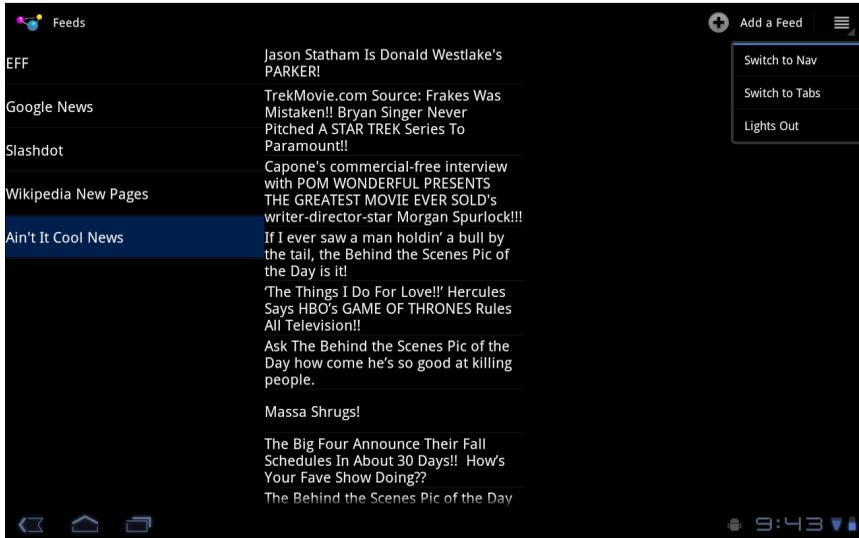
Figure 42. FeedFragments' options menu

Choosing "Switch to Nav" brings up a new activity, one where the roster of feeds appears in a Spinner in the action bar, with the default selection's items loaded on the left:
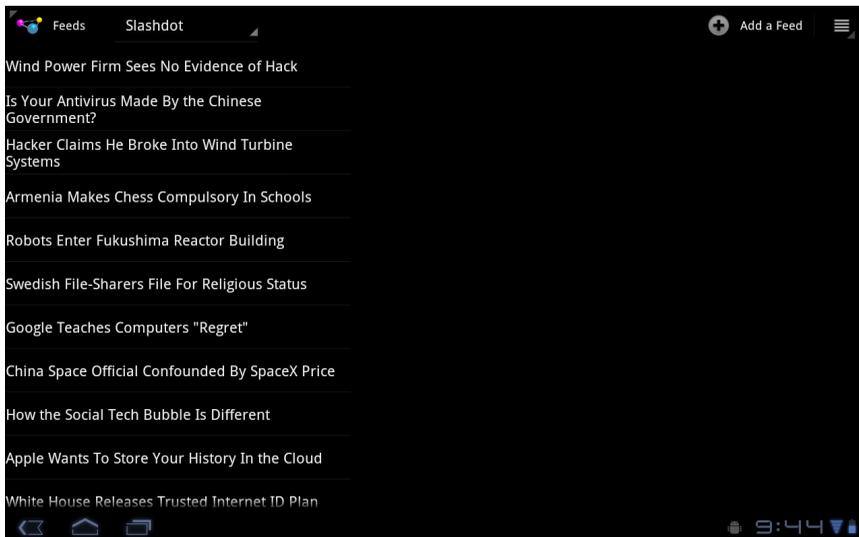
Figure 43. FeedFragments in "nav" mode

144

Tapping on any given feed item brings up the Web content on the right, as before, just a bit larger since we do not have the other list taking up its space.

Pressing the BACK button, or tapping on the logo in the upper-left corner, returns control to the original activity.

Choosing "Switch to Tabs" from the main activity's options menu brings up yet another activity, this time with the feeds shown in tabs in the action bar:
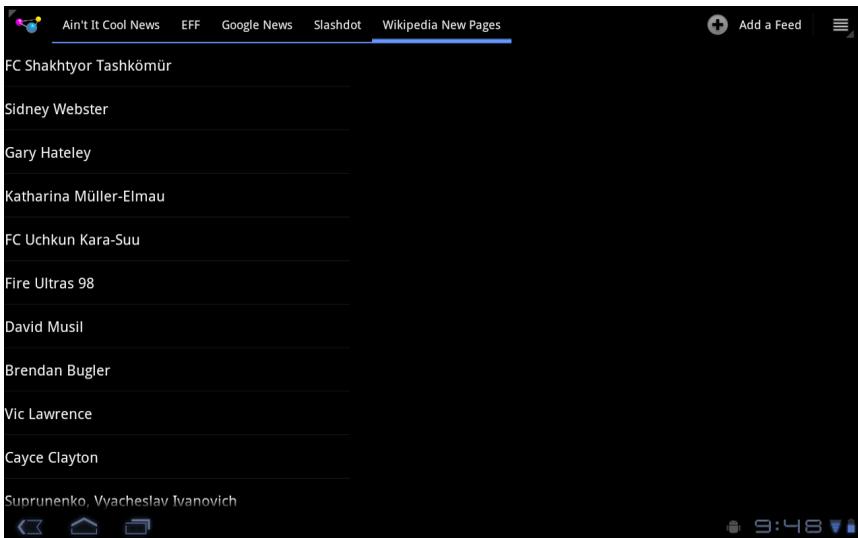
Figure 44. FeedFragments in "tabs" mode

Clicking on a tab switches the list on the left, and clicking on a feed item once again brings up the Web content on the right.

From any of the activities, choosing the "Lights Out" menu item hides the action bar and dims the system bar:
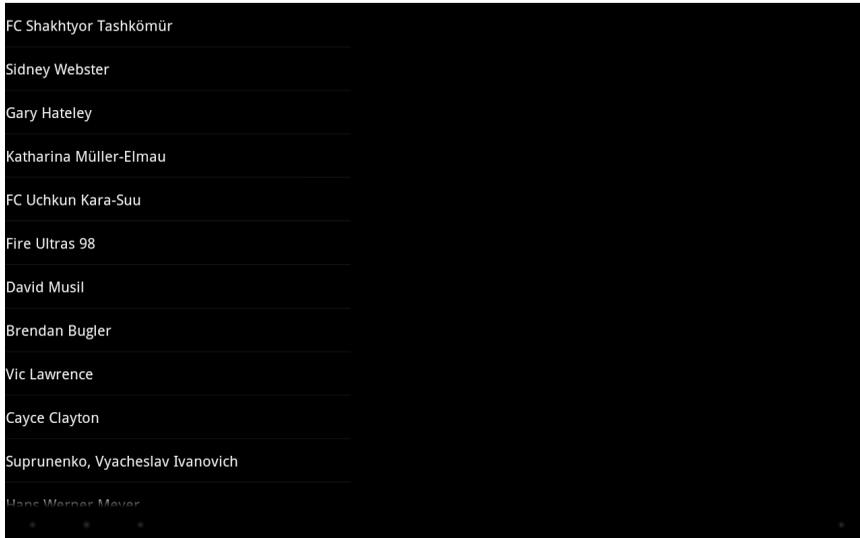
Figure 45. FeedFragments in lights-out mode

Five seconds later, they return to normal.

On a non-Honeycomb large-screen device, like the Samsung Galaxy Tab, we get the same basic three-pane UI as before:
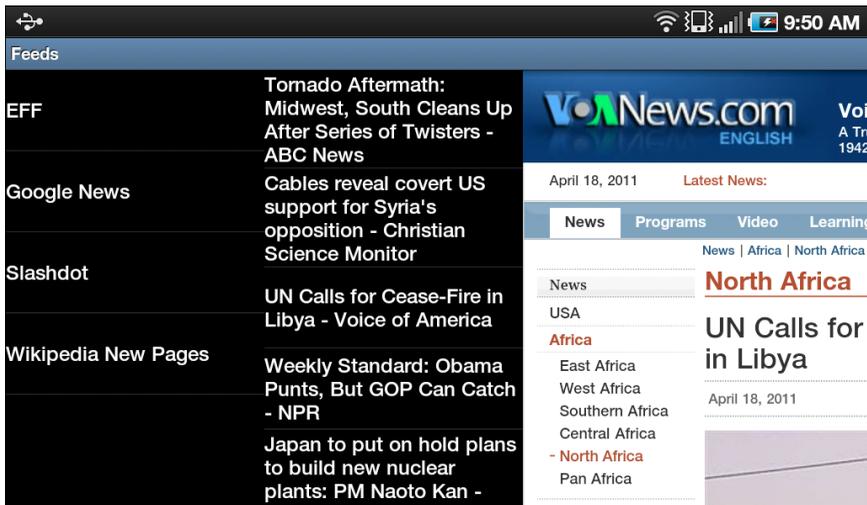
Figure 46. FeedFragments on a Galaxy Tab

However:

- There is no persistent highlight in the lists
- The options menu just has an "Add Feed" item, without the ability to switch to other versions of the activity or to go into "lights out" mode

On a normal or small-screen device, the UI breaks those three panes into three separate screens, with the Web content being brought up in the default browser:
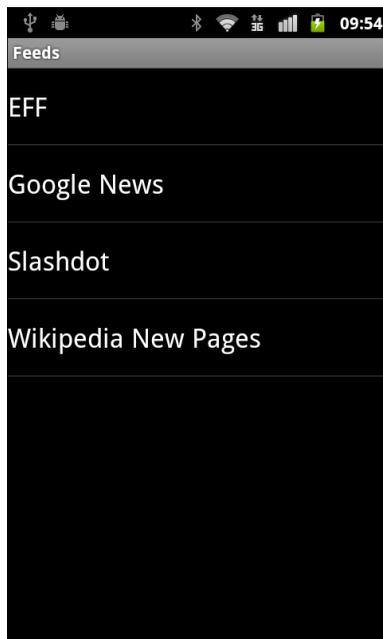
Figure 47. FeedFragments' list of feeds, on a phone

147
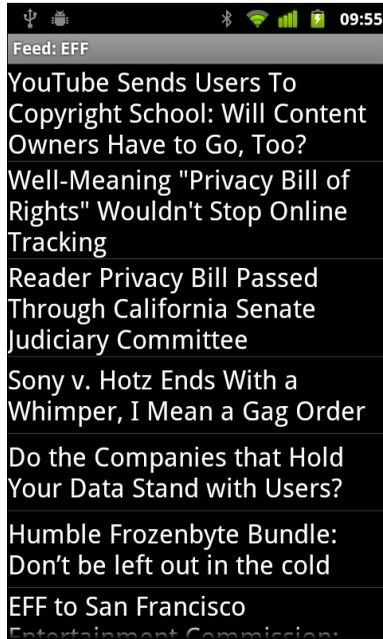
Figure 48. FeedFragments' list of items, on a phone

Only the list-of-feeds screen has the "Add a Feed" options menu item, and you cannot switch to list or tabs mode.

The Data Model (Such As It Is and What There Is Of It)

The FeedFragments project has a Feed class that represents the data model, consisting of a display name and a URL per feed:

```
package com.commonsware.android.feedfrags;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;

public class Feed implements Comparable<Feed> {
  static private HashMap<String, Feed> FEEDS=new HashMap<String, Feed>();

  static {
    addFeed("Slashdot", "http://rss.slashdot.org/Slashdot/slashdot");
    addFeed("Wikipedia New Pages",
```

```
                "http://en.wikipedia.org/w/index.php?
title=Special:NewPages&feed=rss");
    addFeed("EFF", "http://www.eff.org/rss/updates.xml");
    addFeed("Google News", "http://news.google.com/news?
pz=1&cf=all&ned=us&hl=en&output=rss");
  }

  private String name;
  private String url;

  public static Feed addFeed(String name, String url) {
    Feed result=new Feed(name, url);

    addFeed(result);

    return(result);
  }

  private static void addFeed(Feed feed) {
    FEEDS.put(feed.getKey(), feed);
  }

  public static ArrayList<Feed> getFeeds() {
    ArrayList<Feed> result=new ArrayList<Feed>(FEEDS.values());

    Collections.sort(result);

    return(result);
  }

  public static Feed getFeed(String key) {
    return(FEEDS.get(key));
  }

  private Feed(String name, String url) {
    this.name=name;
    this.url=url;
  }

  public String getKey() {
    return(toString());
  }

  public String toString() {
    return(name);
  }

  public String getUrl() {
    return(url);
  }

  public int compareTo(Feed another) {
    return(toString().compareTo(another.toString()));
  }
```

```
}
```

`Feed` holds onto a static `HashMap` of defined feeds, initializing a handful of them when the class is loaded. Other feeds can be defined at runtime, though since these are only stored in the `HashMap`, they will evaporate when the process is terminated. A more sophisticated application would use a database – or some other persistence approach – for storing the feeds. The implementation of this is left as an exercise for the reader.

Dynamic Fragments

The common theme among most of these interactions is having fragments that come and go based on user input. We go from one pane to two panes to three panes in the original activity. Also, we switch out fragments based upon the selection of tabs or action bar list items in the other two main activities. This provides a nice feel to the app – stuff comes and goes based upon context – but is clearly a bit more involved than simply having one or two of fragments always around, based solely on screen size.

Let's first focus on the original activity, where we have a list of feeds, possibly a list of items in a selected feed, and possibly the Web content for a selected item.

Fragments and Panes

Fragments can be "wired into" a layout by using the `<fragment>` element. These fragments will remain in the activity's UI until the activity is destroyed. So, for example, the `ListFragment` implementing the list of feeds can be a permanent fixture of its activity, as it is always present.

However, for fragments that come and go, your layout still needs to provide space for them. For example, you can use a `FrameLayout` as the container, providing a "pane" into which a fragment can appear as needed.

For example, the layout used for the original activity – `FeedsActivity` – can be found as the `main.xml` layout resource. However, there are three versions

of that resource, one for large screens in landscape (`res/layout-large-land/main.xml`), one for large screens not in landscape (`res/layout-large/main.xml`), and one for everything else (`res/layout/main.xml`). Only the first two contain the panes for the dynamic fragments – the latter layout is used for smaller screens where only one fragment will be visible.

The large-and-landscape version of the layout looks like this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal"
  android:weightSum="100"
>
  <fragment
    class="com.commonsware.android.feedfrags.FeedsFragment"
    android:id="@+id/feeds"
    android:layout_width="0dip"
    android:layout_height="match_parent"
    android:layout_weight="30"
  />
  <FrameLayout
    android:id="@+id/second_pane"
    android:layout_width="0dip"
    android:layout_height="match_parent"
    android:layout_weight="30"
  />
  <FrameLayout
    android:id="@+id/third_pane"
    android:layout_width="0dip"
    android:layout_height="match_parent"
    android:layout_weight="40"
  />
</LinearLayout>
```

Here, we have a horizontal `LinearLayout` holding our original fragment (`FeedsFragment`), plus two `FrameLayout` containers for our other two fragments. The sizes are dictated on a percentage basis by using `android:layout_weight`, so the right-most fragment will get 40% of the screen, while the other two fragments will get 30% apiece.

For large screens not in landscape mode, we have a related layout:

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

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:weightSum="100"
>
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="40"
    android:orientation="horizontal"
    android:weightSum="100"
  >
    <fragment
      class="com.commonsware.android.feedfrags.FeedsFragment"
      android:id="@+id/feeds"
      android:layout_width="0dip"
      android:layout_height="match_parent"
      android:layout_weight="50"
    />
    <FrameLayout
      android:id="@+id/second_pane"
      android:layout_width="0dip"
      android:layout_height="match_parent"
      android:layout_weight="50"
    />
  </LinearLayout>
  <FrameLayout
    android:id="@+id/third_pane"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="60"
  />
</LinearLayout>
```

In this case, a vertical `LinearLayout` splits the screen between the lists (themselves laid out horizontally) and the content below.

Fragments and Activities

`FeedsActivity` is the entry point of our app, the activity that will be shown in the launcher. It will either support dynamic fragments or not, depending on whether we are loading in a layout that has panes for new fragments. If it does not, we must be on a smaller-screen device, and `FeedsActivity` will just show the list of feeds.

To accomplish this, we need a total of three fragments:

1. FeedsFragment, which is a ListFragment holding the list of feeds – this is the one hard-wired into our layouts used by FeedsActivity

2. ItemsFragment, which is a ListFragment holding the list of items in a feed

3. ContentFragment, which is a Fragment holding a WebView that displays the Web page associated with a feed item

ContentFragment could be a WebViewFragment, but that is only available in API Level 11, not in the Android Support package, at least as of the time of this writing.

However, we have a pair of base classes involved as well:

1. AbstractFeedsActivity holds common behaviors between FeedsActivity (the three-pane model), FeedsNavActivity (the list-based navigation model), and FeedsTabActivity (the tabs model). We will examine those latter two activities later in this chapter.

2. PersistentListFragment is a base class for FeedsFragment and ItemsFragment, implementing the logic to show an "activated" highlight on tapped items by marking the list row as being checked. The notion of "activated" for a persistent highlight is covered in greater depth in *The Busy Coder's Guide to Android Development*, though we will review it briefly later in this chapter.

Running a FragmentTransaction

FeedsActivity loads its layout in onCreate(), much like any other Android activity:

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

  FeedsFragment feeds
    =(FeedsFragment)getSupportFragmentManager()
                        .findFragmentById(R.id.feeds);

  feeds.setOnFeedListener(this);
```

```
    isThreePane=(null!=findViewById(R.id.second_pane));

    if (isThreePane) {
      feeds.enablePersistentSelection();
    }
}
```

Also, in onCreate(), we register FeedsActivity as being the OnFeedListener with the FeedsFragment. This listener is held onto by the FeedsFragment in a listener data member:

```
package com.commonsware.android.feedfrags;

import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class FeedsFragment extends PersistentListFragment {
  private OnFeedListener listener=null;
  private ArrayAdapter<Feed> adapter=null;
  private Bundle state=null;

  @Override
  public void onActivityCreated(Bundle state) {
    super.onActivityCreated(state);

    this.state=state;
  }

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

    loadFeeds();
    restoreState(state);
  }

  @Override
  public void onListItemClick(ListView l, View v, int position,
                              long id) {
    super.onListItemClick(l, v, position, id);

    if (listener!=null) {
      listener.onFeedSelected(adapter.getItem(position));
    }
  }

  public void addNewFeed(Feed feed) {
    adapter.add(feed);
  }
```

```
  private void loadFeeds() {
    adapter=new ArrayAdapter<Feed>(getActivity(), R.layout.row,
                                   Feed.getFeeds());
    setListAdapter(adapter);
  }

  public void setOnFeedListener(OnFeedListener listener) {
    this.listener=listener;
  }

  public interface OnFeedListener {
    void onFeedSelected(Feed feed);
  }
}
```

FeedsFragment then uses this listener to let FeedsActivity know what Feed was clicked upon in the list, in onListItemClick().

FeedsActivity implements onFeedSelected(), to honor the OnFeedListener interface. There, we need to decide what to do:

```
public void onFeedSelected(Feed feed) {
  if (isThreePane) {
    addItemsFragment(feed);
  }
  else {
    Intent i=new Intent(this, ItemsActivity.class);

    i.putExtra(ItemsActivity.EXTRA_FEED_KEY, feed.getKey());

    startActivity(i);
  }
}
```

If we do not have the three-pane layout, we have no choice but to pass control to another activity – ItemsActivity – that will display the feed's items. We will look at ItemsActivity later in this chapter.

However, if we do have the three-pane layout, we pass control to an addItemsFragment(), which will invoke a FragmentTransaction:

```
public void addItemsFragment(Feed feed) {
  FragmentManager fragMgr=getSupportFragmentManager();
  ItemsFragment items=(ItemsFragment)fragMgr.findFragmentById(R.id.second_pane);
  FragmentTransaction xaction=fragMgr.beginTransaction();
```

155

```
  if (items==null) {
    items=new ItemsFragment(true);
    items.setOnItemListener(this);

    xaction
      .add(R.id.second_pane, items)
      .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN)
      .addToBackStack(null)
      .commit();
  }
  else {
    ContentFragment content=
      (ContentFragment)fragMgr.findFragmentById(R.id.third_pane);

    if (content!=null) {
      xaction.remove(content).commit();
      fragMgr.popBackStack();
    }
  }

  items.loadUrl(feed.getUrl());
}
```

And this is where the questions begin...

What is a FragmentManager Again?

The first thing we do in addItemsFragment() is get a FragmentManager.

The FragmentManager is our gateway to the fragments system. It knows about all the active fragments in our activity, plus has the ability to process FragmentTransaction requests to change the dynamic fragments in the activity.

So, for example, after getting the FragmentManager, we immediately query it via findFragmentById() to see if an ItemsFragment has already been added:

```
FragmentManager fragMgr=getSupportFragmentManager();
ItemsFragment items=(ItemsFragment)fragMgr.findFragmentById(R.id.second_pane);
```

Here, findFragmentById() is a bit of a misnomer. However, a method named findFragmentByTheIdOfTheContainerInWhichTheFragmentShouldReside() would be a trifle long. We are supplying an ID of the container in which our

dynamic fragment will reside – in this case, the `second_pane` FrameLayout from our large-screen layouts shown in the previous section.

In a pure API Level 11 environment, any activity can call `getFragmentManager()`. For applications using the Android Support package, though, you have to inherit from a `FragmentActivity` class (handled for us by `AbstractFeedsActivity`, the parent of `FeedsActivity`) and you have to call `getSupportFragmentManager()` instead of `getFragmentManager()`.

What is a FragmentTransaction?

A `FragmentTransaction` is the means by which we modify the mix of dynamic fragments in our activity. Here, "modify the mix" means:

- Add a new dynamic fragment
- Remove an existing dynamic fragment
- Replace an existing dynamic fragment with a different one
- Hide a dynamic fragment, such that is is no longer seen but is still in the `FragmentManager`
- Show a previously hidden fragment

`FragmentTransaction` follows the builder pattern: most of the methods return the `FragmentTransaction` itself, so you can chain a bunch of operations into one long Java statement. When the transaction is fully described, a call to `commit()` will cause the results to be processed by the main application thread at the next available opportunity.

For example, if our check for an `ItemsFragment` determines that one is not already on the screen, we need to add one. So, we create an `ItemsFragment`, then create a `FragmentTransaction` and perform a series of operations upon it:

- Add the `ItemsFragment` to the `second_pane` container
- Indicate that we want the standard "open" transition animation to be applied

- Add this transaction to the back stack (more on this below)

- Commit the transaction

```
if (items==null) {
  items=new ItemsFragment(true);
  items.setOnItemListener(this);

  xaction
    .add(R.id.second_pane, items)
    .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN)
    .addToBackStack(null)
    .commit();
}
```

What Happens If An ItemsFragment Is Already Shown?

When the user fires up FeedsFragments and taps on a feed for the first time, there is no ItemsFragment, so we add one via a FragmentTransaction, as shown above.

If there already is an ItemsFragment on the screen, though, we do not need to add a second one. Nor do we need to replace the ItemsFragment with a near-identical duplicate. Instead, we can simply tell it to load the new feed via a call to loadUrl(), supplying it the URL to the feed in question. We need to call loadUrl() on a new ItemsFragment anyway, so no matter what we do in addItemsFragment(), we call loadUrl() towards the end of that method.

What's With This "Back Stack" Stuff?

When you add a dynamic fragment to the activity, you have two choices:

- Pressing the BACK button should remove that fragment

- Pressing the BACK button should not have anything to do with the fragment (and, if applied to all your dynamic fragments, means the BACK button would exit the activity)

Whether you want to have the BACK button remove dynamic fragments or not is up to you and is a UI design decision. For illustration purposes, FeedFragments has the BACK button remove dynamic fragments.

FragmentManager and your FragmentTransaction objects manage this "back stack" of fragments for you. When you invoke a FragmentTransaction, you can specifically add it to the the back stack via addToBackStack(). Then, when the user presses BACK, if there is a dynamic fragment set up added via a FragmentTransaction that is on the back stack, the BACK button press will remove that fragment. Once the fragment back stack is depleted, the next BACK button press will perform the normal behavior for an activity, in the form of finishing it.

Where Does Our Web Content Come Into Play?

Somewhere along the line, when the user taps on an item in the ItemsFragment, we are supposed to display the Web page associated with this item.

When we create our ItemsFragment, we register the FeedsActivity as being the OnItemSelected listener via a call to setOnItemListener(). ItemsFragment holds onto that listener and later calls onItemSelected() on it when the user taps on a feed item in the list.

The implementation of onItemSelected() is on the AbstractFeedsActivity class, since all three AbstractFeedsActivity subclasses have the same behavior:

```
public void onItemSelected(RSSItem item) {
  FragmentManager fragMgr=getSupportFragmentManager();
  ContentFragment content=
    (ContentFragment)fragMgr.findFragmentById(R.id.third_pane);
  FragmentTransaction xaction=fragMgr.beginTransaction();

  if (content==null || content.isRemoving()) {
    content=new ContentFragment(item.getLink().toString());

    xaction
      .add(R.id.third_pane, content)
      .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN)
      .addToBackStack(null)
      .commit();
  }
  else {
    content.loadUrl(item.getLink().toString());
```

```
  }
}
```

The recipe is similar to the one where we add the `ItemsFragment`:

- Get a `FragmentManager`

- Ask the `FragmentManager` to find our `ContentFragment` in `third_pane`

- Create a `FragmentTransaction`

- If there is no `ContentFragment`, create one and add it to the `third_pane` container, plus put it on the back stack

- If there is an existing `ContentFragment` (i.e., the user tapped on a second item in the same `ItemsFragment`), load the new URL into the `ContentFragment`

The details of `ContentFragment` itself are covered later in this chapter.

The one thing that is a bit unusual is that we are not only checking to see if we have a `ContentFragment`, but also whether that `ContentFragment` is in the process of being removed. If it is being removed as part of a previous `FragmentTransaction`, there is no point trying to load the URL into it, as it will not be visible for long, if at all. Instead, in that case as well, we add a new `ContentFragment`.

What Happens If A ContentFragment Is Already Shown?

Ah, but what happens if the user had previously tapped on a feed, then tapped on a item, bringing up the `ContentFragment`?

In this case, when the user taps on another feed, we need to get rid of the `ContentFragment`. After all, that is showing the content of some item from some other feed.

To do that, we:

- Call `findFragmentById()` again on the `FragmentManager`, this time seeking the `ContentFragment` that may be in `third_pane`.

- If that fragment exists, we create a new `FragmentTransaction`, ask to remove the fragment, and commit the transaction

```
ContentFragment content=
  (ContentFragment)fragMgr.findFragmentById(R.id.third_pane);

if (content!=null) {
  xaction.remove(content).commit();
  fragMgr.popBackStack();
}
```

Now, you might think that removing a fragment via a `FragmentTransaction` will also remove the transaction that created it from the back stack.

You would be mistaken.

Instead, we have to handle that ourselves, via a call to `popBackStack()` on `FragmentManager`. As the method name suggests, this pops the last transaction off of the back stack. If we have a `ContentFragment` live in the activity, the last transaction should have been one to put that fragment there in the first place, so we pop that transaction off the back stack, so the BACK button behaves as the user might expect.

What Happens When We Have Fragments That Are Retained?

Fragments that are retained across configuration changes via `setRetainInstance()` will not be destroyed when the screen is rotated, when the user puts the device in a dock, etc. The activity that hosts the fragments will be destroyed and recreated.

This can cause a problem with our listener approach. We need to make sure that the retained fragments are talking to the right activity instance after a configuration change.

The easiest way to do that is to fire the `setOnItemListener()` setter every time the activity comes to the foreground, via `onResume()`:

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

  FragmentManager fragMgr=getSupportFragmentManager();
  ItemsFragment items=(ItemsFragment)fragMgr.findFragmentById(R.id.second_pane);

  if (items!=null) {
    items.setOnItemListener(this);
  }
}
```

Action Bar Navigation Options

In the `FeedsActivity` described above, all navigation is handled by the activity and its fragments. More specifically, our roster of feeds is shown in a `ListFragment`.

This is not the only option.

The Honeycomb action bar is capable of providing navigation as well, in an effort to save screen space for other things. There are two built-in navigation options in the action bar: tabs and the so-called "list" mode.

Tabs Mode

In Android 1.x and 2.x, if you wanted tabs, you usually trotted out `TabHost` and `TabWidget`.

On Android 3.x, that is being replaced by tabs managed in the action bar.

Setting up tabs is fairly straightforward, once you know the recipe:

- Call `setNavigationMode(ActionBar.NAVIGATION_MODE_TABS)` on the `ActionBar`, which you get via `getActionBar()`

- Call `addTab()` on `ActionBar` for each tab you want

- Probably get rid of the title – to make room for more tabs – via a call to `setDisplayOptions()` on the `ActionBar`

For example, `FeedsTabActivity` is an `AbstractFeedsActivity` that uses two panes in its layout instead of the original three:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal"
  android:weightSum="100"
>
  <FrameLayout
    android:id="@+id/second_pane"
    android:layout_width="0dip"
    android:layout_height="match_parent"
    android:layout_weight="40"
  />
  <FrameLayout
    android:id="@+id/third_pane"
    android:layout_width="0dip"
    android:layout_height="match_parent"
    android:layout_weight="60"
  />
</LinearLayout>
```

Rather than hard-wire in a `FeedsFragment`, the feeds will be represented as individual tabs. The basics are set up in `onCreate()`:

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

  for (final Feed feed : Feed.getFeeds()) {
    addNewFeed(feed);
  }

  ActionBar bar=getActionBar();

  bar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
  bar.setDisplayOptions(0, ActionBar.DISPLAY_SHOW_TITLE);
  bar.setDisplayHomeAsUpEnabled(true);

  int screenLayout=getResources()
                    .getConfiguration()
                    .screenLayout;

  if (((screenLayout &
Configuration.SCREENLAYOUT_SIZE_NORMAL)==Configuration.SCREENLAYOUT_SIZE_NORMAL)

      || ((screenLayout &
Configuration.SCREENLAYOUT_SIZE_SMALL)==Configuration.SCREENLAYOUT_SIZE_SMALL))
```

163

```
{
    bar.setDisplayShowHomeEnabled(false);
  }
}
```

The call to `setDisplayHomeAsUpEnabled()` will be discussed later in this chapter.

The tabs themselves are added via the `addNewFeed()` method:

```
public void addNewFeed(final Feed feed) {
  ActionBar bar=getActionBar();

  bar.addTab(bar
              .newTab()
              .setText(feed.toString())
              .setTabListener(new TabListener(feed)));
}
```

Here, `addNewFeed()` calls `newTab()` on the `ActionBar`, which gives us an `ActionBar.Tab` object. As with `FragmentTransaction`, `ActionBar.Tab` uses the builder pattern, where the setters all return the `ActionBar.Tab` itself, facilitating chained method calls. Here, we are setting a text caption on the tab (though you have options for an icon or a custom `View`, as with `TabHost`), plus attaching an `ActionBar.TabListener` object to the tab. The listener will get control on the lifecycle of the tab:

- `onTabSelected()` is called when the tab is selected by the user

- `onTabUnselected()` is called when some other tab is selected by the user

- `onTabReselected()` is called, presumably, when the user taps on an already-selected tab (e.g., to refresh the tab's contents)

Our implementation ignores the latter and focuses on the first two:

```
 private class TabListener implements ActionBar.TabListener {
    Feed feed=null;

    TabListener(Feed feed) {
      this.feed=feed;
    }
```

```
    public void onTabSelected(ActionBar.Tab tab,
                              android.app.FragmentTransaction unused) {
      FragmentManager fragMgr=getSupportFragmentManager();
      FragmentTransaction xaction=fragMgr.beginTransaction();

      addItems(xaction, feed);
      xaction.commit();
    }

    public void onTabUnselected(ActionBar.Tab tab,
                                android.app.FragmentTransaction unused) {
      FragmentManager fragMgr=getSupportFragmentManager();
      FragmentTransaction xaction=fragMgr.beginTransaction();

      removeFragments(fragMgr, xaction);
      xaction.commit();
    }

    public void onTabReselected(ActionBar.Tab tab,
                                android.app.FragmentTransaction xaction) {
      // NO-OP
    }
  }
```

You will notice that the declarations of these methods supply an `android.app.FragmentTransaction` object. The vast majority of code samples in the CommonsWare books use `import` statements for packages, rather than attaching the package as part of the class name. Here, though, the full `android.app.FragmentTransaction` name is used in these methods to highlight a very important point:

These are not the `FragmentTransaction` objects you are looking for.

`FeedFragments` is implemented using the Android Support package. The `FragmentTransaction` being used by the Android Support package is `android.support.v4.app.FragmentTransaction`, not `android.app.FragmentTransaction`. The problem is that the Android Support package cannot replace the standard API Level 11+ action bar, nor its `TabListener` interface. Hence, Honeycomb will happily pass us an `android.app.FragmentTransaction` object, but we cannot use it with our Android Support package-based fragments.

Instead, we ignore it.

Our listener creates its own FragmentTransaction, using the Android Support package implementation, and adds and removes the ItemsFragment. The addItems() method is from AbstractFeedActivity:

```
protected void addItems(FragmentTransaction xaction, Feed feed) {
  ItemsFragment items=new ItemsFragment(true);

  items.setOnItemListener(this);
  items.loadUrl(feed.getUrl());

  xaction.add(R.id.second_pane, items, "items");
}
```

Conversely, removeFragments() is implemented on FeedsTabActivity itself:

```
private void removeFragments(FragmentManager fragMgr,
                             FragmentTransaction xaction) {
  ItemsFragment items=(ItemsFragment)fragMgr.findFragmentById(R.id.second_pane);

  if (items!=null) {
    xaction.remove(items);

    ContentFragment content=
      (ContentFragment)fragMgr.findFragmentById(R.id.third_pane);

    if (content!=null && !content.isRemoving()) {
      xaction.remove(content);
      fragMgr.popBackStack();
    }
  }
}
```

Here, we see if the ItemsFragment already exists. If it does, we remove it via our FragmentTransaction, plus we check to see if the ContentFragment is already here. If it exists, we remove it too, also popping it off the back stack. This way, when the user switches tabs, they see only the new list of items, and the content – if any – is removed.

If we were using the supplied android.app.FragmentTransaction object, we would not need to commit() the transaction ourselves – the ActionBar handles that for us. Since we are creating our own Android Support package-style FragmentTransaction, we need to make the commit() call.

"List" Mode

Android's action bar also supports a "list navigation" option. Despite the name, the "list" is really a Spinner, hosted in the action bar. Just as you can flip between fragments using tabs when in tab navigation mode, list navigation mode has you flip between fragments based upon the selection in the Spinner.

To set this up:

- Call `setNavigationMode(ActionBar.NAVIGATION_MODE_LIST)` on the `ActionBar` to enable the list navigation mode

- Call `setListNavigationCallbacks()` on the `ActionBar`, simultaneously supplying the `SpinnerAdapter` to use to populate the `Spinner` and an `ActionBar.OnNavigationListener` object to be notified when there is a selection change in the `Spinner`

In `FeedsNavActivity`, these things are done as part of the `onCreate()` method:

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

  ActionBar bar=getActionBar();

  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
    adapter=
        new ArrayAdapter<Feed>(bar.getThemedContext(), R.layout.row,
                               Feed.getFeeds());
  }
  else {
    adapter=
        new ArrayAdapter<Feed>(this, R.layout.row, Feed.getFeeds());
  }

  bar.setListNavigationCallbacks(adapter, new NavListener());
  bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
  bar.setDisplayOptions(ActionBar.DISPLAY_SHOW_TITLE,
                        ActionBar.DISPLAY_SHOW_TITLE);
  bar.setDisplayHomeAsUpEnabled(true);
}
```

Android 4.0 (Ice Cream Sandwich) offers a `getThemedContext()` method on `ActionBar`. Use the `Context` returned by this method when working with resources that relate to the `ActionBar`. In this case, we use it when creating our `ArrayAdapter` to use with the `Spinner`. However, since this is only available on API Level 14 and higher, you need to check for that and fall back to using the `Activity` as your `Context` for earlier versions of Android.

The `ActionBar.OnNavigationListener` object needs to implement `onNavigationItemSelected()`, which in turn needs to update the UI to reflect the new selection:

```java
private class NavListener implements ActionBar.OnNavigationListener {
   public boolean onNavigationItemSelected(int itemPosition,
                                           long itemId) {
     FragmentManager fragMgr=getSupportFragmentManager();
     FragmentTransaction xaction=fragMgr.beginTransaction();

     addItems(xaction, Feed.getFeeds().get(itemPosition));
     removeFragments(fragMgr, xaction);
     xaction.commit();

     return(true);
   }
 }
```

In our case, we create a `FragmentTransaction`, use `addItems()` and `removeFragments()` to switch around our existing fragment mix as needed, and `commit()` the transaction.

Dialog Fragments

Traditional fragments go in the layout of the activity, either statically (`<fragment>` elements) or dynamically (`FragmentTransaction`).

A `DialogFragment`, as you might expect, is designed to display something over top of the layout of the activity, operating as a modal UI, taking over foreground input.

There are two basic approaches for implementing your own subclass of `DialogFragment`:

- Override `onCreateView()` and provide the contents to go in a plain `Dialog` UI

- Override `onCreateDialog()` and return your own instance of a `Dialog`, such as the results of using `AlertDialog.Builder`

The key is that you are not the one showing the dialog – that is managed by the system, much along the lines of "managed dialogs" in Android 1.x and 2.x.

One advantage of the former approach is that you can implement a fragment that works either as a dialog or as a regular fragment. Calling `setShowAsDialog()` with a `false` parameter would mean that the results of `onCreateView()` should be shown in the `DialogFragment` object's container.

In this case, `FeedFragments` uses the latter approach, overriding `onCreateDialog()`.

Each of our `AbstractFeedActivity` subclasses offers an "Add a Feed" options menu, whether as a toolbar button in the action bar or as a classic options menu item. That options menu item (`R.id.add`) is processed by the `onOptionsItemSelected()` method in `AbstractFeedsActivity`:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case R.id.add:
      new AddFeedDialogFragment()
        .show(getSupportFragmentManager(), "add_feed");

      return(true);

    case R.id.lights_out:
      lightsOut();

      return(true);
  }

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

Here, we create a new `AddFeedDialogFragment` and show it. You can either show it as part of a `FragmentTransaction` or in its own transaction

---

automatically created by a `FragmentManager` – in this case, we opt for the latter. The second parameter is the tag that will be associated with this fragment, for later retrieval via `findFragmentByTag()` on `FragmentManager`.

The `AddFeedDialogFragment` itself is fairly simple:

```
package com.commonsware.android.feedfrags;

import android.app.AlertDialog;
import android.app.Dialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.support.v4.app.DialogFragment;
import android.view.View;
import android.widget.EditText;

public class AddFeedDialogFragment extends DialogFragment {
  @Override
  public Dialog onCreateDialog(Bundle savedInstanceState) {
    final View form=getActivity()
                      .getLayoutInflater()
                      .inflate(R.layout.add_feed, null);

    return(new AlertDialog.Builder(getActivity())
            .setTitle(R.string.add_feed)
            .setView(form)
            .setPositiveButton(R.string.add,
              new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int whichButton) {
                  processAdd(form);
                }
              }
            )
            .setNegativeButton(R.string.cancel, null)
            .create());
  }

  private void processAdd(View form) {
    EditText name=(EditText)form.findViewById(R.id.name);
    EditText url=(EditText)form.findViewById(R.id.url);
    Feed feed=Feed.addFeed(name.getText().toString(),
                      url.getText().toString());

    ((AbstractFeedsActivity)getActivity()).addNewFeed(feed);
  }
}
```

In `onCreateDialog()`, we inflate our layout for the dialog contents (`res/layout/add_feed.xml`):

```xml
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
    >
  <TableRow>
    <TextView
      android:text="@string/name"
      android:layout_marginRight="4dip"
    />
    <EditText
      android:id="@+id/name"
      android:inputType="text|textCapWords"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="@string/url"
      android:layout_marginRight="4dip"
    />
    <EditText
      android:id="@+id/url"
      android:inputType="textUri"
    />
  </TableRow>
</TableLayout>
```

Then, we use `AlertDialog.Builder` to define the dialog itself, complete with Add and Cancel buttons. Note that we do not actually show the dialog with a call to `show()`, but rather simply create the `AlertDialog` with a call to `create()`. It is this `AlertDialog` that is returned from `onCreateDialog()`.

If the user presses the Add button, that invokes a `processAdd()` method which gets the values out of the `EditText` widgets, add a `Feed` object to the data model, and calls an `addNewFeed()` method on `AbstractFeedsActivity`. Another approach would be to have the activity register a listener, as we did with click events from our two `ListFragment` implementations.

The `addNewFeed()` method is abstract on `AbstractFeedsActivity`. The implementation in `FeedsActivity` delegates the event to the `FeedsFragment`, which simply adds the new `Feed` to its `ArrayAdapter`. The implementation in `FeedsTabActivity` is the `addNewFeed()` method we saw in an earlier section, which adds another tab to the action bar. Finally, `FeedsNavActivity` behaves

like `FeedsFragment`, adding the new `Feed` to its `ArrayAdapter` for the action bar navigation `Spinner`.

"Lights Out"

In Android 1.x and 2.x, you could go into "full screen mode" via a theme (`android:theme="@android:style/Theme.NoTitleBar.Fullscreen"`). This would hide the title bar and the status bar, giving you access to everything.

That is no longer strictly possible. The reason: the system bar needs to remain visible at all times, because it is where the HOME and BACK buttons reside.

What you can do is:

- Hide the action bar
- Dim the system bar, in so-called "lights out" mode, where the icons for buttons like HOME and BACK are replaced by small dots, to be less visually distracting

For example, here is the `lightsOut()` method from `AbstractFeedsActivity` that is called whenever the "Lights Out" menu item is chosen from any of the activities:

```
private void lightsOut() {
  final View view=findViewById(android.R.id.content);

  view.setSystemUiVisibility(View.STATUS_BAR_HIDDEN);
  getActionBar().hide();

  view.postDelayed(new Runnable() {
    public void run() {
      view.setSystemUiVisibility(View.STATUS_BAR_VISIBLE);
      getActionBar().show();
    }
  }, 5000);
}
```

We call `setSystemUiVisibility()` – available on any `View` – to "hide" the status bar, which on API Level 11 and higher will simply dim it. We also call

hide() on the ActionBar. Finally, we queue up a Runnable to be executed in five seconds to reverse our actions.

Now, you may be wondering why it is safe to have this code here. The getActionBar() and setSystemUiVisibility() methods were only introduced in API Level 11, so you might think that this will crash when the AbstractFeedsActivity class is loaded on older versions of Android.

Certainly, the author was wondering why this was working, upon realizing the potential problem.

By Android 2.1 (API Level 7), the Dalvik VM became more lenient about loading in classes that might refer to non-existent methods. So long as the lightsOut() method is not invoked, we are safe. However, on Android 1.6, the application will crash when started, because AbstractFeedsActivity cannot be loaded – we get a VerifyError.

Tactically, this project sets the minSdkVersion to be 7, to restrict us to devices on which this code will work. If you really wanted to support back to Android 1.6, you could move lightsOut() to a static method on a separate class (e.g., HoneycombHelper). Then, so long as we do not attempt to reference HoneycombHelper except on API Level 11, we are safe. This technique is also demonstrated elsewhere in this book, as well as in *The Busy Coder's Guide to Android Development*.

Leveraging the Home Icon

By default, the icon in the upper left corner of the action bar will be whatever icon you specify for your activity in the manifest (via the android:icon attribute), or the icon for your application if your activity does not supply one. However, you have some options:

- You can specify a separate drawable called the logo on your <application> element via the android:logo attribute. You can then switch to using this drawable in the action bar via a call to setDisplayUseLogoEnabled() on your ActionBar.

- You can disable the icon entirely via a call to `setDisplayShowHomeEnabled()` on your `ActionBar`. Similarly, you can get rid of the title via `setDisplayShowTitleEnabled()`.

- If the icon, when tapped, will lead the user "up" your navigation tree, you can add a northwest-pointing arrowhead via a call to `setDisplayHomeAsUpEnabled()` on the `ActionBar`. Here, "up" typically means a parent activity, if this activity is a "drill-down" into some subset of your application's functionality.

As an example of the latter, here again is `onCreate()` from `FeedsNavActivity`:

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

   ActionBar bar=getActionBar();

   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
     adapter=
         new ArrayAdapter<Feed>(bar.getThemedContext(), R.layout.row,
                                Feed.getFeeds());
   }
   else {
     adapter=
         new ArrayAdapter<Feed>(this, R.layout.row, Feed.getFeeds());
   }

   bar.setListNavigationCallbacks(adapter, new NavListener());
   bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
   bar.setDisplayOptions(ActionBar.DISPLAY_SHOW_TITLE,
                         ActionBar.DISPLAY_SHOW_TITLE);
   bar.setDisplayHomeAsUpEnabled(true);
 }
```

The `setDisplayHomeAsUpEnabled()` call is what adds the arrowhead to the icon.

When the user taps this icon, your `onOptionsItemSelected()` method will be called, with a `MenuItem` whose ID is `android.R.id.home`. If you chose to add the arrowhead, then your mission – should you choose to accept it – will be to send the user to the appropriate activity. This might be via a call to `startActivity()` on an `Intent` identifying the right activity and using `FLAG_ACTIVITY_REORDER_TO_FRONT` to bring the existing instance back to the

foreground. In the case of `FeedsNavActivity` and `FeedsTabActivity`, they simply call `finish()`, since the "up" and the "back" activity both happen to be the same thing:

```
@Override
 public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
     case android.R.id.home:
       finish();

       return(true);
   }

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

And All The Rest

Most of the `FeedFragments` classes were covered in the preceding sections. In this section, we will quickly examine the remainder, which provide needed functionality for the application but do not really demonstrate much in the way of new fragment/action bar techniques.

PersistentListFragment

The way we get the blue background on the list items that were tapped is via a combination of the `ListView` in the `ListFragment` being `CHOICE_MODE_SINGLE` and using an "activated" style. `PersistentListFragment` wraps up:

- Toggling the `ListView` into `CHOICE_MODE_SINGLE` when needed (i.e., we are in a multi-fragment feeds activity, as opposed only having one fragment per activity)
- Setting the clicked-upon list item as being checked
- Persisting the checked state via `onSaveInstanceState()`, so the state can be restored (by a `restoreState()` method) after a configuration change, etc.

Here is the complete implementation of `PersistentListFragment`:

```
package com.commonsware.android.feedfrags;

import android.os.Bundle;
import android.support.v4.app.ListFragment;
import android.view.View;
import android.widget.ListView;

public class PersistentListFragment extends ListFragment {
  static public final String
STATE_CHECKED="com.commonsware.android.feedfrags.STATE_CHECKED";

  @Override
  public void onListItemClick(ListView l, View v, int position,
                              long id) {
    l.setItemChecked(position, true);
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    state.putInt(STATE_CHECKED,
                 getListView().getCheckedItemPosition());
  }

  protected void restoreState(Bundle state) {
    if (state!=null) {
      int position=state.getInt(STATE_CHECKED, -1);

      if (position>-1) {
        getListView().setItemChecked(position, true);
      }
    }
  }

  public void enablePersistentSelection() {
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
  }
}
```

FeedsActivity calls `enablePersistentSelection()` from `onCreate()`, if and only if it is hosting multiple fragments:

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

  FeedsFragment feeds
    =(FeedsFragment)getSupportFragmentManager()
                       .findFragmentById(R.id.feeds);

  feeds.setOnFeedListener(this);
```

```
  isThreePane=(null!=findViewById(R.id.second_pane));

  if (isThreePane) {
    feeds.enablePersistentSelection();
  }
}
```

FeedsFragment and ItemsFragment inherit from PersistentListFragment, so they adopt these behaviors automatically. They also need to use a row layout that supports the "activated" style:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/text1"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:textAppearance="?android:attr/textAppearanceLarge"
  android:gravity="center_vertical"
  android:layout_marginLeft="4dip"
  android:minHeight="?android:attr/listPreferredItemHeight"
  style="@style/activated"
/>
```

The activated style is defined separately for API Level 11 (res/values-v11/styles.xml):

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?
android:attr/activatedBackgroundIndicator</item>
  </style>
</resources>
```

...and for older versions (res/values/styles.xml):

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated">
  </style>
</resources>
```

FeedsFragment and ItemsFragment also need to call restoreState(). In theory, you would do this from a fragment lifecycle method like onActivityCreated(). In fact, that is precisely how this is handled in

ItemsFragment, since that fragment is retained and so its ListView contents should remain intact:

```
public void onActivityCreated(Bundle state) {
   super.onActivityCreated(state);

   registerForContextMenu(getListView());
   restoreState(state);

   if (persistentSelection) {
     enablePersistentSelection();
   }
 }
```

However, FeedsFragment is not loading the contents of the feeds into the ListView until onResume(), and so we cannot set the checked row item until that has been accomplished. Hence, we need to cache the instance state Bundle from onActivityCreated() and apply it in onResume():

```
private Bundle state=null;

@Override
public void onActivityCreated(Bundle state) {
  super.onActivityCreated(state);

  this.state=state;
}

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

  loadFeeds();
  restoreState(state);
}
```

ContentFragment

For displaying the feed's Web content in a FragmentActivity, we need a Fragment. Ideally, we would use a WebViewFragment, but that is not available in the Android Support package, so we need to create our own, named ContentFragment:

```
package com.commonsware.android.feedfrags;

import android.os.Bundle;
```

```
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.support.v4.app.Fragment;

public class ContentFragment extends Fragment {
  private String urlToLoad=null;

  public ContentFragment() {
    this(null);
  }

  public ContentFragment(String url) {
    super();

    urlToLoad=url;
    setRetainInstance(true);
  }

  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
    return(inflater.inflate(R.layout.content_fragment, container, false));
  }

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

    WebView browser=getBrowser();

    browser.setWebViewClient(new RedirectFixer());
    browser.getSettings().setJavaScriptEnabled(true);

    if (savedInstanceState!=null) {
      browser.restoreState(savedInstanceState);
    }
    else if (urlToLoad!=null) {
      loadUrl(urlToLoad);
    }
  }

  @Override
  public void onSaveInstanceState(Bundle state) {
    super.onSaveInstanceState(state);

    WebView browser=getBrowser();

    if (browser!=null) {
      browser.saveState(state);
    }
  }
```

179

```
public void loadUrl(String url) {
  getBrowser().loadUrl(url);
}

private WebView getBrowser() {
  return((WebView)(getView().findViewById(R.id.browser)));
}

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

    return(true);
  }
}
}
```

The only UI is a WebView, inflated in onCreateView() from a content_fragment layout resource:

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

There are a few curiosities of note in ContentFragment:

- Obviously, we need to tell the fragment what URL to load into the WebView. We do that in two places: a constructor parameter (for a newly-created fragment) and a loadUrl() method (to re-populate an existing fragment with new content). The constructor parameter cannot be applied immediately, as we do not have a WebView yet. And, we also have to deal with the case of a rotation or other configuration change. So, onSaveInstanceState() persists the WebView widget's state, which is restored in onActivityCreated(). If, in onActivityCreated(), we have no state but we do have a URL from the constructor, we apply it.

- The onActivityCreated() method is also where we configure the inflated WebView, enabling JavaScript and connecting it to a RedirectFixer implementation of WebViewClient, so redirects (and,

unfortunately, link clicks) route back to the `WebView` instead of popping up a full browser activity.

ItemsActivity

If `FeedFragments` runs on a phone, we are not loading all the fragments into one activity. Rather, `FeedsActivity` will display the `FeedsFragment`, and an `ItemsActivity` will display the `ItemsFragment`:

```java
package com.commonsware.android.feedfrags;

import org.mcsoxford.rss.RSSItem;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class ItemsActivity extends FragmentActivity
    implements ItemsFragment.OnItemListener {
  public static final String EXTRA_FEED_KEY=
    "com.commonsware.android.feedfrags.EXTRA_FEED_KEY";
  ItemsFragment items=null;

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

    items=(ItemsFragment)getSupportFragmentManager()
                        .findFragmentById(R.id.items);

    String key=getIntent().getStringExtra(EXTRA_FEED_KEY);

    if (key!=null) {
      Feed feed=Feed.getFeed(key);

      setTitle(String.format(getString(R.string.feed_title),
                        feed.toString()));
      items.loadUrl(feed.getUrl());
    }
  }

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

    items.setOnItemListener(this);
  }

  public void onItemSelected(RSSItem item) {
    startActivity(new Intent(Intent.ACTION_VIEW, item.getLink()));
```

```
  }
}
```

This loads an `ItemsFragment`-only layout resource named `items`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  class="com.commonsware.android.feedfrags.ItemsFragment"
  android:id="@+id/items"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

In `ItemsActivity`, we:

- Get the inflated `ItemsFragment` and register the `ItemsActivity` as the `OnItemListener`

- If we were supplied a feed's key via an `Intent` extra, we get the corresponding `Feed` object and use that to customize our title bar plus load the items into the fragment

- In `onItemSelected()`, we fire up the default browser on whatever URL is associated with the clicked-upon feed item

- In `onResume()`, we tell the `ItemsFragment` to notify us via `onItemSelected()` via a call to `setOnItemListener()` – this will handle both when the activity is newly created and when it is destroyed and recreated (but the `ItemsFragment` is retained)

Other Bits of Goodness

There are many more things that you can accomplish using fragments and the action bar. Besides the main techniques outlined in the rest of this chapter, this section outlines a few other features that you may find useful.

Custom Navigation Mode

`FeedFragments` demonstrated "list" and tabs as the two built-in navigation modes implemented in the action bar via fragments. However, you can roll your own. For example, a Web browser might put the browser bar (for

URLs, searching, etc.) in the action bar, while an instant messenger might go with an `AutoCompleteTextView` of contacts.

To do this, call `setCustomView()` on the `ActionBar` to supply a layout resource ID or a `View` to be added to the action bar. Also, call `setOptions()` on the `ActionBar`, OR-ing in (via the | operator) the `DISPLAY_SHOW_CUSTOM` flag. This combination should put your desired view between the title on the left and the toolbar buttons on the right.

It is up to you to attach listeners or whatever to respond to user input on that custom `View`. You can retrieve the inflated layout via `getCustomView()` for this purpose.

Dynamic Menus

One thing that may not be obvious when implementing the Honeycomb UI is that `onCreateOptionsMenu()` and `onPrepareOptionsMenu()` – normally only called when the MENU button is pressed – get called right away. If you think about it, this makes sense. The action bar contents are largely populated by the options menu system, and so Android needs to invoke those methods to define the options menu before presenting the action bar to the user.

This may cause some additional complexity for you, however.

First, `onPrepareOptionsMenu()` is called once up front in addition to being called on every MENU press. Hopefully, your implementation of this method will work in either case. Otherwise, you may have to manage your own flag to indicate whether a particular `onPrepareOptionsMenu()` call is the first time (to define the action bar) or later (on a MENU press).

Also, if you decide that you need to more substantively modify the menu – beyond what you would ordinarily handle in `onPrepareOptionsMenu()` – there is a new `invalidateOptionsMenu()` method in API Level 11 on `Activity` that forces the menu to be recreated from scratch on the next invocation.

More On Retained Instances

While much of the focus on fragments is tied to their UI presentation, the concept of retained instances has its own merits. Marking fragments as being retained can simplify configuration changes, such as the ever-popular screen rotation. Rather than having to mess around with `onSaveInstanceState()` or `onRetainNonConfigurationInstance()`, you have your retained fragments manage the state, and those fragments (and their state) will be retained across configuration changes.

For example, in *The Busy Coder's Guide to Android Development*, there are a couple of samples of using the US National Weather Service's Web service to find out the weather for a given location, determined by GPS. The original sample was fairly simple but performed its network I/O on the main application thread, which is not a good idea. The second sample demonstrated having a bound service manage the Web service call on a background thread... but this sample mostly served to illustrate that bound services are a serious pain with respect to configuration changes.

In the `Honeycomb/WeatherFragment` sample project in this book's source code is a third rendition of the same sample application, this time using retained fragments. This section will not discuss the business logic – please refer to *The Busy Coder's Guide to Android Development* for that. Instead, we will focus on how a retained fragment helps simplify configuration changes.

The Activity

In the original examples, the activity had the `WebView` that served as the UI, with the forecast being converted from the Web service's XML payload into HTML. Similarly, the activity was responsible for binding to the service, passing it the location obtained from GPS, and eventually unbinding from that service.

Now, all of that logic is managed by the fragment.

You might think that the activity would have nothing more than an onCreate() method loading some layout file in setContentView() that had a <fragment> element. This does not work – retained instances must be dynamic. Any fragment declared by <fragment> will be destroyed and recreated regardless of any other setting.

So, instead, the onCreate() of WeatherDemo looks like this:

```java
package com.commonsware.android.weather;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;

public class WeatherDemo extends FragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if (savedInstanceState == null) {
        getSupportFragmentManager()
          .beginTransaction()
          .add(android.R.id.content, new WeatherFragment())
          .commit();
    }
  }
}
```

Here, we check to see if savedInstanceState is null. If so, then we are being created anew, and therefore we need to create our fragment and apply it via a FragmentTransaction. We place the fragment in the container identified by android.R.id.content, which is the same container that the results of setContentView() are poured into, supplied by the activity itself.

If the savedInstanceState parameter is not null, though, then our activity was recreated as a result of some configuration change or similar event. In that case, Android is responsible for handling the fragments from the previous activity instance, so we can skip initializing another one.

The Fragment

The fragment is the one responsible for binding to the service and eventually unbinding as well, plus finding our location via GPS. However,

we need to ensure that the `ServiceConnection` object that is used to represent the binding survives configuration changes. To that end, `WeatherFragment` calls `setRetainInstance(true)` in its `onCreate()` method:

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

    setRetainInstance(true);

    getActivity()
      .getApplicationContext()
      .bindService(new Intent(getActivity(), WeatherService.class),
                 this, Context.BIND_AUTO_CREATE);

    mgr=(LocationManager)getActivity().getSystemService(Context.LOCATION_SERVICE
);
    mgr.requestLocationUpdates(LocationManager.GPS_PROVIDER,
                              3600000, 1000, this);
  }
```

With that one statement, we can simply hold onto the `ServiceConnection` in an ordinary data member of the fragment and not have to worry about configuration changes. And, we can cleanly unbind in `onDestroy()`, when we know our service is no longer needed:

```
@Override
public void onDestroy() {
  if (mgr!=null) {
    mgr.removeUpdates(this);
  }

  getActivity().getApplicationContext().unbindService(this);
  super.onDestroy();
}
```

Hence, that one call to `setRetainInstance()` replaces all of the `onRetainNonConfigurationInstance()` shenanigans we had to deal with in the service example from *The Busy Coder's Guide to Android Development*.

What We Really Should Be Doing

In truth, none of these samples is the best solution to the problem. While keeping the network I/O off the main application thread is important,

using a bound service to accomplish that end is not necessary and adds complexity.

The best solution, in all likelihood, is to use an `AsyncTask` for the network I/O, but have it be spawned by a retained fragment rather than by a service. This way, the task can communicate with the fragment in `onPostExecute()` to supply the results, and it will always talk to the right fragment. This eliminates the headache of binding and unbinding from the service.

Action Modes

If you have spent much time on an Android tablet (Honeycomb or higher), you probably have run into a curious phenomenon. Sometimes, when you select an item in a list or other widget, the action bar magically transforms from its normal look:

Figure 49. The Gmail action bar on a Honeycomb tablet, in normal mode

to one designed to perform operations on what you have selected:

Figure 50. The Gmail action bar on a Honeycomb tablet, showing an action mode

The good news is that this is not some sort of magic limited only to firmware applications like Gmail. You too can have this effect in your application, by triggering an "action mode".

Saying Goodbye to Context Menus

Context menus are certainly useful. Power users can save screen taps if they know where context menus reside and what features they offer. For example, rather than tapping on a list item, then opening an options menu, then tapping a menu item to delete something, a power user could long-tap to open a context menu, then tap on a context menu item – saving one tap and switching back and forth between activities.

The problem is that context menus are invisible and are triggered by an action not used elsewhere very much (long tap).

In theory, users would find out about context menus in your application from reading your documentation. That would imply that we were in some alternate universe where all users read documentation, all people live in peach and harmony, and all book authors have great heads of hair. In this universe, power users will find your context menus, but ordinary users may be completely oblivious to them.

The action bar itself is designed to help raise the visibility of the options menu (e.g., turning menu items into toolbar buttons) and standardizing the location of navigation elements (e.g., tabs, search fields). The action bar takes advantage of the fact that we have a lot more screen space on a tablet than we do on a phone, and uses some of that space to consistently benefit the user.

The action mode is designed to perform a similar bit of magic for context menus. Rather than have context menus be buried under a long-tap, action modes let the contextual actions take over the action bar, putting them front-and-center in the user experience.

Manual Action Modes

A common pattern will be to activate an action mode when the user checks off something in a multiple-choice `ListView`, as is the case with applications

like Gmail. If you want to go that route, there is some built-in scaffolding to make that work, described later in this chapter.

You can, if you wish, move the action bar into an action mode whenever you want. This would be particularly important if your UI is not based on a `ListView`. For example, tapping on an image in a `GridView` might activate it and move you into an action mode for operations upon that particular image.

In this section, we will examine the `Honeycomb/ActionMode` sample project. This is based on the `Menus/ActionBarBC` sample project from *The Busy Coder's Guide to Android Development*, augmented to support action modes on Honeycomb and higher, yet still remain backwards-compatible to Android 2.x devices.

Choosing Your Trigger

As noted above, Gmail switches into an action mode when the user checks off one or more conversations in the conversations list. Selecting a word or passage in an `EditText` (e.g., via a long-tap) brings up an action mode for cut/copy/paste operations. And so on.

You will need to choose, for your own UI, what the trigger mechanism will bring up an action mode. It should be some trigger that makes it obvious to the user what the action mode will be acting upon. For example:

- If the user taps on the current selected item in a `Gallery` widget, bring up an action mode for operations on that particular item

- If the user long-taps on an item in a `GridView`, bring up an action mode, and treat future taps on `GridView` items as adding or removing items from the "selection" while that action mode is visible

- If the user "rubber-bands" some figures in your vector art drawing `View`, bring up an action mode for operations on those figures (e.g., rotate, resize)

In the case of the `ActionMode` sample project, we stick with the classic long-tap on a `ListView` row to bring up an action mode that replaces the context menu when run on a API Level 11+ device. However, we still want to use a context menu on pre-Honeycomb devices, so we conditionally set up on our long-click listener based on Android API level:

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

  initAdapter();
  if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
    getListView().setLongClickable(true);
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    getListView()
      .setOnItemLongClickListener(new ActionModeHelper(this,
                                              getListView()));
  }
  else {
    registerForContextMenu(getListView());
  }
}
```

Starting the Action Mode

Starting an action mode is trivially easy: just call `startActionMode()` on your `Activity`, passing in an implementation of `ActionMode.Callback`, which will be called with various lifecycle methods for the action mode itself.

In the case of the `ActionMode` sample project, `ActionModeHelper` – our `OnItemLongClickListener` from the preceding section – also is our `ActionMode.Callback` implementation. Hence, when the user long-clicks on an item in the `ListView`, the `ActionModeHelper` establishes itself as the action mode:

```java
@Override
public boolean onItemLongClick(AdapterView<?> view, View row,
                        int position, long id) {
  lastPosition=position;
  modeView.clearChoices();
  modeView.setItemChecked(lastPosition, true);

  if (activeMode==null) {
    activeMode=host.startActionMode(this);
  }
```

```
   return(true);
 }
```

Note that `startActionMode()` returns an `ActionMode` object, which we can use later on to configure the mode's behavior.

We also do a couple of other things here:

- We record which position the user long-tapped upon, as we will need this information eventually when it comes time to perform actions on this particular list item

- We make that position be "checked", to show which item the action mode will act upon (given our row layout, this will show up with the "activated" style)

- We only start the action mode if it is not already started

Implementing the Action Mode

The real logic behind the action mode lies in your `ActionMode.Callback` implementation. It is in these four lifecycle methods where you define what the action mode should look like and what should happen when choices are made in it.

onCreateActionMode()

The `onCreateActionMode()` method will be called shortly after you call `startActionMode()`. Here, you get to define what goes in the action mode. You get the `ActionMode` object itself (in case you do not already have a reference to it). More importantly, you are passed a `Menu` object, just as you get in `onCreateOptionsMenu()`. And, just like with `onCreateOptionsMenu()`, you can inflate a menu resource into the `Menu` object to define the contents of the action mode:

```
@Override
 public boolean onCreateActionMode(ActionMode mode, Menu menu) {
   MenuInflater inflater=host.getMenuInflater();
```

```
    inflater.inflate(R.menu.context, menu);
    mode.setTitle(R.string.context_title);

    return(true);
  }
```

In addition to inflating our context menu resource into the action mode's menu, we also set the title of the ActionMode, which shows up to the right of the Done button:

Figure 51. The ActionMode sample application's action bar on a Honeycomb tablet, showing the active action mode

onPrepareActionMode()

If you determine that you need to change the contents of your action mode, you can call invalidate() on the ActionMode object. That, in turn, will trigger a call to onPrepareActionMode(), where you once again have an opportunity to configure the Menu object. If you do make changes, return true – otherwise, return false. In the case of ActionModeHelper, we take the latter approach:

```
@Override
public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
  return(false);
}
```

onActionItemClicked()

Just as onCreateActionMode() is the action mode analogue to onCreateOptionsMenu(), onActionItemClicked() is the action mode analogue to onOptionsItemSelected(). This will be called if the user clicks on something related to your action mode. You are passed in the corresponding MenuItem object (plus the ActionMode itself), and you can take whatever steps are necessary to do whatever the work is.

In the case of the ActionMode sample application, we want the same work to be done whether the user clicks on an action mode item on an API Level 11+

device or chooses a context menu items on a pre-Honeycomb device. So, in the ActionModeDemo class, we have the business logic for both of these cases in a performAction() method:

```
@SuppressWarnings("unchecked")
 public boolean performAction(MenuItem item, int position) {
   ArrayAdapter<String> adapter=(ArrayAdapter<String>)getListAdapter();

   switch (item.getItemId()) {
     case R.id.cap:
       String word=words.get(position);

       word=word.toUpperCase();

       adapter.remove(words.get(position));
       adapter.insert(word, position);

       return(true);

     case R.id.remove:
       adapter.remove(words.get(position));

       return(true);
   }

   return(false);
 }
```

The activity's own onContextItemSelected() routes to performAction():

```
@Override
 public boolean onContextItemSelected(MenuItem item) {
   AdapterView.AdapterContextMenuInfo info=
       (AdapterView.AdapterContextMenuInfo)item.getMenuInfo();

   boolean result=performAction(item, info.position);

   if (!result) {
     result=super.onContextItemSelected(item);
   }

   return(result);
 }
```

Also, the onActionItemClicked() method calls performAction():

```
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
  boolean result=host.performAction(item, lastPosition);
```

```
  if (item.getItemId()==R.id.remove) {
    activeMode.finish();
  }

  return(result);
}
```

onActionItemClicked() also dismisses the action mode if the user chose the "remove" item, since the action mode is no longer needed. You get rid of an active action mode by calling finish() on it.

onDestroyActionMode()

The onDestroyActionMode() callback will be invoked when the action mode goes away, for any reason, such as:

- The user clicks the Done button on the left

- The user clicks the BACK button

- You call finish() on the ActionMode

Here, you can do any necessary cleanup. ActionModeHelper tries to clean things up, notably the "checked" state of the last item long-tapped-upon:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  activeMode=null;
  modeView.clearChoices();
}
```

However, for reasons that are not yet clear, clearChoices() does not update the UI when called from onDestroyActionMode().

Multiple-Modal-Choice Action Modes

For many cases, the best user experience will be for you to have a multiple-choice ListView, where checking items in that list enables an action mode for performing operations on the checked items. For this scenario, Android has a new built-in ListView choice mode, CHOICE_MODE_MULTIPLE_MODAL, that

automatically sets up an ActionMode for you as the user checks and unchecks items.

To see how this works, let's examine the Honeycomb/ActionModeMC project. This is the same project as in the preceding section, but altered to have a multiple-choice ListView, utilizing an action mode on Honeycomb.

Once again, in onCreate(), we need to set up the smarts for our ListView. This time, though, we will use CHOICE_MODE_MULTIPLE_MODAL:

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

    initAdapter();
    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
      getListView()
        .setMultiChoiceModeListener(new HCMultiChoiceModeListener(this,
                                                      getListView()));
    }
    else {
      getListView().setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
      registerForContextMenu(getListView());
    }
  }
```

If we are on an API Level 11+ device, we enable CHOICE_MODE_MULTIPLE_MODAL for the ListView, and register an instance of an HCMultiChoiceModeListener object via setMultiChoiceModeListener(). This object is an implementation of the MultiChoiceModeListener interface that we will examine shortly.

If we are not on a Honeycomb device, we set up a regular CHOICE_MODE_MULTIPLE ListView, with a registered context menu. Since we now may have multiple checked items, our performAction() method must take this into account, capitalizing or removing all checked words:

```
@SuppressWarnings("unchecked")
 public boolean performActions(MenuItem item) {
    ArrayAdapter<String> adapter=(ArrayAdapter<String>)getListAdapter();
    SparseBooleanArray checked=getListView().getCheckedItemPositions();

    switch (item.getItemId()) {
      case R.id.cap:
```

```
        for (int i=0;i<checked.size();i++) {
          if (checked.valueAt(i)) {
            int position=checked.keyAt(i);
            String word=words.get(position);

            word=word.toUpperCase();

            adapter.remove(words.get(position));
            adapter.insert(word, position);
          }
        }

        return(true);

      case R.id.remove:
        ArrayList<Integer> positions=new ArrayList<Integer>();

        for (int i=0;i<checked.size();i++) {
          if (checked.valueAt(i)) {
            positions.add(checked.keyAt(i));
          }
        }

        Collections.sort(positions, Collections.reverseOrder());

        for (int position : positions) {
          adapter.remove(words.get(position));
        }

        getListView().clearChoices();

        return(true);
    }

    return(false);
  }
```

Back in the Honeycomb-or-higher code, `MultiChoiceModeListener` extends the `ActionMode.Callback` interface we used with our manual action mode earlier in this book. Hence, we need to implement all the standard `ActionMode.Callback` methods, plus a new `onItemCheckedStateChanged()` method introduced by `MultiChoiceModeListener`:

```
package com.commonsware.android.actionmodemc;

import android.view.ActionMode;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.AbsListView;
import android.widget.ListView;
```

```
public class HCMultiChoiceModeListener implements
AbsListView.MultiChoiceModeListener {
  ActionModeDemo host;
  ActionMode activeMode;
  ListView lv;

  HCMultiChoiceModeListener(ActionModeDemo host, ListView lv) {
    this.host=host;
    this.lv=lv;
  }

  @Override
  public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    MenuInflater inflater=host.getMenuInflater();

    inflater.inflate(R.menu.context, menu);
    mode.setTitle(R.string.context_title);
    mode.setSubtitle("(1)");
    activeMode=mode;

    return(true);
  }

  @Override
  public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
    return(false);
  }

  @Override
  public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    boolean result=host.performActions(item);

    updateSubtitle(activeMode);

    return(result);
  }

  @Override
  public void onDestroyActionMode(ActionMode mode) {
    activeMode=null;
  }

  @Override
  public void onItemCheckedStateChanged(ActionMode mode, int position,
                                        long id, boolean checked) {
    updateSubtitle(mode);
  }

  private void updateSubtitle(ActionMode mode) {
    mode.setSubtitle("("+lv.getCheckedItemCount()+")");
  }
}
```

Android will automatically start our action mode for us when the user checks the first item in the list, using our `MultiChoiceModeListener` as the callback. Android will also automatically finish the action mode if the user unchecks all previously-checked items.

In `onCreateActionMode()`, we populate the menu, plus set up a title and subtitle on the `ActionMode`. The subtitle appears below the title, as you might expect. In this case, we are indicating how many words are checked and therefore will be affected by the actions the user chooses in the action mode:
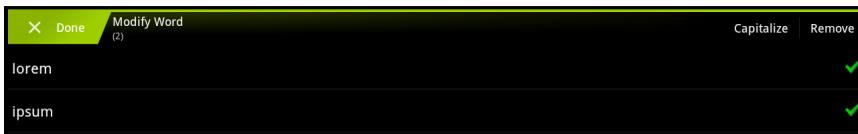
Figure 52. The ActionModeMC sample application's action bar on a Honeycomb tablet, showing the active action mode

Then, in `onActionItemClicked()`, we both call `performActions()` to affect the desired changes, plus update the subtitle in case the user removed words (which means they are no longer checked).

The new `onItemCheckedStateChanged()` will be called whenever the user checks or unchecks an item, up until the last item is unchecked. `HCMultiChoiceModeListener` simply updates the subtitle to reflect the new count of checked items.

On the whole, using `CHOICE_MODE_MULTIPLE_MODAL` is simpler than setting up your own trigger mechanism and managing the action mode yourself. That being said, both are completely valid options, which is particularly important for situations where a multiple-choice `ListView` is not the desired user interface.

PART II – Advanced Media

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.

203

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 `Views` 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,
                            getHeight(),
                            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,
                            getHeight());
```

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:
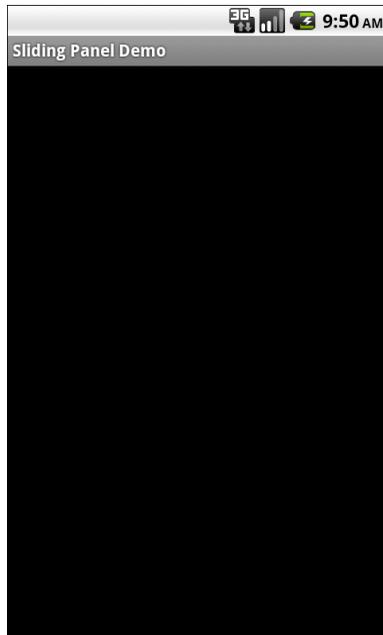
Figure 53. The SlidingPanel sample application, with the panel closed

...or open, in this case toggled via a menu choice in the SlidingPanelDemo activity:
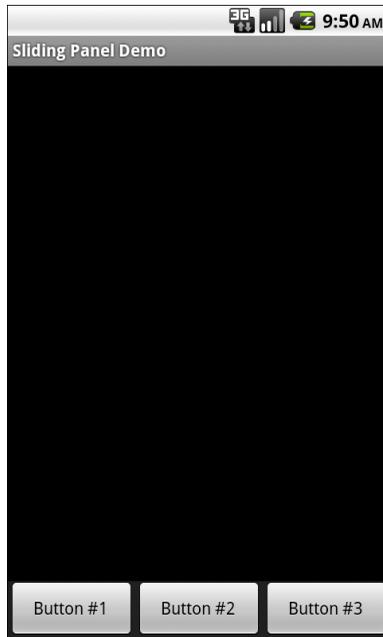
Figure 54. The SlidingPanel sample application, with the panel open

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
<?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, seen here in our SlidingPanel constructor:

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

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

  speed=a.getInt(R.styleable.SlidingPanel_speed, 300);

  a.recycle();

  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`:

```java
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,
                                getHeight(),
```

213

```
                              0.0f);
  }
  else {
    anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                getHeight());
    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

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

217

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 and the Feature

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
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.camera"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-sdk android:minSdkVersion="5"
            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:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:configChanges="keyboardHidden|orientation"
              android:label="@string/app_name"
              android:name=".PreviewDemo"
              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>
```

The manifest also contains a <uses-feature> element, declaring that we need a camera on the device. This will not block installation of this application on a camera-less device, but it will cause the Android Market to filter out this app when viewed on camera-less devices. Other markets could conceivably do the same thing.

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/Theme.NoTitleBar.Fullscreen"` to get rid of the title bar and status bar, so the preview is truly full-screen (e.g., 480x320 on an HVGA device).

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="match_parent"
    android:layout_height="match_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. Note that `setType()` is not needed on Android 3.0 and higher, as Android will auto-detect that you need this type of `SurfaceView` when you attach it to the `Camera`. It is safe to call `setType()` on all versions of Android, though, so as long as you are continuing to support Android 1.x and 2.x, continue to call `setType()`. For applications supporting API Level 11 and higher, you can skip `setType()`.

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);
}
```

We get our `Camera` in `onResume()`, using the original `Camera` `open()` method to retrieve the default camera for the device:

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

  camera=Camera.open();
}
```

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 `SurfaceView` 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) {
  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.

At this point, though, we have a problem. The SurfaceView may be of an arbitrary size, depending on the device. However, not all devices support arbitrary-sized previews. Hence, we need to do the following:

1. Get a Camera.Parameters object, by calling getParameters() on the Camera

2. Call getSupportedPreviewSizes() to determine what preview sizes are available

3. Choose one of those sizes...somehow...

In our case, the determination of which preview size to use is implemented in a getBestPreviewSize() method:

```java
private Camera.Size getBestPreviewSize(int width, int height,
                                       Camera.Parameters parameters) {
  Camera.Size result=null;

  for (Camera.Size size : parameters.getSupportedPreviewSizes()) {
    if (size.width<=width && size.height<=height) {
      if (result==null) {
        result=size;
      }
      else {
        int resultArea=result.width*result.height;
        int newArea=size.width*size.height;

        if (newArea>resultArea) {
          result=size;
        }
      }
    }
  }

  return(result);
}
```

Here, we use a fairly crude algorithm: we choose the largest preview size that is smaller than our actual `SurfaceView`, where "largest" is the one that has the largest area.

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()`:

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

  if (size!=null) {
    parameters.setPreviewSize(size.width, size.height);
    camera.setParameters(parameters);
    camera.startPreview();
    inPreview=true;
  }
}
```

Eventually, the preview needs to stop. More importantly, we need to let go of the `Camera` in `onPause()`, so while our activity is not in the foreground, we do not tie up the camera, preventing other applications from using it. So, we keep track of whether preview is turned on yet via the `inPreview` flag, and in `onPause()` we stop the preview (if needed), `release()` the `Camera`, set the data member to `null` (to incrementally increase the speed of garbage collection), and reset the `inPreview` flag to `false`:

```java
@Override
public void onPause() {
  if (inPreview) {
    camera.stopPreview();
  }

  camera.release();
  camera=null;
  inPreview=false;

  super.onPause();
}
```

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

Image Is Everything

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

In principle, this is easy. Where things get a bit complicated comes with ensuring the application (and device as a whole) has decent performance, not slowing down to process the pictures. Also, to make this even more fun, we will use the front-facing camera if one is available; otherwise, we will use the device's default camera.

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 Camera. Maybe.

We can include a second `<uses-feature>` element in our manifest, asking not only for a camera, but specifically for a front-facing camera:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.picture"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-sdk android:minSdkVersion="7" android:targetSdkVersion="11"/>
  <supports-screens android:largeScreens="false"
                    android:normalScreens="true"
                    android:smallScreens="false" android:xlargeScreens="true"/>
  <uses-feature android:name="android.hardware.camera" />
  <uses-feature android:name="android.hardware.camera.front"
                android:required="false" />
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

```
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:configChanges="keyboardHidden|orientation"
              android:label="@string/app_name"
              android:name=".PictureDemo"
              android:screenOrientation="landscape">

      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

However, since we do not absolutely need a front-facing camera, we use `android:required="false"`. Again, `<uses-feature>` is mostly a hint for the Android Market and related tools.

Getting the Camera

If we want to use the front-facing camera, we have three possible scenarios:

1. We are on an Android 2.3+ device that has a front-facing camera

2. We are on an Android 2.3+ device that does not have a front-facing camera

3. We are on a device running Android 2.2 or older, before Android had standardized ways to access the front-facing camera

While in principle we could use some manufacturer-specific techniques to get to the front-facing camera on Android 2.1 or 2.2, that is beyond the scope of this chapter, and should be a moot point by the end of 2011 as those older Android versions start to fade into the distance.

Since we are writing version-specific code, we need to take steps to make sure our Android 2.3 logic is not run on, say, an Android 2.2 device. We will cheat a bit and only support Android 2.0 and higher in this sample, which means we can have references to classes and methods from newer versions of Android in our code, so long as we only execute statements containing

those references on Android versions that actually have those classes and methods.

Hence, our `onResume()` method now looks like this:

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

   if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.GINGERBREAD) {
     Camera.CameraInfo info=new Camera.CameraInfo();

     for (int i=0;i<Camera.getNumberOfCameras();i++) {
       Camera.getCameraInfo(i, info);

       if (info.facing==Camera.CameraInfo.CAMERA_FACING_FRONT) {
         camera=Camera.open(i);
       }
     }
   }

   if (camera==null) {
     camera=Camera.open();
   }
 }
```

If we are on Gingerbread (API Level 9) or higher, then we iterate over all cameras on the device (`getNumberOfCameras()`), get the `Camera.CameraInfo` for each (`getCameraInfo()`), and see if the camera index in question points to a front-facing camera. If it does , we open that specific camera via the `open()` method that takes a camera index as a parameter. If we make it through the full list without finding a front-facing camera, we shrug our virtual shoulders and default to the default camera via the zero-parameter `open()`.

Asking for a Format

We need to tell the `Camera` what sort of picture to take when we decide to take a picture. 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 want a simple JPEG image:

```
   public void surfaceChanged(SurfaceHolder holder,
       int format, int width, int height) {
```

```
    Camera.Parameters parameters=camera.getParameters();
    Camera.Size size=getBestPreviewSize(width, height,
        parameters);

    if (size!=null) {
      parameters.setPreviewSize(size.width, size.height);
      parameters.setPictureFormat(PixelFormat.JPEG);

      camera.setParameters(parameters);
      camera.startPreview();
      inPreview=true;
    }
  }
```

Taking a Picture

Somehow, your application will need to indicate when a picture should be taken. In our case, we will use a simple options menu. On devices with an action bar, we will have a toolbar button to take a picture:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:title="Take Picture" android:id="@+id/camera"
android:icon="@drawable/ic_menu_camera" android:showAsAction="ifRoom"></item>
</menu>
```

On other devices, the user can open up the options menu using the MENU button as normal.

Of course, we need to set up the menu and handle menu item choices/toolbar button clicks:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  new MenuInflater(this).inflate(R.menu.options, menu);

  return(super.onCreateOptionsMenu(menu));
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId()==R.id.camera) {
    if (inPreview) {
      camera.takePicture(null, null, photoCallback);
      inPreview=false;
    }
  }
```

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

When 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 we requested JPEG output, and because we do not want to fuss with a shutter click, `PictureDemo` only passes in the third parameter to `takePicture()`, as is shown in the above listing.

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();
    inPreview=true;
  }
};
```

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

---

230

`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. For example, `getSupportedFlashModes()` on `Camera.Parameters` gives you a `List` of flash mode names; the user's choice from that list can be supplied to Android via `setFlashMode()` (also on `Camera.Parameters`).

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

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.

We start with basic coverage of where you can obtain the media that you want to play back. Then, we cover how to use `MediaPlayer` for playing back audio files, such as an Ogg Vorbis clip. We then look at the use of `VideoView` for playing back video files. We wrap with brief coverage of other audio playback APIs in Android.

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 audio 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 audio clips as assets (`assets/` in your project) and reference them via `file:///android_asset/` URLs in a `Uri`. The

233

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
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
 <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_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="match_parent"
      android:layout_height="match_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
```

```
        android:textAppearance="?android:attr/textAppearanceLarge"
      />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_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="match_parent"
      android:layout_height="match_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_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="match_parent"
      android:layout_height="match_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;
```

```java
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:
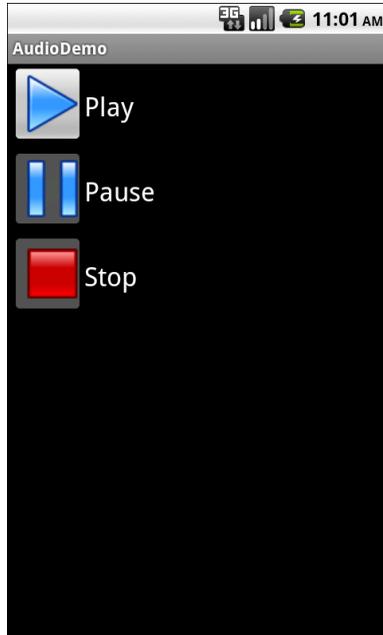
Figure 55. The AudioDemo sample application

Streaming Limitations

You can use the same basic code for streaming media, using an `http://` or `rtsp://` URL. However, bear in mind that Android does not support streaming MP3 over RTSP, as that exceeds the relevant RTSP specifications. That being said, there *are* MP3-over-RTSP streams in the world, and clients and servers that have negotiated an ad-hoc extension to the specification to accommodate this. Android cannot play these streams.

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="match_parent"
    android:layout_height="match_parent"
    >
  <VideoView
    android:id="@+id/video"
      android:layout_width="match_parent"
      android:layout_height="match_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:

```
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();
     video.start();
   }
 }
}
```

Here, we:

- Confirm that our video file exists on external storage
- Tell the VideoView which file to play

- Create a `MediaController` pop-up panel and cross-connect it to the `VideoView`

- Give the `VideoView` the focus and start playback

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 named `test.mp4` in the root of external storage on your device or emulator. Once there, the Java code shown above will give you a working video player:
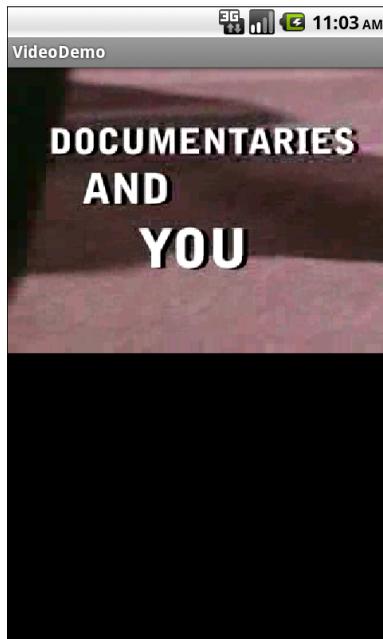
Figure 56. The VideoDemo sample application, showing a Creative Commons-licensed video clip

Tapping on the video will pop up the playback controls:

242

**Figure 57. 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 58. The VideoDemo sample application, in landscape mode, with the
video clip scaled to fit**

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.

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

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 android:versionCode="1"
          android:versionName="1.0"
```

```
          package="com.commonsware.android.sysevents.boot"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <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:icon="@drawable/cw"
               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;
```

251

```
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 directly modify the UI

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.

New Behavior With Android 3.1

It used to be that Android applications registering a BOOT_COMPLETED BroadcastReceiver would get control at boot time. Starting with Android 3.1, that may or may not occur.

If you install an application that registers a BOOT_COMPLETED receiver, and simply restart the Android 3.1 device, the receiver does not get control at boot time. It appears that the user has to start up an activity in that application first (e.g., from the launcher) before Android will deliver a BOOT_COMPLETED Intent to that application.

Google has long said that users should launch an activity from the launcher first, before that application can go do much. Preventing BOOT_COMPLETED from being delivered until the first activity is launched is a logical extension of the same argument.

Most apps will be OK with this change. For example, if your boot receiver is there to establish an AlarmManager schedule – as will be covered in a later chapter – you also needed to establish that schedule when the app is first run, so the user does not have to reboot their phone just to set up your alarms. That pattern does not change – it is just that if the user happens to reboot the phone, it will not set up your alarms, until the user runs one of your activities.

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
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.sysevents.wifi"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <application android:icon="@drawable/cw"
               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:

```java
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
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <ProgressBar android:id="@+id/bar"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_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()`:

```java
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

If you plug this into a device, it will show you the device's charge level:
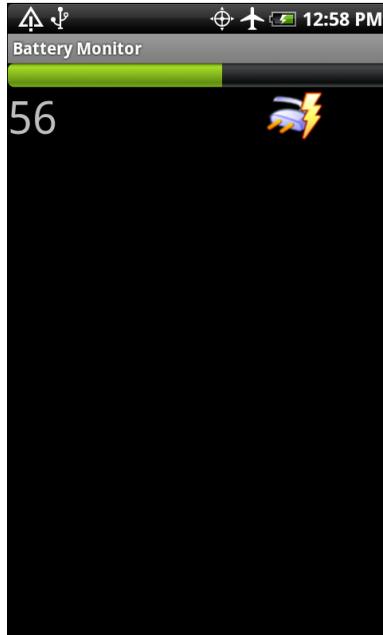
Figure 59. The BatteryMonitor application

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`.

Battery and the Emulator

Your emulator does not really have a battery. If you run this sample application on an emulator, you will see, by default, that your device has 50% fake charge remaining and that it is being charged. However, it is charged infinitely slowly, as it will not climb past 50%... at least, not without help.

While the emulator will only show fixed battery characteristics, you can change what those values are, through the highly advanced user interface known as `telnet`.

You may have noticed that your emulator title bar consists of the name of your AVD plus a number, frequently 5554. That number is not merely some engineer's favorite number. It is also an open port, on your emulator, to which you can `telnet` into, on `localhost` (127.0.0.1) on your development machine.

There are many commands you can issue to the emulator by means of `telnet`. To change the battery level, use `power capacity NN`, where `NN` is the percentage of battery life remaining that you wish the emulator to return. If you do that while you have an `ACTION_BATTERY_CHANGED` `BroadcastReceiver` registered, the receiver will receive a broadcast `Intent`, informing you of the change.

You can also experiment with some of the other `power` subcommands (e.g., `power ac on` or `power ac off`), or other commands (e.g., `geo`, to send simulated GPS fixes, just as you can do from DDMS).

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`.

Advanced Service Patterns

In *The Busy Coder's Guide to Android Development*, we covered how to create and consume services and covered some basic service patterns. However, services can certainly do more than what is covered in those introductory patterns. In this chapter, we will examine some more powerful options for services, including remote services and using services in the role of "cron jobs" or "scheduled tasks".

Remote Services

By default, services are used within the application that publishes them. However, it is possible to expose services for other applications to take advantage of. These are basically inter-process editions of the binding pattern and command patterns outlined in *The Busy Coder's Guide to Android Development*.

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 then look at how one can implement a callback system to allow services, through IPC, to pass information back to clients. After noting the possibility of binder errors, we wrap by examining other ways to get results from remote services, back to clients, without going through binding.

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("com.commonsware.android.advservice.IScript"),
            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
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.advservice"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-sdk android:minSdkVersion="3"
            android:targetSdkVersion="6" />
  <supports-screens android:largeScreens="false"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               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("com.commonsware.android.advservice.IScript"),
```

```
                  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 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:
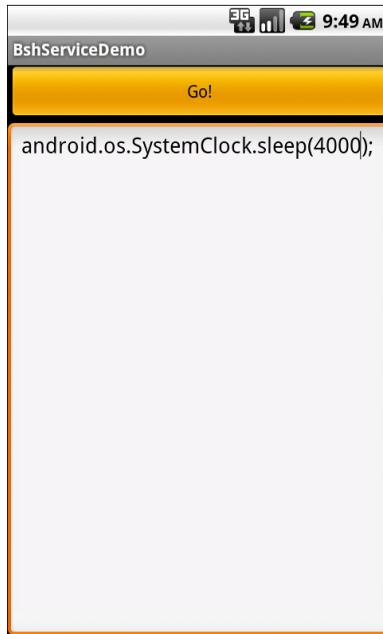
Figure 60. The BshServiceDemo application, running a long script

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:

```java
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("com.commonsware.android.advservice.IScript"),
            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.

`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());
```

279

```
        }
        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.

You may be wondering why we do not simply use an `AsyncTask`. The reason is that remote service methods exposed by AIDL are not invoked on the main application thread – one of the few places in Android where Android calls your code from a background thread. An `AsyncTask` expects to be created on the main application 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.

If the Binding Is Too Tight

Sometimes, binding is more than you really need.

Sending data to a remote service is easy, even without binding. Just package some data in `Intent` extras and use that `Intent` in a `startService()` call. The remote service can grab those extras and operate on that data. This works best with an `IntentService`, which does three things to assist with this pattern:

1. It passes the `Intents`, with their extras, to your code in `onHandleIntent()` on a background thread, so you can take as long as you want to process them

2. It queues up `Intents`, so if another one arrives while you are working on a previous one, there is no problem

3. It automatically shuts down the service when there is no more work to be done

The biggest issue is getting results back to the client. There is no possibility of a callback if there is no binding.

Fortunately, Android offers some alternatives that work nicely with this approach.

Private Broadcasts

The concept of a "private broadcast" may seem like an oxymoron, but it is something available to you in Android.

Sending a broadcast `Intent` is fairly easy – create the `Intent` and call `sendBroadcast()`. However, by default, any application could field a `BroadcastReceiver` to watch for your broadcast. This may or may not concern you.

If you feel that "spies" could be troublesome, you can call `setPackage()` on your `Intent`, to limit the distribution of the broadcast. With `setPackage()`, only components in the named application will be able to receive the broadcast. You can even arrange to send the name of the package via an extra to the remote service, so the service does not need to know the name of the package in advance.

Pending Results

Another way for a remote service to send data back to your activity is via `createPendingResult()`. This is a method on `Activity` that gives you a `PendingIntent` set up to trigger `onActivityResult()` in your activity. In essence, this is the underpinnings behind `startActivityForResult()` and `setResult()`. You create the `PendingIntent` with `createPendingResult()` and pass it in an `Intent` extra to the remote service. The remote service can call `send()` on the `PendingIntent`, supplying an `Intent` with return data, just like `setResult()` would do in an activity started via `startActivityForResult()`. In your activity's `onActivityResult()`, you would get and inspect the returned `Intent`.

This works nicely for activities, but this mechanism does not work for other components. Hence, you cannot use this technique for one service calling another remote service, for example.

BshService, Revisited

Let us take a closer look at those two techniques, as implemented in `AdvServices/RemoteClientUnbound` and `AdvServices/RemoteServiceUnbound`. These versions of the Beanshell sample are designed to demonstrate both private broadcasts and pending results.

AlarmManager: Making the Services Run On Time

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`.

The WakefulIntentService Pattern

Most times, if you are bothering to get control on a periodic basis, you will want to do so even when the device is asleep. For example, if you are writing an email client, you will want to go get new emails even if the device is asleep, so the user has all of the emails immediately upon the next time the device wakes up. You might even want to raise a `Notification` based upon the arrived emails.

Alarms that wake up the device are possible, but tricky, so we will examine `AlarmManager` in the context of this scenario. And, to make that work, we are going to use the `WakefulIntentService` – another of the CommonsWare Android Components, available as open source for you to use. In particular, we will be looking at the demo project from the `WakefulIntentService`

GitHub project, in addition to the implementation of `WakefulIntentService` itself, specifically looking at these from the standpoint of using `AlarmManager` for scheduled tasks.

Note that to use `WakefulIntentService` you will need the `WAKE_LOCK` permission in your application, and if you are using the `AlarmListener` approach described in this chapter, you will also need the `RECEIVE_BOOT_COMPLETED` permission.

Step #1: Create an Alarm Listener

The `WakefulIntentService` offers an `AlarmListener` interface. If you create one and register it properly, the `WakefulIntentService` will handle much of the details of arranging to schedule your alarms whenever they need to be scheduled, plus do the actual periodic work that you need the alarms for.

An `AlarmListener` needs to implement three methods:

- `scheduleAlarms()`, which is where you will teach Android when an alarm is supposed to go off

- `sendWakefulWork()`, which is where you tell `WakefulIntentService` what should occur when an alarm goes off

- `getMaxAge()`, where you indicate how long of a time between alarms should elapse before `WakefulIntentService` assumes that the alarms were lost and need to be re-scheduled

For example, from the `WakefulIntentService` demo project, here is an `AlarmListener` implementation named `AppListener`:

```
package com.commonsware.cwac.wakeful.demo;

import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class AppService extends WakefulIntentService {
  public AppService() {
    super("AppService");
  }
```

```
  @Override
  protected void doWakefulWork(Intent intent) {
    Log.i("AppService", "I'm awake! I'm awake! (yawn)");
  }
}
```

Your `scheduleAlarms()` method will be passed everything you should need to define when you want to get control again in the future. Mostly, this is in the form of an `AlarmManager`, to save you from having to call `getSystemService()` on a `Context` to get an `AlarmManager` instance. `AlarmManager` has three methods for defining when your scheduled tasks should run:

- `setRepeating()`, where you stipulate when the first time is you want your task to run, and how frequently thereafter

- `setInexactRepeating()`, which does the same thing but gives Android the flexibility to slightly tweak your schedule for more efficiency

- `set()`, which is for "one-shot" alarms, where you want a task to run once at a designated time

The `AlarmListener` pattern is for use with the first two approaches, not for one-shot alarms via `set()`.

Each of these methods takes a "type" of alarm. There are four types, based upon two orthogonal axes of decision:

- Do you want the device to wake up out of sleep mode to perform the task?

- Do you want to specify the time for the first task in terms of actual real world time, or simply an amount of time from now?

There are four values, therefore, you can choose from for the first parameter:

- `ELAPSED_REALTIME` indicates you want to specify a time relative to `SystemClock.elapsedRealtime()` (a good choice for specifying a time relative to now), and you do not want to wake up the device for the alarms

- `ELAPSED_REALTIME_WAKEUP` is the same as the above one, except that you do want the device to wake up out of sleep mode

- `RTC` indicates that you want to specify a time relative to `System.currentTimeMillis()` (a good choice for specifying a real-world time, as you can get a suitable value for this out of a `Calendar` object), and you do not want to wake up the device for the alarms

- `RTC_WAKEUP` is the same as the above one, except that you do want the device to wake up out of sleep mode

So, in the above sample, we are using `setInexactRepeating()`, indicating that we want the first event to occur one minute (60,000 milliseconds) from now, then recur every 15 minutes or so.

The `sendWakefulWork()` method is where you indicate to `WakefulIntentService` what is to be done when each alarm goes off. Most times, you will simply turn around and call `sendWakefulWork()` on `WakefulIntentService` itself, identifying your custom subclass of `WakefulIntentService`, where your business logic will reside. In the above sample, our business logic is in `AppService`, which we will examine shortly.

The `getMaxAge()` method... takes a bit of explaining, which we will do later in this chapter. For the moment, take it on faith that a value of twice your period is a reasonable approach.

Register the AlarmReceiver

The `WakefulIntentService` library supplies a class named `AlarmReceiver`. You will need to add this to your manifest via a `<receiver>` element, to get control when the device wakes up. For example, here is the manifest from the demo project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.cwac.wakeful.demo"
  android:versionCode="4"
  android:versionName="1.0"
  android:installLocation="auto">
```

```
  <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"/>

  <application android:label="@string/app_name">
    <receiver android:name="com.commonsware.cwac.wakeful.AlarmReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>

      <meta-data
        android:name="com.commonsware.cwac.wakeful"
        android:resource="@xml/wakeful"/>
    </receiver>

    <service android:name=".AppService">
    </service>

    <activity
      android:label="@string/app_name"
      android:name=".DemoActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Our `<receiver>` points to the `com.commonsware.cwac.wakeful.AlarmReceiver` class and ties it to a `BOOT_COMPLETED` action. This requires a `<uses-permission>` element to request the `RECEIVE_BOOT_COMPLETED` permission, so users will know at install time that we need to get control when the device turns on.

Our `<receiver>` element also has a `<meta-data>` element. These elements are used to supply more information about a component than the manifest XML calls for. In our case, we are indicating that there is additional data, known as `com.commonsware.cwac.wakeful`, available via a `wakeful` XML

287

resource (/res/xml/wakeful.xml). That resource contains a single WakefulIntentService element:

```
<WakefulIntentService
  listener="com.commonsware.cwac.wakeful.demo.AppListener"
/>
```

The one attribute in that element, listener, points to our AlarmListener implementation from the previous step.

Step #2: Do Your Wakeful Work

Our AppService will get control in a method named doWakefulWork(). The doWakefulWork() method has similar semantics to the onHandleIntent() of a regular IntentService – we get control in a background thread, and the service will shut down once the method returns if there is no other outstanding work. The difference is that WakefulIntentService will keep the device awake while doWakefulWork() is doing its work.

In this case, AppService just logs a message to LogCat:

```
package com.commonsware.cwac.wakeful.demo;

import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class AppService extends WakefulIntentService {
  public AppService() {
    super("AppService");
  }

  @Override
  protected void doWakefulWork(Intent intent) {
    Log.i("AppService", "I'm awake! I'm awake! (yawn)");
  }
}
```

And that's it. Those two steps – plus WakefulIntentService – is all you need to get control on a periodic basis to do work, waking up the phone as needed.

Of course, we have not yet scheduled any alarms. To understand how to do that, we need to first understand *when* to do that.

When Alarms Come and Go

When your app is initially installed by the user, none of your code is immediately executed. Hence, you will not have had a chance to register any alarms. So, the first place you need to think about registering your alarms is when your launcher activity is executed. In the `WakefulIntentService` demo, there is a `DemoActivity` that does just that:

```
package com.commonsware.cwac.wakeful.demo;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class DemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    WakefulIntentService.scheduleAlarms(new AppListener(),
                                        this, false);

    Toast.makeText(this, "Alarms active!",
                   Toast.LENGTH_LONG).show();

    finish();
  }
}
```

The `scheduleAlarms()` method on `WakefulIntentService` takes an implementation of your `AlarmListener`, a `Context`, and an optional boolean flag indicating whether or not the alarms should be forced to be scheduled, even if they probably are already active. By default, this is set to `true`, though our `DemoActivity` has it as `false`.

So, the first time your activity is executed, the alarms will be scheduled, following the rules set down in your `AlarmListener`. In an ideal world, this is all you would need. If this were an ideal world, though, this part of the book

would not need to be written. Since you are reading it, you can tell that this is not an ideal world.

So, when else do we need to schedule alarms?

On a Reboot

A Unix-style cron daemon will pick up jobs where it left off after a reboot. Similarly, Windows "Scheduled Tasks" will resume after a reboot.

On the other hand, `AlarmManager` in Android is wiped clean, starting with zero scheduled alarms. The argument that we have been given is that the core Android team does not want apps to schedule alarms and "forget" about them, causing them to pile up and slow down the device. While this is a noble objective, it does cause some annoyance, as we need to get control at boot time and arrange to reschedule the alarms.

After Being Force-Stopped

Users can go into the Settings application, find your application, and click a "Force Stop" button. When they do this, your app moves into a "stoppped" state. Nothing of your code will run again – including any `BroadcastReceiver` components that you have defined – until the user proactively runs one of your activities again, typically from the launcher. Hence, any user that force-stops your app will cause you to not receive any scheduled alarms ever again, as the existing alarms are removed and you will not be able to get control again at boot time.

This is why we are looking at scheduling the alarms every time the `DemoActivity` runs. Not only might it be the very first time the user has launched our activity, but it might be the very first time the user has launched our activity after being force-stopped.

Fortunately, `WakefulIntentService` handles these scenarios for you, so long as you call `scheduleAlarms()` whenever your app is launched. This is where `getMaxAge()` comes into play – if `WakefulIntentService` detects that an alarm

has not gone off in the number of milliseconds you return from `getMaxAge()`, it assumes that the user force-stopped your application and it schedules the alarms again.

The "Wakeful" of WakefulIntentService

Now, let us take a look "under the covers" to see how `WakefulIntentService` actually keeps the device awake long enough for `doWakefulWork()` to do its, um, wakeful work.

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 WakeLock Problem

For a _WAKEUP alarm, 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:
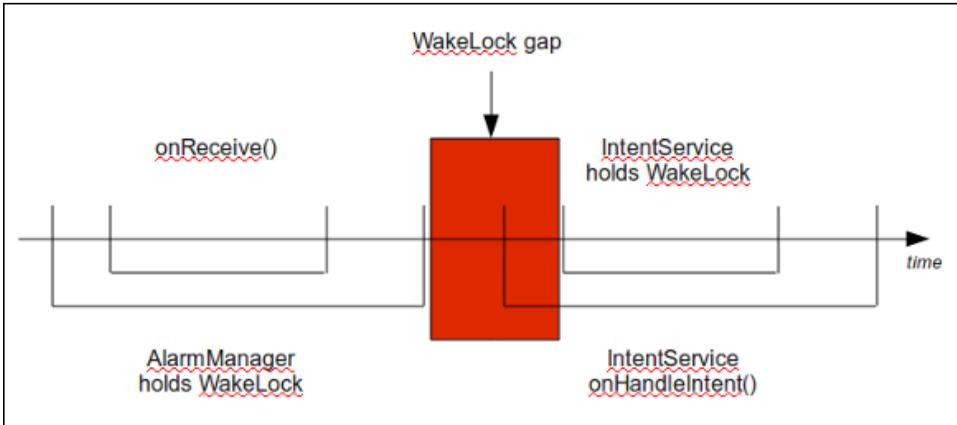
Figure 61. The WakeLock gap

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`:
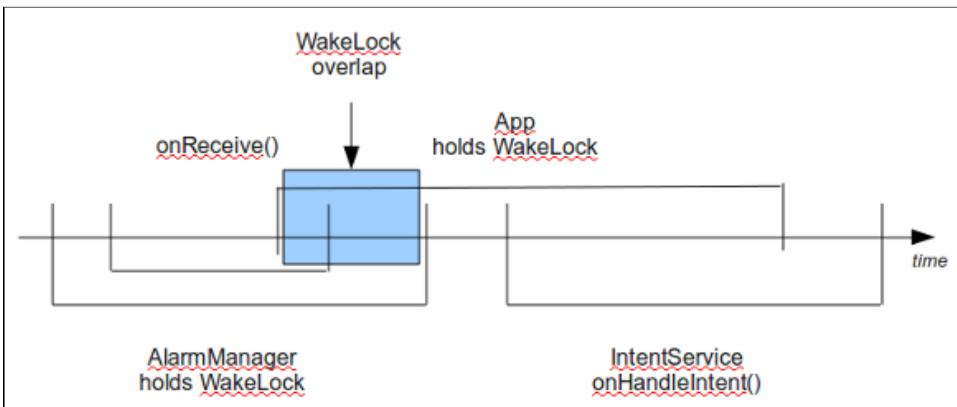
Figure 62. The WakeLock overlap

Then you are assured that the device will stay awake as long as the work remains to be done.

WakefulIntentService and WakeLocks

By now, you have noticed that the `WakefulIntentService` recipe does not have you manage your own `WakeLock`. That is because `WakefulIntentService` handles it for you. One reason why `WakefulIntentService` exists is to manage that `WakeLock`, because `WakeLocks` suffer from one major problem: they are not `Parcelable`, and therefore cannot be passed in an `Intent` extra. Hence, for our `BroadcastReceiver` and our `WakefulIntentService` to use the same `WakeLock`, they have to be shared via a static data member... which is icky. `WakefulIntentService` is designed to hide this icky part from you, so you do not have to worry about it.

Inside WakefulIntentService

With all that behind us, now we can take a look at some pieces of how `WakefulIntentService` performs its magic.

scheduleAlarms()

Our `DemoActivity` called `scheduleAlarms()`. The job of `scheduleAlarms()` is to get your `AlarmListener` to schedule the alarms, if it appears that this is necessary:

```
public static void scheduleAlarms(AlarmListener listener,
                                  Context ctxt) {
  scheduleAlarms(listener, ctxt, true);
}

public static void scheduleAlarms(AlarmListener listener,
                                  Context ctxt,
                                  boolean force) {
  SharedPreferences prefs=ctxt.getSharedPreferences(NAME, 0);
  long lastAlarm=prefs.getLong(LAST_ALARM, 0);

  if (lastAlarm==0 || force ||
      (System.currentTimeMillis()>lastAlarm &&
         System.currentTimeMillis()-lastAlarm>listener.getMaxAge())) {
```

```
    AlarmManager mgr=(AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, AlarmReceiver.class);
    PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0,
                                        i, 0);

    listener.scheduleAlarms(mgr, pi, ctxt);
  }
}
```

The `scheduleAlarms()` method will think that scheduling alarms is necessary if:

- You forced it via passing `true` to the `force` parameter, or

- If it cannot find the last time the alarm went off, via a value stored in a service-specific `SharedPreferences` object, or

- If the last time the alarm went off is longer than your desired maximum age, suggesting that perhaps the user force-stopped your app

In principle, we could just always schedule the alarms. If you call `setRepeating()` (or kin) on `AlarmManager` for a `PendingIntent`, any existing alarm for an equivalent `PendingIntent` is replaced. Here, "equivalent" means that the underlying Intent has the same action, component, categories, and MIME type.

The downside of always scheduling the alarms is that if you use `setRepeating()`, the alarm will next go off at your designated time, which may be far sooner than the existing alarm would call for. For example, suppose that you want to get control once per day. If you call `setRepeating()` or `setInexactRepeating()` every time the user launches your main activity, you will trigger the alarm to go off each of those times, in addition to the once per day. Using `setRepeating()` with a specific time of the day (e.g., 3am) will not have that effect, but `setRepeating()` with "do the first one a minute from now" would.

AlarmReceiver

The `WakefulIntentService`-supplied `AlarmReceiver` will get control when the device powers on (via our manifest entry) and when the alarm goes off. At those times, `onReceive()` will be called:

```java
@Override
 public void onReceive(Context ctxt, Intent intent) {
   AlarmListener listener=getListener(ctxt);

   if (listener!=null) {
     if (intent.getAction()==null) {
       SharedPreferences
prefs=ctxt.getSharedPreferences(WakefulIntentService.NAME, 0);

       prefs
         .edit()
         .putLong(WakefulIntentService.LAST_ALARM, System.currentTimeMillis())
         .commit();

       listener.sendWakefulWork(ctxt);
     }
     else {
       WakefulIntentService.scheduleAlarms(listener, ctxt, true);
     }
   }
 }
```

If the `Intent` that was broadcast does not have an action, we assume this is the broadcast from `AlarmManager`, and so we call `sendWakefulWork()` on your `AlarmListener` after updating our private `SharedPreferences` to note that the alarm went off (for use by `scheduleAlarms()`). If, however, the action is not-null, that means we are being started from the `BOOT_COMPLETED` broadcast, in which case we call `scheduleAlarms()` on your `AlarmListener`.

Where does the `AlarmListener` object come from? A private `getListener()` method that takes advantage of the `<meta-data>` from the manifest:

```java
@SuppressWarnings("unchecked")
private WakefulIntentService.AlarmListener getListener(Context ctxt) {
  PackageManager pm=ctxt.getPackageManager();
  ComponentName cn=new ComponentName(ctxt, getClass());

  try {
    ActivityInfo ai=pm.getReceiverInfo(cn,
                                       PackageManager.GET_META_DATA);
```

```
    XmlResourceParser xpp=ai.loadXmlMetaData(pm,
                                             WAKEFUL_META_DATA);

    while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
      if (xpp.getEventType()==XmlPullParser.START_TAG) {
        if (xpp.getName().equals("WakefulIntentService")) {
          String clsName=xpp.getAttributeValue(null, "listener");
          Class<AlarmListener> cls=(Class<AlarmListener>)Class.forName(clsName);

          return(cls.newInstance());
        }
      }

      xpp.next();
    }
  }
  catch (NameNotFoundException e) {
    Log.e(getClass().getName(), "Cannot find own info???", e);
  }
  catch (XmlPullParserException e) {
    Log.e(getClass().getName(), "Malformed metadata resource XML", e);
  }
  catch (IOException e) {
    Log.e(getClass().getName(), "Could not read resource XML", e);
  }
  catch (ClassNotFoundException e) {
    Log.e(getClass().getName(), "Listener class not found", e);
  }
  catch (IllegalAccessException e) {
    Log.e(getClass().getName(), "Listener is not public or lacks public
constructor", e);
  }
  catch (InstantiationException e) {
    Log.e(getClass().getName(), "Could not create instance of listener", e);
  }

  return(null);
}
```

To read data from a `<meta-data>` element, you use `PackageManager` to get information about your `BroadcastReceiver` via `getReceiverInfo()` and the `ActivityInfo` object it returns. You can ask the `ActivityInfo` object to get you the metadata for a given name via the `loadXmlMetaData()` method, which returns an `XmlResourceParser`. This is an implementation of the `XmlPullParser` interface, so you can start iterating over the parse events until you encounter your desired element. In this case, we scan for the start-tag event for the `WakefulIntentService` element, then read out the listener attribute, use `Class.forName()` to find the class, and call `newInstance()` on the `Class` object to create an instance of your `AlarmListener`.

An alternative approach would be to have you subclass `AlarmReceiver` and override `getListener()`, and you are certainly welcome to do that if the `<meta-data>` approach causes you difficulty.

So, the net of `AlarmReceiver` is that we allow you to schedule your alarms (if the device is freshly booted) or we allow you to do your wakeful work (if the alarm is from `AlarmManager`).

Command Processing

`AlarmReceiver`, when triggered by `AlarmManager`, detects that there is no supplied action and, in that case, passes control to `WakefulIntentService` via `sendWakefulWork()`. You can call this method yourself from other, non-alarm scenarios where you want work to be done that can be completed without the device falling asleep.

There are two flavors of `sendWakefulWork()`. Both take a `Context` – the difference is in the second parameter. One takes a `Class` object, identifying the `WakefulIntentService` subclass that should be invoked to do the wakeful work. This is a convenience method, wrapping around the other flavor of `sendWakefulWork()` that takes an `Intent`.

Either flavor of `sendWakefulWork()` on `WakefulIntentService` eventually routes to a `getLock()` method:

```
static final String NAME="com.commonsware.cwac.wakeful.WakefulIntentService";
static final String LAST_ALARM="lastAlarm";
private static volatile PowerManager.WakeLock lockStatic=null;

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,
                          NAME);
    lockStatic.setReferenceCounted(true);
  }

  return(lockStatic);
}
```

```
public static void sendWakefulWork(Context ctxt, Intent i) {
  getLock(ctxt.getApplicationContext()).acquire();
  ctxt.startService(i);
}
```

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 AlarmReceiver, 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, sendWakefulWork() starts up our service and exits. Since this is the only thing onReceive() was doing in AlarmReceiver, onReceive() exits. Notably, AlarmReceiver does not release the WakeLock it acquired via sendWakefulWork(). 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.

So, WakefulIntentService will now get control, under an active WakeLock. Since it is an IntentService subclass, onHandleIntent() is called. Here, we just route control to the subclass' implementation of an abstract doWakefulWork() method, ensuring that we release the WakeLock when the work is done, even if a RuntimeException is raised:

```
@Override
final protected void onHandleIntent(Intent intent) {
  try {
```

```
    doWakefulWork(intent);
  }
  finally {
    getLock(this.getApplicationContext()).release();
  }
}
```

As a result, each piece of work that gets sent to the `WakefulIntentService` will acquire a `WakeLock` via `sendWakefulWork()` and will release that `WakeLock` when `doWakefulWork()` ends. Once that `WakeLock` is fully released, the device can fall back asleep.

Background Data Setting

Users can check or uncheck a checkbox in the Settings application that indicates if they want applications to use the Internet in the background. Services employing `AlarmManager` should honor this setting.

To find out whether background data is allowed, use the `ConnectivityManager` system service and call `getBackgroundDataSetting()`. For example, your alarm-triggered `BroadcastReceiver` could check this before bothering to arrange for the `IntentService` (or `WakefulIntentService`) to do work.

You can also register a `BroadcastReceiver` to watch for the `ACTION_BACKGROUND_DATA_SETTING_CHANGED` broadcast, also defined on `ConnectivityManager`. For example, you could elect to completely cancel your alarm if the background data setting is flipped to `false`.

The "Everlasting Service" Anti-Pattern

One anti-pattern that is all too prevalent in Android is the "everlasting service". Such a service is started via `startService()` and never stops – the component starting it does not stop it and it does not stop itself via `stopSelf()`.

Why is this an anti-pattern?

- The service takes up memory all of the time. This is bad in its own right if the service is not continuously delivering sufficient value to be worth the memory.

- Users, fearing services that sap their device's CPU or RAM, may attack the service with so-called "task killer" applications or may terminate the service via the Settings app, thereby defeating your original goal.

- Android itself, due to user frustration with sloppy developers, will terminate services it deems ill-used, particularly ones that have run for quite some time.

Occasionally, an everlasting service is the right solution. Take a VOIP client, for example. A VOIP client usually needs to hold an open socket with the VOIP server to know about incoming calls. The only way to continuously watch for incoming calls is to continuously hold open the socket. The only component capable of doing that would be a service, so the service would have to continuously run.

However, in the case of a VOIP client, or a music player, the user is the one specifically requesting the service to run forever. By using `startForeground()`, a service can ensure it will not be stopped due to old age for cases like this.

As a counter-example, imagine an email client. The client wishes to check for new email messages periodically. The right solution for this is the `AlarmManager` pattern described earlier in this chapter. The anti-pattern would have a service running constantly, spending most of its time waiting for the polling period to elapse (e.g., via `Thread.sleep()`). There is no value to the user in taking up RAM to watch the clock tick. Such services should be rewritten to use `AlarmManager`.

Most of the time, though, it appears that services are simply leaked. That is one advantage of using `AlarmManager` and an `IntentService` – it is difficult to leak the service, causing it to run indefinitely. In fact, `IntentService` in general is a great implementation to use whenever you use the command pattern, as it ensures that the service will shut down eventually. If you use a

regular service, be sure to shut it down when it is no longer actively delivering value to the user.

Using System Settings and 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.

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

- HAPTIC_MODE_ENABLED to control whether the user receives "haptic feedback" (vibrations) from things like the MENU button

- ACCELEROMETER_ROTATION to control whether the screen orientation will change based on the position of the device

The SystemServices/Settings project has a SettingsSetter sample application that displays a checklist:

```xml
<?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="match_parent"
  android:layout_height="match_parent"
/>
```
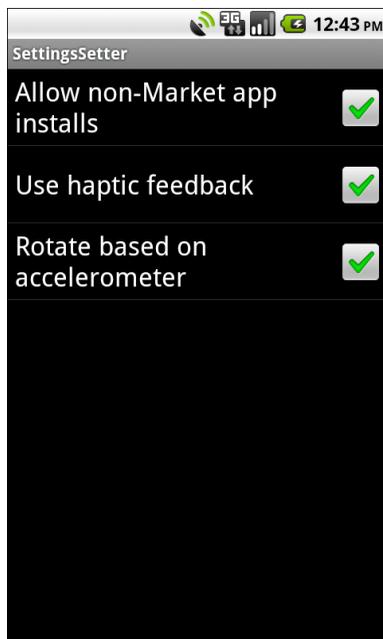

Figure 63. The SettingsSetter application

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));
      }
    }
    catch (Throwable t) {
      Log.e("SettingsSetter", "Exception in setChecked()", t);
    }
```

305

```
    }
}
```

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:

```java
@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
<?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 Android Market setting, it does not seem to take effect. And, if you look at the LogCat output, you will see complaints.

Once upon a time, you could modify this setting, and others like it.

Now, though, these settings are ones that Android deems "secure". The constants have been moved from `Settings.System` to `Settings.Secure`, though the old constants are still there, flagged as deprecated.

These so-called "secure" settings are ones that Android does not allow applications to change. While theoretically the `WRITE_SECURE_SETTINGS` permission resolves this problem, ordinary SDK applications cannot hold that permission. 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
<?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="match_parent"
  android:layout_height="match_parent"
>
  <TableRow
    android:paddingTop="10px"
    android:paddingBottom="20px">
    <TextView android:text="Alarm:" />
    <SeekBar
      android:id="@+id/alarm"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="Music:" />
    <SeekBar
      android:id="@+id/music"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    />
  </TableRow>
  <TableRow
    android:paddingBottom="20px">
    <TextView android:text="Ring:" />
    <SeekBar
      android:id="@+id/ring"
      android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow
      android:paddingBottom="20px">
      <TextView android:text="System:" />
      <SeekBar
        android:id="@+id/system"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow>
      <TextView android:text="Voice:" />
      <SeekBar
        android:id="@+id/voice"
        android:layout_width="match_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:

```java
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);
  }
```

310

```
  private void initBar(SeekBar bar, final int stream) {
    bar.setMax(mgr.getStreamMaxVolume(stream));
    bar.setProgress(mgr.getStreamVolume(stream));

    bar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
      public void onProgressChanged(SeekBar bar, int progress,
                                    boolean fromUser) {
        mgr.setStreamVolume(stream, progress,
                            AudioManager.FLAG_PLAY_SOUND);
      }

      public void onStartTrackingTouch(SeekBar bar) {
        // no-op
      }

      public void onStopTrackingTouch(SeekBar bar) {
        // no-op
      }
    });
  }
}
```

In initBar(), we set the appropriate size for the SeekBar 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.

The net result is that when the user slides a SeekBar, it adjusts the stream to match:
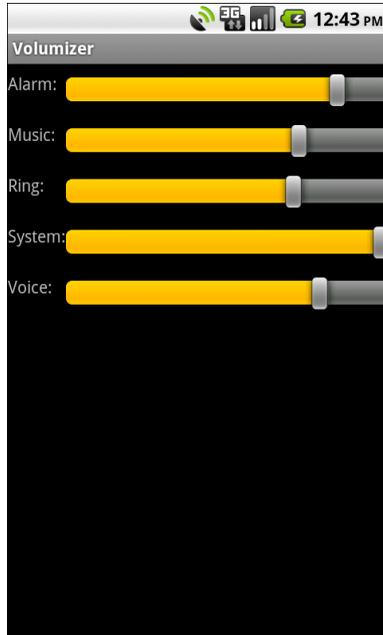
Figure 64. The Volumizer application

Putting Stuff on the Clipboard

Being able to copy and paste is something that mobile device users seem to want almost as much as their desktop brethren. Most of the time, we think of this as copying and pasting text, and for a long time that was all that was possible on Android. Android 3.0 added in new clipboard capabilities for more rich content, which application developers can choose to support as well. This section will cover both of these techniques.

Using the Clipboard on Android 1.x/2.x

Android has a `ClipboardManager` that allows you to interact with the clipboard manually, in addition to built-in clipboard facilities for users (e.g., copy/paste context menus on `EditText`). `ClipboardManager`, like `AudioManager`, is obtained via a call to `getSystemService()`:

```
ClipboardManager cm=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
```

312

From there, you have three simple methods:

- `getText()` to retrieve the current clipboard contents

- `hasText()`, to determine if there are any clipboard contents, so you can react accordingly (e.g., disable "paste" menus when there is nothing to paste)

- `setText()`, to put text on the clipboard

For example, `SystemServices/ClipIP` is a little application that puts your current IP address on the clipboard, for pasting into some `EditText` of an application. The UI is simply an `EditText` that you can use to test out the paste operation:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Long-tap me to paste!"
    />
</LinearLayout>
```

The `IPClipper` activity's `onCreate()` does the work of putting text onto the clipboard via `setText()` and notifying the user via a `Toast`:

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

  try {
    String addr=getLocalIPAddress();

    if (addr==null) {
      Toast.makeText(this,
                  "IP address not available – are you online?",
                  Toast.LENGTH_LONG)
          .show();
    }
    else {
      ClipboardManager cm=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
```

313

```
      cm.setText(addr);
      Toast.makeText(this, "IP Address clipped!", Toast.LENGTH_SHORT)
            .show();
    }
  }
  catch (Exception e) {
    Log.e("IPClipper", "Exception getting IP address", e);
    Toast.makeText(this,
                  "Could not obtain IP address",
                  Toast.LENGTH_LONG)
          .show();
  }
}
```

The work of figuring out what the IP address is can be found in the
`getLocalIPAddress()` method:

```
public String getLocalIPAddress() throws SocketException {
  Enumeration<NetworkInterface> nics=NetworkInterface.getNetworkInterfaces();

  while (nics.hasMoreElements()) {
    NetworkInterface intf=nics.nextElement();
    Enumeration<InetAddress> addrs=intf.getInetAddresses();

    while (addrs.hasMoreElements()) {
      InetAddress addr=addrs.nextElement();

      if (!addr.isLoopbackAddress()) {
        return(addr.getHostAddress().toString());
      }
    }
  }

  return(null);
}
```

This uses the `NetworkInterface` and `InetAddress` classes from the `java.net`
package to loop through all network interfaces and find the first one that
has a non-localhost (loopback) IP address. The emulator will return
10.0.2.15 all of the time; your device will return whatever IP address it has
from WiFi, 3G, etc. If no such address is available, it returns `null`.

After starting the activity, the user will hopefully see the "successful" `Toast`:
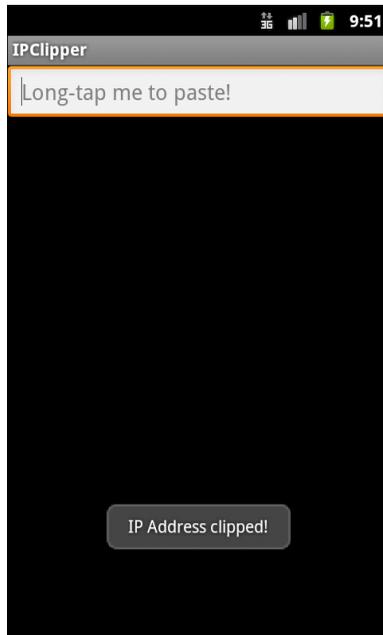
Figure 65. The IPClipper, shortly after launching

Then, if the user long-taps on the `EditText` and chooses Paste, the IP address is added to the `EditText` contents:
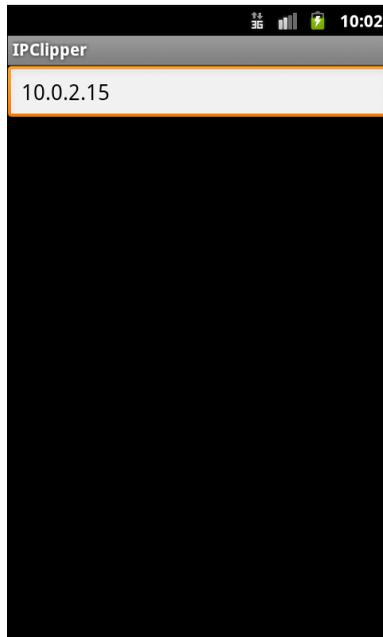
Figure 66. The IPClipper, after the user pastes the IP address into the EditText

Note that the clipboard is system-wide, not merely application-wide. You can test this by pasting the IP address into the EditText of some other application.

Advanced Clipboard on Android 3.x

Android 3.0 added in new ways of working with ClipboardManager to clip things that transcend simple text. In part, this is expected to be used for advanced copy and paste features between applications. However, this also forms the foundation for a rich drag-and-drop model within an application.

Note that they also moved ClipboardManager to the android.content package. You can still refer to it via the android.text package, for backwards compatibility. However, if your project will be on API Level 11 or higher only, you might consider using the new android.content package edition of the class.

316

Copying Rich Data to the Clipboard

In addition to methods like `setText()` to put a piece of plain text on the clipboard, `ClipboardManager` (as of API Level 11) offers `setPrimaryClip()`, which allows you to put a `ClipData` object on the clipboard.

What's a `ClipData`? In some respects, it is whatever you want. It can hold:

- plain text

- a `Uri` (e.g., to a piece of music)

- an `Intent`

The `Uri` means that you can put anything on the clipboard that can be referenced by a `Uri`… and if there is nothing in Android that lets you reference some data via a `Uri`, you can invent your own content provider to handle that chore for you. Furthermore, a single `ClipData` can actually hold as many of these as you want, each represented as individual `ClipData.Item` objects. As such, the possibilities are endless.

There are static factory methods on `ClipData`, such as `newUri()`, that you can use to create your `ClipData` objects. In fact, that is what we use in the SystemServices/ClipMusic sample project and the `MusicClipper` activity.

`MusicClipper` has the classic two-big-button layout:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="Pick"
    android:onClick="pickMusic"
  />
  <Button android:id="@+id/view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

317

```
    android:layout_weight="1"
    android:text="Play"
    android:onClick="playMusic"
  />
</LinearLayout>
```
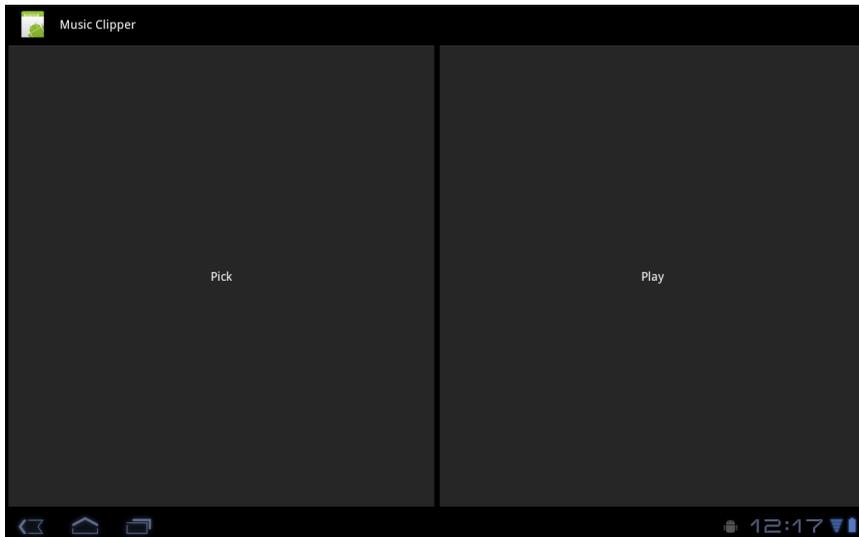

Figure 67. The Music Clipper main screen

In onCreate(), we get our hands on our ClipboardManager system service:

```
private ClipboardManager clipboard=null;

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

  clipboard=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
}
```

Tapping the "Pick" button will let you pick a piece of music, courtesy of the pickMusic() method wired to that Button object:

```
public void pickMusic(View v) {
  Intent i=new Intent(Intent.ACTION_GET_CONTENT);

  i.setType("audio/*");
  startActivityForResult(i, PICK_REQUEST);
}
```

318

Here, we tell Android to let us pick a piece of music from any available audio MIME type (audio/*). Fortunately, Android has an activity that lets us do that:
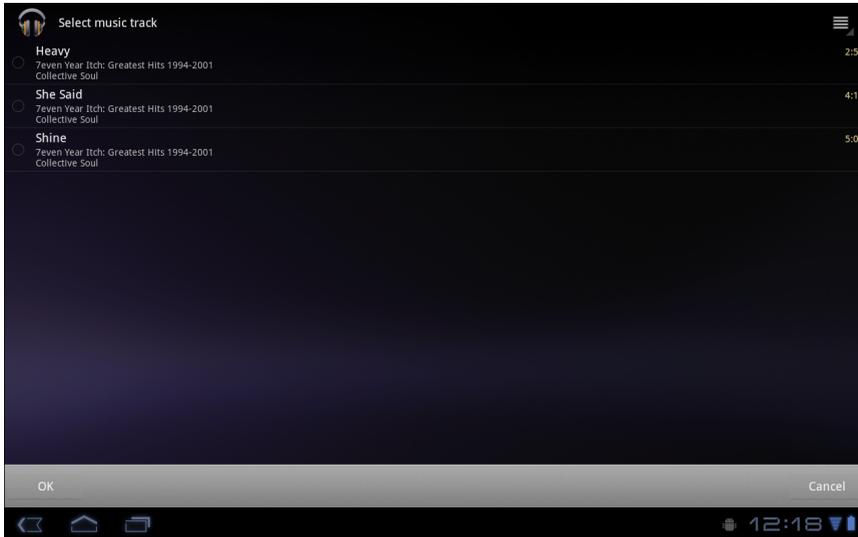
Figure 68. The XOOM tablet's music track picker

We get the result in onActivityResult(), since we used startActivityForResult() to pick the music. There, we package up the content:// Uri to the music into a ClipData object and put it on the clipboard:

```java
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==PICK_REQUEST) {
    if (resultCode==RESULT_OK) {
      ClipData clip=ClipData.newUri(getContentResolver(),
                                "Some music", data.getData());

      clipboard.setPrimaryClip(clip);
    }
  }
}
```

Pasting Rich Data from the Clipboard

The catch with rich data on the clipboard is that somebody has to know about the sort of information you are placing on the clipboard. Eventually, the Android development community will work out common practices in this area. Right now, though, you can certainly use it within your own application (e.g., clipping a note and pasting it into another folder).

Since putting `ClipData` onto the clipboard involves a call to `setPrimaryClip()`, it should not be surprising that the reverse operation – getting a `ClipData` from the clipboard – uses `getPrimaryClip()`. However, since you do not know where this clip came from, you need to validate that it has what you expect and to let the user know when the clipboard contents are not something you can leverage.

The "Play" button in our UI is wired to a `playMusic()` method. This will only work when we have pasted a `Uri ClipData` to the clipboard pointing to a piece of music. Since we cannot be sure that the user has done that, we have to sniff around:

```java
public void playMusic(View v) {
  ClipData clip=clipboard.getPrimaryClip();

  if (clip==null) {
    Toast.makeText(this, "There is no clip!", Toast.LENGTH_LONG).show();
  }
  else {
    ClipData.Item item=clip.getItemAt(0);
    Uri song=item.getUri();

    if (song!=null &&
        getContentResolver().getType(song).startsWith("audio/")) {
      startActivity(new Intent(Intent.ACTION_VIEW, song));
    }
    else {
      Toast.makeText(this, "There is no song!", Toast.LENGTH_LONG).show();
    }
  }
}
```

First, there may be nothing on the clipboard, in which case the `ClipData` returned by `getPrimaryClip()` would be `null`. Or, there may be stuff on the clipboard, but it may not have a `Uri` associated with it (`getUri()` on

`ClipData`). Even then, the `Uri` may point to something other than music, so even if we get a `Uri`, we need to use a `ContentResolver` to check the MIME type (`getContentResolver().getType()`) and make sure it seems like it is music (e.g., starts with `audio/`). Then, and only then, does it make sense to try to start an `ACTION_VIEW` activity on that `Uri` and hope that something useful happens. Assuming you clipped a piece of music with the "Pick" button, "Play" will kick off playback of that song.

ClipData and Drag-and-Drop

Android 3.0 also introduced Android's first built-in drag-and-drop framework. One might expect that this would related entirely to `View` and `ViewGroup` objects and have nothing to do with the clipboard. In reality, the drag-and-drop framework leverages `ClipData` to say what it is that is being dragged and dropped. You call `startDrag()` on a `View`, supplying a `ClipData` object, along with some objects to help render the "shadow" that is the visual representation of this drag operation. A `View` that can receive objects "dropped" via drag-and-drop needs to register an `OnDragListener` to receive drag events as the user slides the shadow over top of the `View` in question. If the user lifts their finger, thereby dropping the shadow, the recipient `View` will get an `ACTION_DROP` drag event, and can get the `ClipData` out of the event.

The Rest of the Gang

There are quite a few system services you can get from `getSystemService()`. Beyond the ones profiled in this chapter, you have access to:

- `AccessibilityManager`, for being notified of key system events (e.g., activities starting) that might be relayed to users via haptic feedback, audio prompts, or other non-visual cues

- `AccountManager`, for working with Android's system of user accounts and synchronization

- `ActivityManager`, for getting more information about what processes and components are presently running on the device

- `AlarmManager`, for scheduled tasks (a.k.a., "cron jobs"), covered elsewhere in this book

- `ConnectivityManager`, for a high-level look as to what sort of network the device is connected to for data (e.g., WiFi, 3G)

- `DevicePolicyManager`, for accessing device administration capabilities, such as wiping the device

- `DownloadManager`, for downloading large files on behalf of the user, covered in *The Busy Coder's Guide to Android Development*

- `DropBoxManager`, for maintaining your own ring buffers of logging information akin to LogCat

- `InputMethodManager`, for working with input method editors

- `KeyguardManager`, for locking and unlocking the keyguard, where possible

- `LayoutInflater`, for inflating layout XML files into Views, covered in *The Busy Coder's Guide to Android Development*

- `LocationManager`, for determining the device's location (e.g., GPS), covered in *The Busy Coder's Guide to Android Development*

- `NotificationManager`, for putting icons in the status bar and otherwise alerting users to things that have occurred asynchronously, covered in *The Busy Coder's Guide to Android Development*

- `PowerManager`, for obtaining `WakeLock` objects and such, covered elsewhere in this book

- `SearchManager`, for interacting with the global search system – search in general is covered elsewhere in this book

- `SensorManager`, for accessing data about sensors, such as the accelerometer

- `TelephonyManager`, for finding out about the state of the phone and related data (e.g., SIM card details)

- `UiModeManager`, for dealing with different "UI modes", such as being docked in a car or desk dock

- `Vibrator`, for shaking the phone (e.g., haptic feedback)

- `WifiManager`, for getting more details about the active or available WiFi networks

- `WindowManager`, mostly for accessing details about the default display for the device

Content Provider Theory

Android publishes data to you via an abstraction known as a "content provider". Access to contacts and the call log, for example, are given to you via a set of content providers. In a few places, Android expects you to supply a content provider, such as for integrating your own search suggestions with the Android Quick Search Box. And, content providers are one way for you to supply data to third party applications, or to consume information from third party applications. As such, content providers have the potential to be something you would encounter frequently, even if in practice they do not seem used much.

Using a Content Provider

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

Given a `Uri`, you may be able to perform basic CRUD (create, read, update, delete) operations using a content provider. `Uri` instances can represent either collections or individual pieces of content. Given a collection `Uri`, you may be able to create new pieces of content via insert operations. Given an

325

instance `Uri`, you may be able to read data represented by the `Uri`, update that data, or delete the instance outright. Or, given an `Uri`, you may be able to open up a handle to what amounts to a file, that you can read and, possibly, write to.

These are all phrased as "may" because the content provider system is a facade. The actual implementation of a content provider dictates what you can and cannot do, and not all content providers will support all capabilities.

Pieces of Me

The simplified model of the construction of a content `Uri` is the scheme, the namespace of data, and, optionally, the instance identifier, all separated by slashes in URL-style notation. The scheme of a content Uri is always `content://`.

So, a content `Uri` of `content://constants/5` represents the constants instance with an identifier of `5`.

The combination of the scheme and the namespace is known as the "base Uri" of a content provider, or a set of data supported by a content provider. In the example above, `content://constants` is the base `Uri` for a content provider that serves up information about "constants" (in this case, physical constants).

The base `Uri` can be more complicated. For example, if the base `Uri` for contacts were `content://contacts/people`, the contacts content provider may serve up other data using other base `Uri` values.

The base `Uri` represents a collection of instances. The base `Uri` combined with an instance identifier (e.g., 5) represents a single instance.

Most of the Android APIs expect these to be `Uri` objects, though in common discussion, it is simpler to think of them as strings. The `Uri.parse()` static method creates a `Uri` out of the string representation.

Getting a Handle

So, where do these `Uri` instances come from?

The most popular starting point, if you know the type of data you want to work with, is to get the base `Uri` from the content provider itself in code. For example, `CONTENT_URI` is the base `Uri` for contacts represented as people – this maps to `content://contacts/people`. If you just need the collection, this `Uri` works as-is; if you need an instance and know its identifier, you can call `addId()` on the `Uri` to inject it, so you have a `Uri` for the instance.

You might also get `Uri` instances handed to you from other sources, such as getting `Uri` handles for contacts via sub-activities responding to `ACTION_PICK` intents. In this case, the `Uri` is truly an opaque handle...unless you decide to pick it apart using the various getters on the `Uri` class.

You can also hard-wire literal `String` objects (e.g., `"content://contacts/people"`) and convert them into `Uri` instances via `Uri.parse()`. This is not an ideal solution, as the base `Uri` values could conceivably change over time. For example, the contacts content provider's base `Uri` is no longer `content://contacts/people` due to an overhaul of that subsystem. However, when you integrate with content providers from third parties, most likely you will not have a choice but to "hard-wire" in the content `Uri` based on a string.

The Database-Style API

Of the two flavors of API that a content provider may support, the database-style API is more prevalent. Using a `ContentResolver`, you can perform standard "CRUD" operations (create, read, update, delete) using what looks like a SQL interface.

Makin' Queries

Given a base `Uri`, you can run a query to return data out of the content provider related to that `Uri`. This has much of the feel of SQL: you specify the "columns" to return, the constraints to determine which "rows" to return, a sort order, etc. The difference is that this request is being made of a content provider, not directly of some database (e.g., SQLite).

While you can conduct a query using a `ContentResolver`, another approach is the `managedQuery()` method available to your activity. This method takes five parameters:

1. The base `Uri` of the content provider to query, or the instance `Uri` of a specific object to query

2. An array of properties (think "columns") from that content provider that you want returned by the query

3. A constraint statement, functioning like a SQL `WHERE` clause

4. An optional set of parameters to bind into the constraint clause, replacing any `?` that appear there

5. An optional sort statement, functioning like a SQL `ORDER BY` clause

This method returns a `Cursor` object, which you can use to retrieve the data returned by the query.

This will hopefully make more sense given an example.

Our content provider examples come from the `ContentProvider/ConstantsPlus` sample application, specifically the `ConstantsBrowser` class. Here, we make a call to our `ContentProvider` via `managedQuery()`:

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

  constantsCursor=managedQuery(Provider.Constants.CONTENT_URI,
                        PROJECTION, null, null, null);
```

```
ListAdapter adapter=new SimpleCursorAdapter(this,
                    R.layout.row, constantsCursor,
                    new String[] {Provider.Constants.TITLE,
                                  Provider.Constants.VALUE},
                    new int[] {R.id.title, R.id.value});

setListAdapter(adapter);
registerForContextMenu(getListView());
}
```

In the call to managedQuery(), we provide:

- The Uri passed into the activity by the caller (CONTENT_URI), in this case representing the collection of physical constants managed by the content provider

- A list of properties to retrieve (see code below)

- Three null values, indicating that we do not need a constraint clause (the Uri represents the instance we need), nor parameters for the constraint, nor a sort order (we should only get one entry back)

The biggest "magic" here is the list of properties. The lineup of what properties are possible for a given content provider should be provided by the documentation (or source code) for the content provider itself. In this case, we define logical values on the Provider content provider implementation class that represent the various properties (namely, the unique identifier, the display name or title, and the value of the constant).

Adapting to the Circumstances

Now that we have a Cursor via managedQuery(), we have access to the query results and can do whatever we want with them. You might, for example, manually extract data from the Cursor to populate widgets or other objects.

However, if the goal of the query was to return a list from which the user should choose an item, you probably should consider using SimpleCursorAdapter. This class bridges between the Cursor and a selection widget, such as a ListView or Spinner. Pour the Cursor into a

SimpleCursorAdapter, hand the adapter off to the widget, and you are set –
your widget will show the available options.

After executing the managedQuery() and getting the Cursor, ConstantsBrowser
creates a SimpleCursorAdapter with the following parameters:

- The activity (or other Context) creating the adapter; in this case, the
  ConstantsBrowser itself

- The identifier for a layout to be used for rendering the list entries
  (R.layout.row)

- The cursor (constantsCursor)

- The properties to pull out of the cursor and use for configuring the
  list entry View instances (TITLE and VALUE)

- The corresponding identifiers of TextView widgets in the list entry
  layout that those properties should go into (R.id.title and
  R.id.value)

If you need more control over the views than you can reasonably achieve
with the stock view construction logic, subclass SimpleCursorAdapter and
override getView() to create your own widgets to go into the list, as
demonstrated earlier in this book.

And, of course, you can manually manipulate the Cursor (e.g.,
moveToFirst(), getString()), just like you can with a database Cursor.

Give and Take

Of course, content providers would be astonishingly weak if you couldn't
add or remove data from them, only update what is there. Fortunately,
content providers offer these abilities as well.

To insert data into a content provider, you have two options available on
the ContentProvider interface (available through getContentProvider() to
your activity):

1. Use `insert()` with a collection `Uri` and a `ContentValues` structure describing the initial set of data to put in the row

2. Use `bulkInsert()` with a collection `Uri` and an array of `ContentValues` structures to populate several rows at once

The `insert()` method returns a `Uri` for you to use for future operations on that new object. The `bulkInsert()` method returns the number of created rows; you would need to do a query to get back at the data you just inserted.

For example, here is a snippet of code from `ConstantsBrowser` to insert a new constant into the content provider, given a `DialogWrapper` that can provide access to the title and value of the constant:

```
private void processAdd(DialogWrapper wrapper) {
  ContentValues values=new ContentValues(2);

  values.put(Provider.Constants.TITLE, wrapper.getTitle());
  values.put(Provider.Constants.VALUE, wrapper.getValue());

  getContentResolver().insert(Provider.Constants.CONTENT_URI,
                              values);
  constantsCursor.requery();
}
```

Since we already have an outstanding `Cursor` for the content provider's contents, we call `requery()` on that to update the `Cursor`'s contents. This, in turn, will update any `SimpleCursorAdapter` you may have wrapping the `Cursor` – and that will update any selection widgets (e.g., `ListView`) you have using the adapter.

To delete one or more rows from the content provider, use the `delete()` method on `ContentResolver`. This works akin to a SQL DELETE statement and takes three parameters:

1. A `Uri` representing the collection (or instance) from which you wish to delete rows

2. A constraint statement, functioning like a SQL WHERE clause, to determine which rows should be deleted

3. An optional set of parameters to bind into the constraint clause, replacing any ? that appear there

The File System-Style API

Sometimes, what you are trying to retrieve does not look like a set of rows and columns, but rather looks like a file. For example, the `MediaStore` content provider manages the index of all music, video, and image files available on external storage, and you can use `MediaStore` to open up any such file you find.

Some content providers, like `MediaStore`, support both the database-style and file system-style APIs – you query to find media that matches your criteria, then can open some file that matches. Other content providers might only support the file system-style API.

Given a `Uri` that represents some file managed by the content provider, you can use `openInputStream()` and `openOutputStream()` on a `ContentResolver` to access an `InputStream` or `OutputStream`, respectively. Note, though, that not all content providers may support both modes. For example, a content provider that serves files stored inside the application (e.g., assets in the APK file), you will not be able to get an `OutputStream` to modify the content.

Building Content Providers

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

That being said, building a content provider is of huge importance if your application wishes to make data available to other applications. If your application is keeping its data solely to itself, you may be able to avoid creating a content provider, just accessing the data directly from your

activities. But, if you want your data to possibly be used by others – for example, you are building a feed reader and you want other programs to be able to access the feeds you are downloading and caching – then a content provider is right for you.

This chapter shows some sample bits of code from the `ContentProvider/ConstantsPlus` application. This is the same basic application as was first shown back in the chapter on database access in *The Busy Coder's Guide to Android Development*, but rewritten to pull the database logic into a content provider, which is then used by the activity.

First, Some Dissection

As was discussed in the previous chapter, the content `Uri` is the linchpin behind accessing data inside a content provider. When using a content provider, all you really need to know is the provider's base `Uri`; from there you can run queries as needed, or construct a `Uri` to a specific instance if you know the instance identifier.

When building a content provider, though, you need to know a bit more about the innards of the content `Uri`.

A content `Uri` has two to four pieces, depending on situation:

- It always has a scheme (`content://`), indicating it is a content `Uri` instead of a `Uri` to a Web resource (`http://`).

- It always has an authority, which is the first path segment after the scheme. The authority is a unique string identifying the content provider that handles the content associated with this `Uri`.

- It may have a data type path, which is the list of path segments after the authority and before the instance identifier (if any). The data type path can be empty, if the content provider only handles one type of content. It can be a single path segment (`foo`) or a chain of path segments (`foo/bar/goo`) as needed to handle whatever data access scenarios the content provider requires.

- It may have an instance identifier, which is an integer identifying a specific piece of content. A content `Uri` without an instance identifier refers to the collection of content represented by the authority (and, where provided, the data path).

For example, a content `Uri` could be as simple as `content://sekrits`, which would refer to the collection of content held by whatever content provider was tied to the `sekrits` authority (e.g., `SecretsProvider`). Or, it could be as complex as `content://sekrits/card/pin/17`, which would refer to a piece of content (identified as `17`) managed by the `sekrits` content provider that is of the data type `card/pin`.

Next, Some Typing

Next, you need to come up with some MIME types corresponding with the content your content provider will provide.

Android uses both the content `Uri` and the MIME type as ways to identify content on the device. A collection content `Uri` – or, more accurately, the combination authority and data type path – should map to a pair of MIME types. One MIME type will represent the collection; the other will represent an instance. These map to the `Uri` patterns above for no-identifier and identifier, respectively. As you saw earlier in this book, you can fill in a MIME type into an `Intent` to route the `Intent` to the proper activity (e.g., `ACTION_PICK` on a collection MIME type to call up a selection activity to pick an instance out of that collection).

The collection MIME type should be of the form `vnd.X.cursor.dir/Y`, where `X` is the name of your firm, organization, or project, and `Y` is a dot-delimited type name. So, for example, you might use `vnd.tlagency.cursor.dir/sekrits.card.pin` as the MIME type for your collection of secrets.

The instance MIME type should be of the form `vnd.X.cursor.item/Y`, usually for the same values of `X` and `Y` as you used for the collection MIME type (though that is not strictly required).

Implementing the Database-Style API

Just as an activity and receiver are both Java classes, so is a content provider. So, the big step in creating a content provider is crafting its Java class, with a base class of ContentProvider.

In your subclass of ContentProvider, you are responsible for implementing five methods that, when combined, perform the services that a content provider is supposed to offer to activities wishing to create, read, update, or delete content via the database-style API.

Implement onCreate()

As with an activity, the main entry point to a content provider is onCreate(). Here, you can do whatever initialization you want. In particular, here is where you should lazy-initialize your data store. For example, if you plan on storing your data in such-and-so directory on an SD card, with an XML file serving as a "table of contents", you should check and see if that directory and XML file are there and, if not, create them so the rest of your content provider knows they are out there and available for use.

Similarly, if you have rewritten your content provider sufficiently to cause the data store to shift structure, you should check to see what structure you have now and adjust it if what you have is out of date.

Implement query()

As one might expect, the query() method is where your content provider gets details on a query some activity wants to perform. It is up to you to actually process said query.

The query method gets, as parameters:

- A Uri representing the collection or instance being queried

- A `String[]` representing the list of properties that should be returned

- A `String` representing what amounts to a SQL `WHERE` clause, constraining which instances should be considered for the query results

- A `String[]` representing values to "pour into" the `WHERE` clause, replacing any `?` found there

- A `String` representing what amounts to a SQL `ORDER BY` clause

You are responsible for interpreting these parameters however they make sense and returning a `Cursor` that can be used to iterate over and access the data.

As you can imagine, these parameters are aimed towards people using a SQLite database for storage. You are welcome to ignore some of these parameters (e.g., you elect not to try to roll your own SQL `WHERE` clause parser), but you need to document that fact so activities only attempt to query you by instance `Uri` and not using parameters you elect not to handle.

Implement insert()

Your insert() method will receive a `Uri` representing the collection and a `ContentValues` structure with the initial data for the new instance. You are responsible for creating the new instance, filling in the supplied data, and returning a `Uri` to the new instance.

Implement update()

Your `update()` method gets the `Uri` of the instance or collection to change, a `ContentValues` structure with the new values to apply, a String for a SQL `WHERE` clause, and a `String[]` with parameters to use to replace `?` found in the `WHERE` clause. Your responsibility is to identify the instance(s) to be modified (based on the `Uri` and `WHERE` clause), then replace those instances' current property values with the ones supplied.

This will be annoying, unless you are using SQLite for storage. Then, you can pretty much pass all the parameters you received to the `update()` call to the database, though the `update()` call will vary slightly depending on whether you are updating one instance or several.

Implement delete()

As with `update()`, `delete()` receives a `Uri` representing the instance or collection to work with and a WHERE clause and parameters. If the activity is deleting a single instance, the `Uri` should represent that instance and the WHERE clause may be null. But, the activity might be requesting to delete an open-ended set of instances, using the WHERE clause to constrain which ones to delete.

As with `update()`, though, this is simple if you are using SQLite for database storage (sense a theme?). You can let it handle the idiosyncrasies of parsing and applying the WHERE clause – all you have to do is call `delete()` on the database.

Implement getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

Update the Manifest

The glue tying the content provider implementation to the rest of your application resides in your `AndroidManifest.xml` file. Simply add a `<provider>` element as a child of the `<application>` element, such as:

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

  <application android:icon="@drawable/cw"
```

```
              android:label="@string/app_name">
   <provider android:authorities="com.commonsware.android.constants.Provider"
              android:name=".Provider" android:exported="false"/>
   <activity android:label="@string/app_name"
              android:name=".ConstantsBrowser">
     <intent-filter>
       <action android:name="android.intent.action.MAIN" />
       <category android:name="android.intent.category.LAUNCHER" />
     </intent-filter>
   </activity>
 </application>
 <supports-screens android:anyDensity="true"
                   android:largeScreens="true"
                   android:normalScreens="true"
                   android:smallScreens="true" />
</manifest>
```

The `android:name` property is the name of the content provider class, with a leading dot to indicate it is in the stock namespace for this application's classes (just like you use with activities).

The `android:authorities` property should be a semicolon-delimited list of the authority values supported by the content provider. Recall, from earlier in this chapter, that each content `Uri` is made up of a scheme, authority, data type path, and instance identifier. Each authority from each `CONTENT_URI` value should be included in the `android:authorities` list.

Now, when Android encounters a content `Uri`, it can sift through the providers registered through manifests to find a matching authority. That tells Android which application and class implements the content provider, and from there Android can bridge between the calling activity and the content provider being called.

Add Notify-On-Change Support

A feature that your content provider can to its clients is notify-on-change support. This means that your content provider will let clients know if the data for a given content `Uri` changes.

For example, suppose you have created a content provider that retrieves RSS and Atom feeds from the Internet based on the user's feed

subscriptions (via OPML, perhaps). The content provider offers read-only access to the contents of the feeds, with an eye towards several applications on the phone using those feeds versus everyone implementing their own feed poll-fetch-and-cache system. You have also implemented a service that will get updates to those feeds asynchronously, updating the underlying data store. Your content provider could alert applications using the feeds that such-and-so feed was updated, so applications using that specific feed can refresh and get the latest data.

On the content provider side, to do this, call `notifyChange()` on your `ContentResolver` instance (available in your content provider via `getContext().getContentResolver()`). This takes two parameters: the `Uri` of the piece of content that changed and the `ContentObserver` that initiated the change. In many cases, the latter will be `null`; a non-`null` value simply means that the observer that initiated the change will not be notified of its own changes.

On the content consumer side, an activity can call `registerContentObserver()` on its `ContentResolver` (via `getContentResolver()`). This ties a `ContentObserver` instance to a supplied `Uri` – the observer will be notified whenever `notifyChange()` is called for that specific `Uri`. When the consumer is done with the `Uri`, `unregisterContentObserver()` releases the connection.

Implementing the File System-Style API

If you want consumers of your `ContentProvider` to be able to call `openInputStream()` or `openOutputStream()` on a `Uri`, you will need to implement the `openFile()` method. This method is optional – if you are not supporting `openInputStream()` or `openOutputStream()`, you do not need to implement `openFile()` at all.

The `openFile()` method returns a curious object called a `ParcelFileDescriptor`. Given that, the `ContentResolver` can obtain the `InputStream` or `OutputStream` that was requested. There are various static methods on `ParcelFileDescriptor` to create instances of it, such as an `open()`

method that takes a `File` object as the first parameter. Note that this works for both files on external storage and files within your own project's app-local file storage (e.g., `getFilesDir()`).

Note that you are welcome to also implement `onCreate()`, if you wish to do some initialization when the content provider starts up. Also, you will have to provide do-nothing implementations of `query()`, `insert()`, `update()`, and `delete()`, as those methods are mandatory in `ContentProvider` subclasses, even if you do not plan to support them.

Issues with Content Providers

Content providers are not without their issues.

The biggest complaint seems to be the lack of an `onDestroy()` companion to the `onCreate()` method you can implement. Hence, if you open a database in `onCreate()`, you close it...never. Sometimes, you can alleviate this by initializing things on demand and releasing them immediately, such as opening a database as part of `insert()` and closing it within the same method. This does not always work, however – for example, you cannot close the database you query in `query()`, since the `Cursor` you return would become invalid.

The fact that `ContentProvider` is effectively a facade means that a consumer of a `ContentProvider` has no idea what to expect. It is up to documentation to explain what `Uri` values can be used, what columns can be returned, what query syntax is supported, and so on. And, the fact that it is a facade means that much of the richness of the SQLite interface is lost, such as `GROUP BY`. To top it off, the API supported by `ContentProvider` is rather limited – if what you want to share does not look like a database and does not look like a file, it may be difficult to force it into the `ContentProvider` API.

However, perhaps the biggest problem is that, by default, content providers are exported, meaning they can be accessed by other processes (third party applications or the Android OS). Sometimes this is desired. Sometimes, it is not. You need to set `android:exported` to be `false` on your manifest entry for

the content provider if you want to keep the provider private to your application. This is the inverse of all other components, which are private by default, unless they have an `<intent-filter>`.

Content Provider Implementation Patterns

The previous chapter focused on the concepts, classes, and methods behind content providers. This chapter more closely examines some implementations of content providers, organized into simple patterns.

The Single-Table Database-Backed Content Provider

The simplest database-backed content provider is one that only attempts to expose a single table's worth of data to consumers. The `CallLog` content provider works this way, for example.

Step #1: Create a Provider Class

We start off with a custom subclass of `ContentProvider`, named, cunningly enough, Provider. Here we need the database-style API methods: `query()`, `insert()`, `update()`, `delete()`, and `getType()`.

onCreate()

Here is the `onCreate()` method for `Provider`, from the `ContentProvider/ConstantsPlus` sample application:

343

```
@Override
public boolean onCreate() {
  db=(new DatabaseHelper(getContext())).getWritableDatabase();

  return((db == null) ? false : true);
}
```

While that does not seem all that special, the "magic" is in the private `DatabaseHelper` object, a fairly conventional `SQLiteOpenHelper` implementation:

```
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.hardware.SensorManager;

class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, 1);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    Cursor c=db.rawQuery("SELECT name FROM sqlite_master WHERE type='table' AND
name='constants'", null);

    try {
      if (c.getCount()==0) {
        db.execSQL("CREATE TABLE constants (_id INTEGER PRIMARY KEY
AUTOINCREMENT, title TEXT, value REAL);");

        ContentValues cv=new ContentValues();

        cv.put(Provider.Constants.TITLE, "Gravity, Death Star I");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Earth");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_EARTH);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Jupiter");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_JUPITER);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Mars");
```

```
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MARS);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Mercury");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MERCURY);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Moon");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MOON);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Neptune");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_NEPTUNE);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Pluto");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_PLUTO);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Saturn");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SATURN);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Sun");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SUN);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, The Island");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_THE_ISLAND);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Uranus");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_URANUS);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Venus");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_VENUS);
        db.insert("constants", Provider.Constants.TITLE, cv);
      }
    }
    finally {
      c.close();
    }
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    android.util.Log.w("Constants", "Upgrading database, which will destroy all
old data");
    db.execSQL("DROP TABLE IF EXISTS constants");
    onCreate(db);
  }
}
```

Note that we are opening the database here and never closing it. That is because there is no onDestroy() (or equivalent) method in a ContentProvider. While we might be tempted to open and close the database on every operation, that will not work, as we cannot close the database and still hand back a live Cursor from the database. Hence, we leave it open and assume that the core Android team is somehow making sure our database is not corrupted when Android shuts down the ContentProvider.

query()

For SQLite-backed storage providers like this one, the query() method implementation should be largely boilerplate. Use a SQLiteQueryBuilder to convert the various parameters into a single SQL statement, then use query() on the builder to actually invoke the query and give you a Cursor back. The Cursor is what your query() method then returns.

For example, here is query() from Provider:

```
@Override
 public Cursor query(Uri url, String[] projection, String selection,
                     String[] selectionArgs, String sort) {
   SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

   qb.setTables(TABLE);

   String orderBy;

   if (TextUtils.isEmpty(sort)) {
     orderBy=Constants.DEFAULT_SORT_ORDER;
   }
   else {
     orderBy=sort;
   }

   Cursor c=qb.query(db, projection, selection, selectionArgs,
                 null, null, orderBy);

   c.setNotificationUri(getContext().getContentResolver(), url);

   return(c);
 }
```

We create a SQLiteQueryBuilder and pour the query details into the builder, notably the name of the table that we query against and the sort order (substituting in a default sort if the caller did not request one). When done, we use the query() method on the builder to get a Cursor for the results. We also tell the resulting Cursor what Uri was used to create it, for use with the content observer system.

The query() implementation, like many of the other methods on Provider, delegates much of the Provider-specific information to private methods, such as:

- the name of the table (getTableName())
- the default sort order (getDefaultSortOrder())

insert()

Since this is a SQLite-backed content provider, once again, the implementation is mostly boilerplate: validate that all required values were supplied by the activity, merge your own notion of default values with the supplied data, and call insert() on the database to actually create the instance.

For example, here is insert() from Provider:

```java
@Override
public Uri insert(Uri url, ContentValues initialValues) {
  long rowID=db.insert(TABLE, Constants.TITLE, initialValues);

  if (rowID>0) {
    Uri uri=ContentUris.withAppendedId(Provider.Constants.CONTENT_URI, rowID);
    getContext().getContentResolver().notifyChange(uri, null);

    return(uri);
  }

  throw new SQLException("Failed to insert row into " + url);
}
```

The pattern is the same as before: use the provider particulars plus the data to be inserted to actually do the insertion.

update()

Here is `update()` from `Provider`:

```
@Override
public int update(Uri url, ContentValues values,
                  String where, String[] whereArgs) {
  int count=db.update(TABLE, values, where, whereArgs);

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

In this case, updates are always applied across the entire collection, though we could have a smarter implementation that supported updating a single instance via an instance `Uri`.

delete()

Similarly, here is `delete()` from `Provider`:

```
@Override
  public int delete(Uri url, String where, String[] whereArgs) {
    int count=db.delete(TABLE, where, whereArgs);

    getContext().getContentResolver().notifyChange(url, null);

    return(count);
  }
```

This is almost a clone of the `update()` implementation described above.

getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

For example, here is `getType()` from `Provider`:

```
@Override
public String getType(Uri url) {
  if (isCollectionUri(url)) {
    return("vnd.commonsware.cursor.dir/constant");
  }

  return("vnd.commonsware.cursor.item/constant");
}
```

Step #2: Supply a Uri

You may wish to add a public static member...somewhere, containing the Uri for each collection your content provider supports, for use by your own application code. Typically, this is a public static final Uri put on the content provider class itself:

```
public static final Uri CONTENT_URI
    =Uri.parse("content://com.commonsware.android.constants.Provider/constants")
;
```

You may wish to use the same namespace for the content Uri that you use for your Java classes, to reduce the chance of collision with others.

Bear in mind that if you intend for third parties to access your content provider, they will not have access to this public static data member, as your class is not in their project. Hence, you will need to publish the string representation of this Uri that they can hard-wire into their application.

Step #3: Declare the "Columns"

Remember those "columns" you referenced when you were using a content provider, in the previous chapter? Well, you may wish to publish public static values for those too for your own content provider.

Specifically, you may want a public static class implementing BaseColumns that contains your available column names, such as this example from Provider:

```
public static final class Constants implements BaseColumns {
  public static final Uri CONTENT_URI
      =Uri.parse("content://com.commonsware.android.constants.Provider/constants
");
  public static final String DEFAULT_SORT_ORDER="title";
  public static final String TITLE="title";
  public static final String VALUE="value";
}
```

Since we are are using SQLite as a data store, the values for the column
name constants should be the corresponding column name in the table, so
you can just pass the projection (array of columns) to SQLite on a `query()`,
or pass the `ContentValues` on an `insert()` or `update()`.

Note that nothing in here stipulates the types of the properties. They could
be strings, integers, or whatever. The biggest limitation is what a `Cursor` can
provide access to via its property getters. The fact that there is nothing in
code that enforces type safety means you should document the property
types well, so people attempting to use your content provider know what
they can expect.

Step #4: Update the Manifest

Finally, we need to add the provider to the `AndroidManifest.xml` file, by
adding a `<provider>` element as a child of the `<application>` element:

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

  <application android:icon="@drawable/cw"
              android:label="@string/app_name">
    <provider android:authorities="com.commonsware.android.constants.Provider"
              android:name=".Provider" android:exported="false"/>
    <activity android:label="@string/app_name"
              android:name=".ConstantsBrowser">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <supports-screens android:anyDensity="true"
                    android:largeScreens="true"
                    android:normalScreens="true"
```

```
                        android:smallScreens="true" />
</manifest>
```

Loaders

One perpetual problem in Android development is getting work to run outside the main application thread. Every millisecond we spend on the main application thread is a millisecond that our UI is frozen and unresponsive. Disk I/O, in particular, is a common source of such slowdowns, particularly since this is one place where the emulator typically out-performs actual devices. While disk operations rarely get to the level of causing an "application not responding" (ANR) dialog to appear, they can make a UI "janky".

Android 3.0 introduced a new framework to help deal with loading bulk data off of disk, called "loaders". The hope is that developers can use loaders to move database queries and similar operations into the background and off the main application thread. That being said, loaders themselves have issues, not the least of which is the fact that it is new to Android 3.0 and therefore presents some surmountable challenges for use in older Android devices.

This chapter will outline the programming pattern loaders are designed to solve, how to use loaders (both built-in and third-party ones) in your activities, and how to create your own loaders for scenarios not already covered.

Cursors: Issues with Management

Android has had the concept of "managed cursors" since Android 1.0, and perhaps before that. A managed `Cursor` is one that an `Activity`... well... manages. More specifically:

- When the activity is stopped, the managed `Cursor` is deactivated, freeing up all of the memory associated with the result set, and thereby reducing the activity's heap footprint while it is not in the foreground

- When the activity is restarted, the managed `Cursor` is requeried, to bring back the deactivated data, along the way incorporating any changes in that data that may have occurred while the activity was off-screen

- When the activity is destroyed, the managed `Cursor` is closed.

This is a delightful set of functionality. `Cursor` objects obtained from a `ContentProvider` via `managedQuery()` are automatically managed; a `Cursor` from `SQLiteDatabase` can be managed by `startManagingCursor()`.

The problem is that the `requery()` operation that is performed when the activity is restarted is executed on the main application thread. Many times, this is not a huge deal. However, given the nature of on-device flash and the Linux filesystem that many Android devices use (YAFFS2), it is entirely possible that what ordinarily is quick sometimes will not be. Also, you might be testing with small data sets, and your users might be working with bigger ones. As a result, the `requery()` may slow down your UI in ways that the user will notice.

Introducing the Loader Framework

The `Loader` framework was designed to solve three issues with the old managed `Cursor` implementation:

1. Arranging for a `requery()` (or the equivalent) to be performed on a background thread)

354

2. Arranging for the original query that populated the data in the first place to also be performed on a background thread, which the managed `Cursor` solution did not address at all

3. Supporting loading things other than a `Cursor`, in case you have data from other sources (e.g., XML files, JSON files, Web service calls) that might be able to take advantage of the same capabilities as you can get from a `Cursor` via the loaders

There are three major pieces to the `Loader` framework: `LoaderManager`, `LoaderCallbacks`, and the `Loader` itself.

LoaderManager

`LoaderManager` is your gateway to the `Loader` framework. You obtain one by calling `getLoaderManager()` (or `getSupportLoaderManager()`, as is described later in this chapter). Via the `LoaderManager` you can initialize a `Loader`, restart that `Loader` (e.g., if you have a different query to use for loading the data), etc.

LoaderCallbacks

Much of your interaction with the `Loader`, though, comes from your `LoaderCallbacks` object, such as your activity if that is where you elect to implement the `LoaderCallbacks` interface. Here, you will implement three "lifecycle" methods for consuming a `Loader`:

- `onCreateLoader()` is called when your activity requests that a `LoaderManager` initialize a `Loader`. Here, you will create the instance of the Loader itself, teaching it whatever it needs to know to go load your data

- `onLoadFinished()` is called when the `Loader` has actually loaded the data – you can take those results and pour them into your UI, such as calling `swapCursor()` on a `CursorAdapter` to supply the fresh `Cursor`'s worth of data

---

355

- onLoaderReset() is called when you should stop using the data supplied to you in the last onLoadFinished() call (e.g., the Cursor is going to be closed), so you can arrange to make that happen (e.g., call swapCursor(null) on a CursorAdapter)

When you implement the LoaderCallbacks interface, you will need to provide the data type of whatever it is that your Loader is loading (e.g., LoaderCallbacks<Cursor>). If you have several loaders returning different data types, you may wish to consider implementing LoaderCallbacks on multiple objects (e.g., instances of anonymous inner classes), so you can take advantage of the type safety offered by Java generics, rather than implementing LoaderCallbacks<Object> or something to that effect.

Loader

Then, of course, there is Loader itself.

Consumers of the Loader framework will use some concrete implementation of the abstract Loader class in their LoaderCallbacks onCreateLoader() method. API Level 11 introduced only one concrete implementation: CursorLoader, designed to perform queries on a ContentProvider, and described in a later section. This chapter will also outline the use of another concrete implementation, SQLiteCursorLoader, available via a JAR.

You are also welcome to create your own Loader implementations, if your data source is not a ContentResolver or SQLiteDatabase, and even if your data model is not a Cursor. You will typically extend AsyncTaskLoader, which arranges for the actual loading work to be done on a background thread. This chapter will delve into the implementation of SQLiteCursorLoader so you can see what the key methods are that you will need to implement.

Honeycomb… Or Not

`Loader` and its related classes were introduced in Android 3.0 (API Level 11). If your application is only going to be deployed on such devices, you can use loaders "naturally" via the standard implementation.

If, however, you are interested in using loaders but also want to support pre-Honeycomb devices, the Android Support package offers its own implementation of `Loader` and the other classes. However, to use it, you will need to work within four constraints:

1. You will need to add the Android Support package's JAR to your project (e.g., copy the JAR into your `libs/` directory and add it to your build path)

2. You will need to inherit from `FragmentActivity`, not the OS base `Activity` class or other refinements (e.g., `MapActivity`)

3. You will need to import the `support.v4` versions of various classes (e.g., `android.support.v4.app.LoaderManager` instead of `android.app.LoaderManager`)

4. You will need to get your `LoaderManager` by calling `getSupportLoaderManager()`, instead of `getLoaderManager()`, on your `FragmentActivity`

These limitations are the same ones that you will encounter when using fragments on older devices. Hence, while loaders and fragments are not really related, you may find yourself adopting both of them at the same time, as part of incorporating the Android Support package into your project.

Using CursorLoader

Let's start off by examining the simplest case: using a `CursorLoader` to asynchronously populate and update a `Cursor` retrieved from a `ContentProvider`. This is illustrated in the `Loaders/ConstantLoader` sample project, which is the same show-the-list-of-gravity-constants sample application that we examined previously, updated to use the `Loader`

framework. Note that this project does not use the Android Support package and therefore only supports API Level 11 and higher.

In `onCreate()`, rather than executing a `managedQuery()` to retrieve our constants, we ask our `LoaderManager` to initialize a loader, after setting up our `SimpleCursorAdapter` on a null `Cursor`:

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

  adapter=new SimpleCursorAdapter(this,
                      R.layout.row, null,
                      new String[] {Provider.Constants.TITLE,
                                    Provider.Constants.VALUE},
                      new int[] {R.id.title, R.id.value});

  setListAdapter(adapter);
  registerForContextMenu(getListView());
  getLoaderManager().initLoader(0, null, this);
}
```

Using a `null` `Cursor` means we will have an empty list at the outset, a problem we will rectify shortly.

The `initLoader()` call on `LoaderManager` (retrieved via `getLoaderManager()`) takes three parameters:

1. A locally-unique identifier for this loader

2. An optional `Bundle` of data to supply to the loader

3. A `LoaderCallbacks` implementation to use for the results from this loader (here set to be the activity itself, as it implements the `LoaderManager.LoaderCallbacks<Cursor>` interface)

The first time you call this for a given identifier, your `onCreateLoader()` method of the `LoaderCallbacks` will be called. Here, you need to initialize the `Loader` to use for this identifier. You are passed the identifier plus the `Bundle` (if any was supplied). In our case, we want to use a `CursorLoader`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(this, Provider.Constants.CONTENT_URI,
```

```
                        PROJECTION, null, null, null));
}
```

CursorLoader takes a Context plus all of the parameters you would ordinarily use with managedQuery(), such as the content provider Uri. Hence, converting existing code to use CursorLoader means converting your managedQuery() call into an invocation of the CursorLoader constructor inside of your onCreateLoader() method.

At this point, the CursorLoader will query the content provider, but do so on a background thread, so the main application thread is not tied up. When the Cursor has been retrieved, it is supplied to your onLoadFinished() method of your LoaderCallbacks:

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
  adapter.swapCursor(cursor);
}
```

Here, we call the new swapCursor() available on CursorAdapter, to replace the original null Cursor with the newly-loaded Cursor.

Your onLoadFinished() method will also be called whenever the data represented by your Uri changes. That is because the CursorLoader is registering a ContentObserver, so it will find out about data changes and will automatically requery the Cursor and supply you with the updated data.

Eventually, onLoaderReset() will be called. You are passed a Cursor object that you were supplied previously in onLoadFinished(). You need to make sure that you are no longer using that Cursor at this point – in our case, we swap null back into our CursorAdapter:

```
public void onLoaderReset(Loader<Cursor> loader) {
  adapter.swapCursor(null);
}
```

And that's pretty much it, at least for using CursorLoader. Of course, you need a content provider to make this work, and creating a content provider involves a bit of work.

Using SQLiteCursorLoader

What happens if you do not have a content provider? What if you are just using `SQLiteDatabase`, perhaps via `SQLiteOpenHelper`?

There is nothing in the Android SDK directly designed to apply the Loader pattern to `SQLiteDatabase`. However, the author of this book has created his own `SQLiteCursorLoader`, as part of the `LoaderEx` CommonsWare Android Component open source project. The project has a GitHub repository with its own code, plus a `demo/` sub-project illustrating its use. `LoaderEx` is licensed under the Apache Software License 2.0.

The nice thing about the Loader framework is that it isolates much of knowledge of what the specific `Loader` class is. Hence, using `SQLiteCursorLoader` is nearly identical to using `CursorLoader`. The primary difference is that you would create a `SQLiteCursorLoader` in your `onCreateLoader()` method, as shown in the following implementation from the `ConstantsBrowser` activity in the `LoaderEx` sample project:

```java
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new SQLiteCursorLoader(this,
                               db.getReadableDatabase(),
                               "SELECT _ID, title, value "+
                               "FROM constants ORDER BY title",
                               null));
}
```

Just as the constructor for `CursorLoader` takes the same parameters as does `managedQuery()` (plus a `Context`), the constructor for `SQLiteCursorLoader` takes the same parameters as does `rawQuery()` on a `SQLiteDatabase` (plus the `SQLiteDatabase` object itself and a `Context`).

The other difference is that there is no automatic means for `SQLiteCursorLoader` to know that the data in the database has changed. If you modify the data in the activity (e.g., insert or delete a row), you can call `restartLoader()` on your `LoaderManager` to have it execute the query again. This will supply the modified `Cursor` to your `onLoadFinished()` method, where you can once again slide it into the `CursorAdapter`.

There are two flavors of the `SQLiteCursorLoader` class. One, in `com.commonsware.cwac.loaderex`, is designed for use with API Level 11 and higher. The other, in `com.commonsware.cwac.loaderex.acl`, is designed for use with the Android Compatibility Library. If you use the JAR published in the downloads area of the GitHub repository, you can use either package. If you elect to add the project to yours as an Android library project, though, you will need to include the Android Support package's JAR in your build path, otherwise the `.acl` edition of `SQLiteCursorLoader` will fail to compile, even if you are not planning on using it. Alternatively, you could get rid of the `com.commonsware.cwac.loaderex.acl` package entirely to avoid this dependency.

Inside SQLiteCursorLoader

However, there may be times when you want to create your own custom `Loader`:

- You want to load a `Cursor`, but you want to have greater control over what background thread is used

- You want to load a `Cursor`, but not from a content provider and not from SQLite (e.g., a `MatrixCursor` you populate from other sources)

- You want to load something that is not a `Cursor`

In this section, we will examine the implementation of `SQLiteCursorLoader`, so you get an idea of what will be required to make another type of `Loader`.

AbstractCursorLoader

If you are creating your own `Loader` for reasons other than wanting to control the thread it loads on, the `AsyncTaskLoader` class supplied by Android (API Level 11 and Android Support package editions) is a likely class for you to extend. It handles the public `Loader` API and routes key logic to run on a background thread supplied by the ever-popular `AsyncTask`. By extending this class, you do not have to worry about the threading yourself, so you can focus more on your data-loading logic.

If you are creating a custom `Loader` that is loading a `Cursor`, just from an unusual source, you might consider extending `AbstractCursorLoader` instead. This class is in the `LoaderEx` project (API Level 11 and Android Support package editions). It consists mostly of the implementation of `CursorLoader` from the Android Support package, with the actual work to load the Cursor removed and replaced with a call to an abstract `buildCursor()` method. Since `AbstractCursorLoader` itself inherits from `AsyncTaskLoader`, the background thread is handled for you. We will examine an implementation of `AbstractCursorLoader` – the `SQLiteCursorLoader` we saw previously – in the next section. Here, though, we will look at the internals of `AbstractCursorLoader` itself, so you can see the sorts of things an `AsyncTaskLoader` needs to do.

loadInBackground()

The `loadInBackground()` method, as the name suggests, is where you load your data, and it is called on the background thread from the `AsyncTask`.

The key is to make sure that you really do load the data. Sometimes that is obvious, sometimes it is not. For example, when you query a content provider or database, the `Cursor` may just be a stub, delaying the actual work until the first time you try using the `Cursor`. With that in mind, the `AbstractCursorLoader` implementation of `loadInBackground()` not only calls the abstract `buildCursor()` method, but it ensures the `Cursor` data is really loaded by finding out how many rows are in it:

```
@Override
public Cursor loadInBackground() {
  Cursor cursor=buildCursor();

  if (cursor!=null) {
    // Ensure the cursor window is filled
    cursor.getCount();
  }

  return(cursor);
}
```

deliverResult()

This will be called, on the main application thread, when your `loadInBackground()` is complete and there are new results to be delivered to whoever is using this `Loader`. There are three main things you need to do here:

1. Check to see if the `Loader` has been reset by calling `isReset()`. If the `Loader` was reset while `loadInBackground()` was doing its work, we no longer need the results (passed in as a parameter to `deliverResult()`), so you should free it up. In our case, we close the `Cursor`.

2. Check to see if the `Loader` actually was started by calling `isStarted()`. If the `Loader` is started, chain to the superclass to actually hand the results back to the client.

3. Manage any caching of the results, including releasing any previously-cached results that will no longer be used. In our case, we close the last `Cursor` we delivered and cache the new one.

onStartLoading()

The `onStartLoading()` method is called, on the main application thread, when there has been a request to retrieve data from the `Loader`. If you have a cached result that is still valid, you can pass it to `deliverResult()` – if not, you should do something to start loading the data. In our case, if the data might have been changed or is not cached at all, we use `forceLoad()` to kick off the background thread and our `loadInBackground()` logic.

onCanceled()

It is possible to try to cancel an outstanding load request, by calling `cancelLoad()` on the `AsyncTaskLoader`. This, in turn, will try to `cancel()` the `AsyncTask`. That will eventually route to a call to `onCanceled()` in your implementation of `AsyncTaskLoader`. In the case of `AbstractCursorLoader`, we ensure that the `Cursor` we are supplied is closed – this might occur, for example, if we tried to cancel the load but the load completed first.

```
@Override
public void onCanceled(Cursor cursor) {
  if (cursor!=null && !cursor.isClosed()) {
    cursor.close();
  }
}
```

onStopLoading()

More commonly, though, a `Loader` may be told to `stopLoading()`. This keeps the last-delivered bit of data alive, but stops any future loads from occurring. Most of this is handled for us in `AsyncTaskLoader`, but our implementation is called with `onStopLoading()`. `AbstractCursorLoader` uses this to call `cancelLoad()` and stop a load in progress, should one be going on presently:

```
@Override
protected void onStopLoading() {
  // Attempt to cancel the current load task if possible.
  cancelLoad();
}
```

onReset()

It is possible to `reset()` a `Loader`. If you think of `stopLoading()` as the equivalent of a "pause", `reset()` is the equivalent of a "stop" – the `Loader` will no longer do anything until it is restarted. The `Loader` needs to retain enough of its state in order to start up again later on, but it does not need to hold onto anything else, including any previously-delivered results. `AsyncTaskLoader` handles much of this, but also calls `onReset()`, should you wish to hook into the event.

`AbstractCursorLoader` has an `onReset()` implementation that calls `onStopLoading()` first (to ensure that work gets done), then closes any `Cursor` that it might yet be holding onto:

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

   // Ensure the loader is stopped
```

364

```
    onStopLoading();

    if (lastCursor!=null && !lastCursor.isClosed()) {
      lastCursor.close();
    }

    lastCursor=null;
  }
```

SQLiteCursorLoader

All `SQLiteCursorLoader` needs to do is extend `AbstractCursorLoader` and implement `buildCursor()`:

```
@Override
protected Cursor buildCursor() {
  return(db.rawQuery(rawQuery, args));
}
```

Here, we just call `rawQuery()` on the `SQLiteDatabase`, using the parameters supplied to the `SQLiteCursorLoader` constructor:

```
public SQLiteCursorLoader(Context context, SQLiteDatabase db,
                          String rawQuery, String[] args) {
  super(context);
  this.db=db;
  this.rawQuery=rawQuery;
  this.args=args;
}
```

What Else Is Missing?

The Loader framework does an excellent job of handling queries in the background. What it does not do is help us with anything else that is supposed to be in the background, such as inserts, updates, deletes, or creating/upgrading the database. It is all too easy to put those on the main application thread and therefore possibly encounter issues. Moreover, since the thread(s) used by the `Loader` framework are an implementation detail, we cannot use those threads ourselves necessarily for the other CRUD operations.

To help in this area, the `LoaderEx` project has some simple `AsyncTask` subclasses that handle SQLite CRUD operations, to match up with the `AsyncTask` used by `AsyncTaskLoader` (from which `SQLiteCursorLoader` inherits).

Issues, Issues, Issues

Unfortunately, not all is rosy with the `Loader` framework.

There appears to be a bug in the Android Support package's implementation of the framework. If you use a `Loader` from a fragment that has `setRetainInstance()` set to `true`, you will not be able to use the `Loader` again after a configuration change, such as a screen rotation. This bug is not seen with the native API Level 11+ implementation of the framework.

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 a previous chapter, 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
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pick"
  android:layout_width="match_parent"
  android:layout_height="match_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.People.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.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:

```java
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 69. The PickDemo sample application, as initially launched
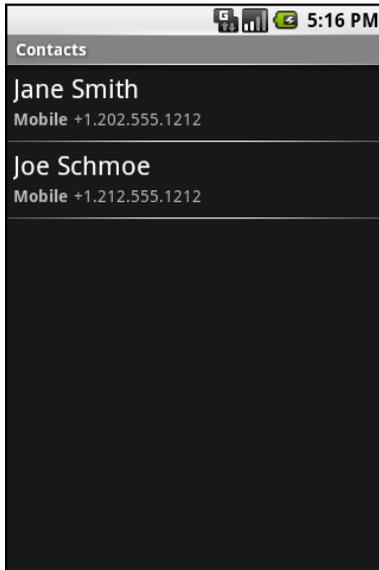
Figure 70. The same application, after clicking the "Gimme!" button, showing the list of available people
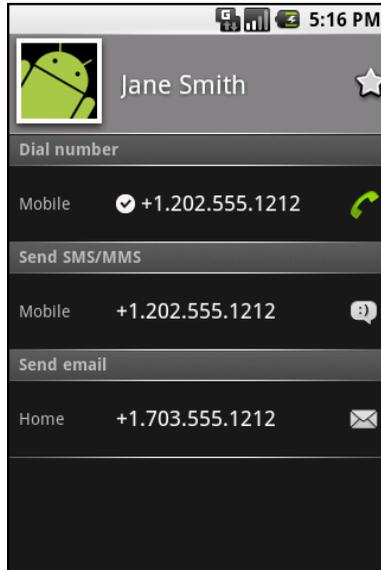
Figure 71. A view of a contact, launched by PickDemo after choosing one of the people from the pick list

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
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.contacts.spinners"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-sdk android:minSdkVersion="3"
            android:targetSdkVersion="6" />
  <supports-screens android:largeScreens="true"
                     android:normalScreens="true"
                     android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name=".ContactSpinners">
      <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
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_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:

```java
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;
```

```java
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[] {
```

379

```
                                   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:
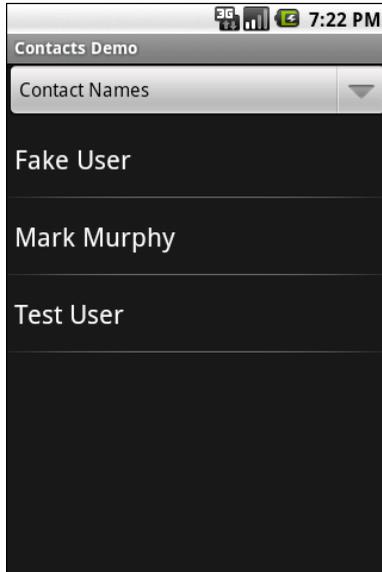
Figure 72. The ContactsDemo sample application, showing all contacts

Accessing Phone Numbers
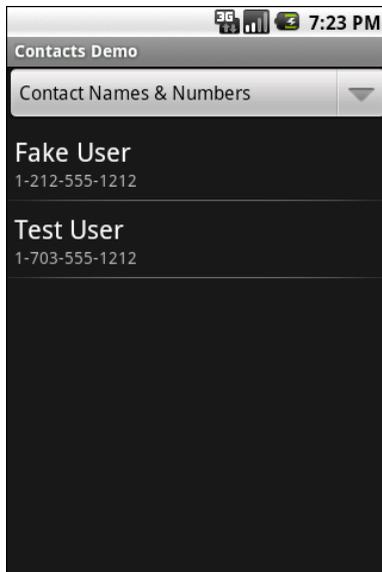
Figure 73. The ContactsDemo sample application, showing all contacts that have phone numbers

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`:
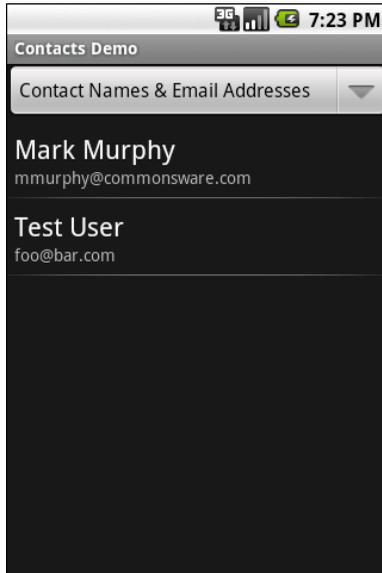
Figure 74. The ContactsDemo sample application, showing all contacts with email addresses

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
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_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:
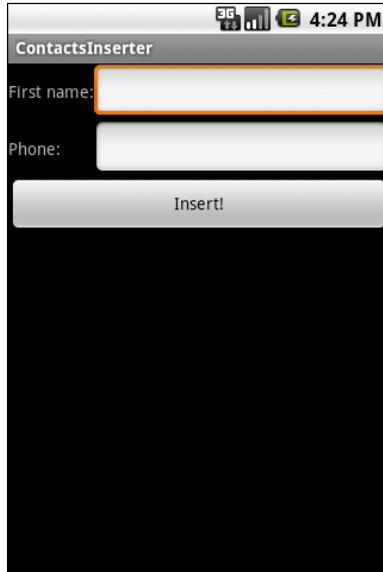
Figure 75. The ContactInserter sample application

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);
  }
```

384

```
  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:
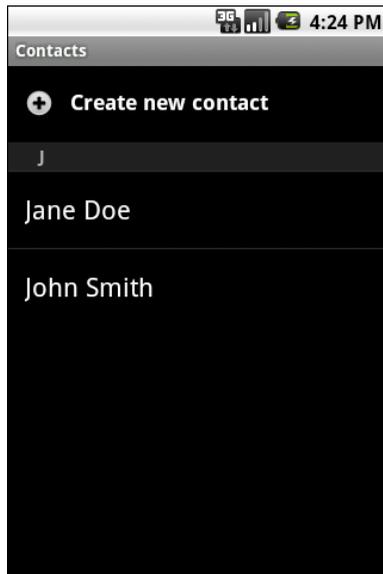
Figure 76. The add-or-edit-a-contact activity

...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 77. 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.

The CalendarContract Content Provider

The Android Open Source Project (AOSP) has had a Calendar application from its earliest days. This application originally was designed to sync with Google Calendar, later extended to other sync sources, such as Microsoft's Exchange. However, this application was not part of the Android SDK, so there was no way to access it from your Android application.

At least, no officially documented and supported way.

Many developers poked through the AOSP source code and found that the Calendar application had a `ContentProvider`. Moreover, this `ContentProvider` was exported (by default). So many developers used undocumented and unsupported means for accessing calendar information. This occasionally broke, as Google modified the Calendar app and changed these pseudo-external interfaces.

Android 4.0 added official SDK support for interacting with the Calendar application via its `ContentProvider`. As part of the SDK, these new interfaces should be fairly stable – if nothing else, they should be supported indefinitely, even if new and improved interfaces are added sometime in the future. So, if you want to tie into the user's calendars, you can. Bear in mind, though, that the new `CalendarContract ContentProvider` is not identical to the older undocumented providers, so if you are aiming to support pre-4.0 devices, you have some more work to do.

Of course, similar to the `ContactsContract ContentProvider`, the `CalendarContract ContentProvider` is severely lacking in documentation, and anything not documented is subject to change.

You Can't Be a Faker

While the Android 4.0 emulator has the `CalendarContract ContentProvider`, it will do you little good. While you can define a Google account on the emulator, the emulator lacks any ability to sync content with that account. Hence, you cannot see any events for your calendars in the Calendar app, and you cannot access any calendar data via `CalendarContract`.

Hence, at present, in order to test your use of `CalendarContract`, you will need to have hardware that runs Android 4.0 (or higher), with one or more accounts set up that have calendar data.

Do You Have Room on Your Calendar?

As a `ContentProvider`, `CalendarContract` is not significantly different from any other such provider that Android supplies or that you write yourself, in that there are `Uri` values representing collections of data, upon which you can query, insert, update, and delete as needed.

The Collections

The two main collections of data that you are likely to be interested in are `CalendarContract.Calendars` (the collection of all defined calendars) and `CalendarContract.Events` (the collection of all defined events across all calendars). Each of those has a `CONTENT_URI` static data member that you would use with `ContentResolver` or a `CursorLoader` to perform operations on those collections. An entry in `CalendarContract.Events` points back to its corresponding calendar via a `CALENDAR_ID` column that you can query upon; the remaining columns on `CalendarContract.Events` have names apparently designed to match with the iCalendar specification (e.g., `DTSTART` and `DTEND` for the start and end times of the event).

Three other collections may be of interest:

- `CalendarContract.Instances` has one entry per occurrence of an event, so recurring events get multiple rows

- `CalendarContract.Attendees` has information about each attendee of an event

- `CalendarContract.Reminders` has information about each reminder scheduled for an event (e.g., when to remind the user), for those events with associated reminders

Each of those ties back to its associated `CalendarContract.Events` row via an `EVENT_ID` column.

Calendar Permissions

There are two permissions for working with `CalendarContract`: `READ_CALENDAR` and `WRITE_CALENDAR`. As you might expect, querying `CalendarContract` requires the `READ_CALENDAR` permission; modifying `CalendarContract` data requires the `WRITE_CALENDAR` permission.

These permissions have existed since Android's earliest days, even in the SDK, as a side effect of the "meat cleaver" approach the core Android team employed to create the initial SDK. Hence, you can request these permissions in the manifest with any Android build target, without compiler errors. Of course, actually referring to `CalendarContract` will require a build target of API Level 14 or higher.

Querying for Events

For example, let's populate a `ListView` with the roster of all events the user has across all calendars, using a `CursorLoader`, showing the name of each event, the event's start date, and the event's end date. You can find this in the `Calendar/Query` sample project in the book's source code.

Our manifest has the `READ_CALENDARS` permission, as you would expect:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.cal.query"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-sdk android:minSdkVersion="14"/>
  <uses-permission android:name="android.permission.READ_CALENDAR"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:label="@string/app_name"
      android:name=".CalendarQueryActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>

</manifest>
```

We will use a simple `ListActivity` and so therefore do not need an activity layout. Our row layout (`res/layout/row.xml`) has three `TextView` widgets for the three pieces of data that we want to display:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/linearLayout1"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">

    <TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      android:layout_marginLeft="4dip"
      android:layout_marginRight="4dip"
      android:layout_weight="1"
      android:ellipsize="end"
      android:textSize="20sp"/>

    <LinearLayout
      android:id="@+id/linearLayout2"
      android:layout_width="wrap_content"
      android:layout_height="match_parent"
      android:layout_marginRight="4dip"
      android:orientation="vertical">
```

```
            <TextView
              android:id="@+id/dtstart"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_gravity="top"
              android:textSize="10sp"/>

            <TextView
              android:id="@+id/dtend"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_gravity="bottom"
              android:textSize="10sp"/>

        </LinearLayout>

    </LinearLayout>
```

In our activity (CalendarQueryActivity), in onCreate(), we set up a
SimpleCursorAdapter on a null Cursor at the outset and define the activity as
being the adapter's ViewBinder:

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

    adapter=
        new SimpleCursorAdapter(this, R.layout.row, null, ROW_COLUMNS,
                                ROW_IDS);
    adapter.setViewBinder(this);
    setListAdapter(adapter);

    getLoaderManager().initLoader(0, null, this);
  }
```

A ViewBinder is a way to tailor how Cursor data is poured into row widgets,
without subclassing the SimpleCursorAdapter. Implementing the
SimpleCursorAdapter.ViewBinder interface requires us to implement a
setViewValue() method, which will be called when the adapter wishes to
pour data from one column of a Cursor into one widget. We will examine
this method shortly.

The SimpleCursorAdapter will pour data from the ROW_COLUMNS in our Cursor
into the ROW_IDS widgets in our row layout:

```
private static final String[] ROW_COLUMNS=
    new String[] { CalendarContract.Events.TITLE,
        CalendarContract.Events.DTSTART,
        CalendarContract.Events.DTEND };
private static final int[] ROW_IDS=
    new int[] { R.id.title, R.id.dtstart, R.id.dtend };
```

Our `onCreate()` also initializes the `Loader` framework, triggering a call to `onCreateLoader()`, where we create and return a `CursorLoader`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(this, CalendarContract.Events.CONTENT_URI,
                          PROJECTION, null, null,
                          CalendarContract.Events.DTSTART));
}
```

We query on `CalendarContract.Events.CONTENT_URI`, asking for a certain set of columns indicated by our `PROJECTION` static data member:

```
private static final String[] PROJECTION=
    new String[] { CalendarContract.Events._ID,
        CalendarContract.Events.TITLE,
        CalendarContract.Events.DTSTART,
        CalendarContract.Events.DTEND };
```

The `ROW_COLUMNS` we map are a subset of the `PROJECTION`, skipping the `_ID` column that `SimpleCursorAdapter` needs but will not be displayed. Our query is also set up to sort by the start date (`CalendarContract.Events.DTSTART`).

When the query is complete, we pop it into the adapter in `onLoadFinished()` and remove it in `onLoaderReset()`:

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
}

public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
}
```

Our `setViewValue()` implementation then converts the `DTSTART` and `DTEND` values into formatted strings by way of `DateUtils` and the `formatDateTime()` method:

```
@Override
 public boolean setViewValue(View view, Cursor cursor, int columnIndex) {
    long time=0;
    String formattedTime=null;

    switch (columnIndex) {
      case 2:
      case 3:
        time=cursor.getLong(columnIndex);
        formattedTime=
            DateUtils.formatDateTime(this, time,
                                     DateUtils.FORMAT_ABBREV_RELATIVE);
        ((TextView)view).setText(formattedTime);
        break;

      default:
        return(false);
    }

    return(true);
 }
```

The setViewValue() method should return true for any columns it handles and false for columns it does not – skipped columns are handled by SimpleCursorAdapter itself.

If you run this on a device with available calendar data, you will get a list of those events:

Figure 78. The Calendar Query sample application, with some events redacted

Penciling In an Event

What is rarely documented in the Android SDK is what activities might exist that support the MIME types of a given `ContentProvider`. In part, that is because device manufacturers have the right to remove or replace many of the built-in applications.

The Calendar application is considered by Google to be a "core" application. Quoting the Android 2.3 version of the Compatibility Definition Document (CDD):

> *The Android upstream project defines a number of core applications, such as a phone dialer, calendar, contacts book, music player, and so on. Device implementers MAY replace these applications with alternative versions. However, any such alternative versions MUST honor the same Intent patterns pro-*

394

vided by the upstream project. For example, if a device contains an alternative music player, it must still honor the Intent pattern issued by third-party applications to pick a song.

Hence, in theory, so long as the CDD does not change and device manufacturers correctly honor it, those Intent patterns described by the Calendar application's manifest should be available across Android 4.0 devices. The Calendar application appears to support `ACTION_INSERT` and `ACTION_EDIT` for both the collection MIME type (`vnd.android.cursor.dir/event`) and the instance MIME type (`vnd.android.cursor.item/event`). Notably, there is no support for `ACTION_PICK` to pick a calendar or event, the way you can use `ACTION_PICK` to pick a contact.

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

397

- 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).
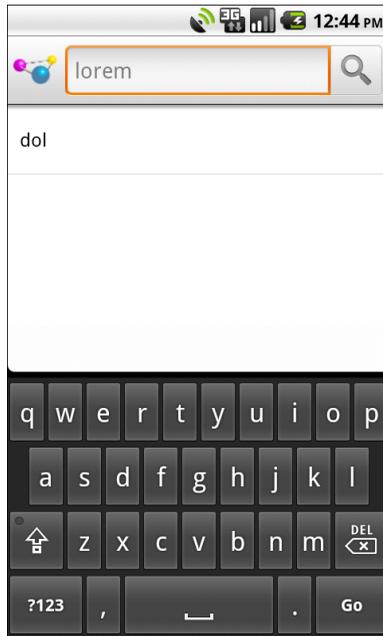
Figure 79. The Android local search popup, showing the IME and a previous search
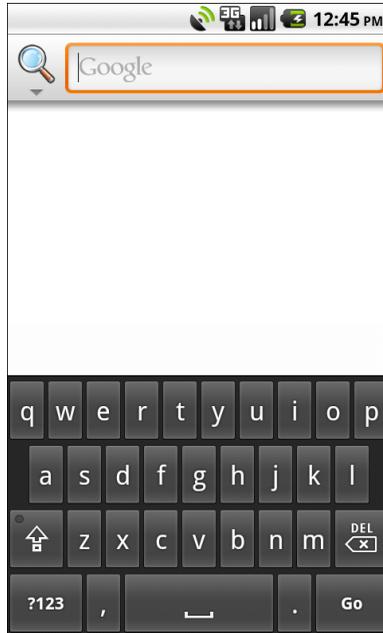
Figure 80. The Android global search popup

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="match_parent"
  android:layout_height="match_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_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;
  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(parent.getAdapter().getItem(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);

    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);
    }

    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 package="com.commonsware.android.search"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-sdk android:minSdkVersion="3"
            android:targetSdkVersion="6" />
```

403

```
  <supports-screens android:largeScreens="false"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
             android:label="Lorem Ipsum">
    <activity android:label="LoremDemo"
             android:name=".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:label="LoremSearch"
             android:launchMode="singleTop"
             android:name=".LoremSearch">
      <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:authorities="com.commonsware.android.search.LoremSuggestionProvider"
             android:name=".LoremSuggestionProvider" />
  </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.
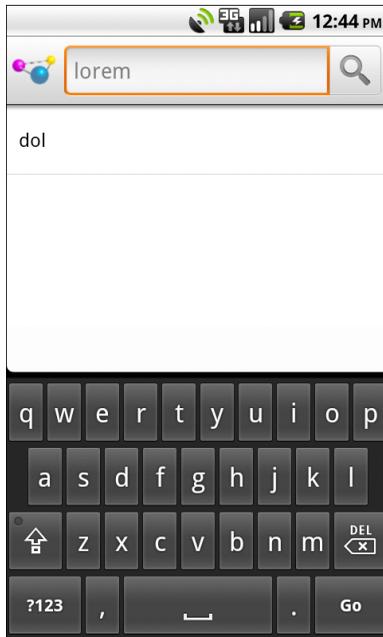
Figure 81. The Lorem sample application, showing the local search popup

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:
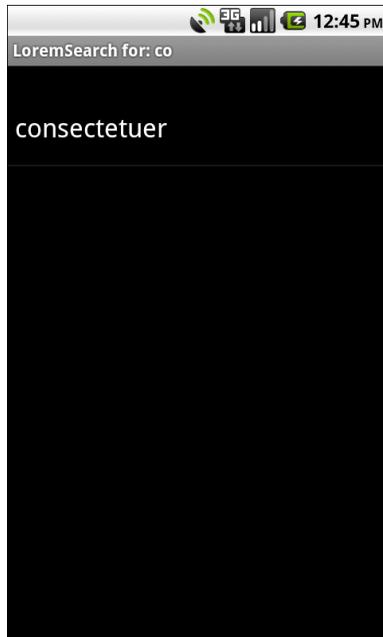
Figure 82. The results of searching for 'co' in the Lorem search sample

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:
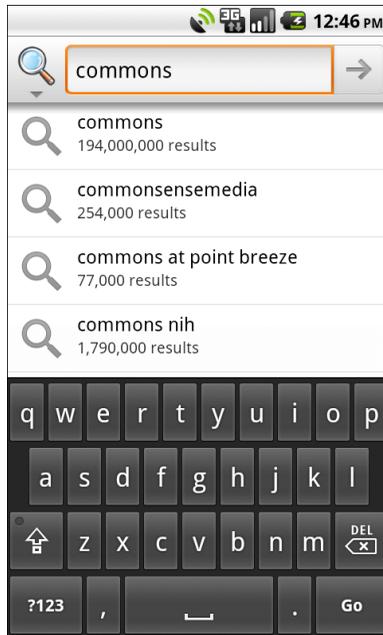
Figure 83. Search suggestions after typing some letters in global search

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:

```
<provider
android:authorities="com.commonsware.android.search.LoremSuggestionProvider"
          android:name=".LoremSuggestionProvider" />
```

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:
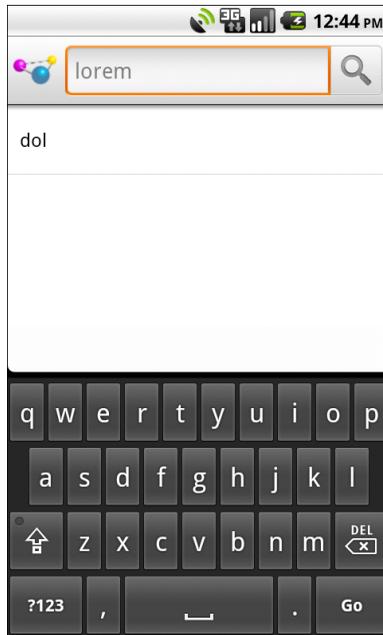
Figure 84. The Android local search popup, showing the IME and a previous search

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.

NOTE: there is some flaw in the Android 2.2 emulator that prevents this from working, though it works fine on Android 2.2 hardware.

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:
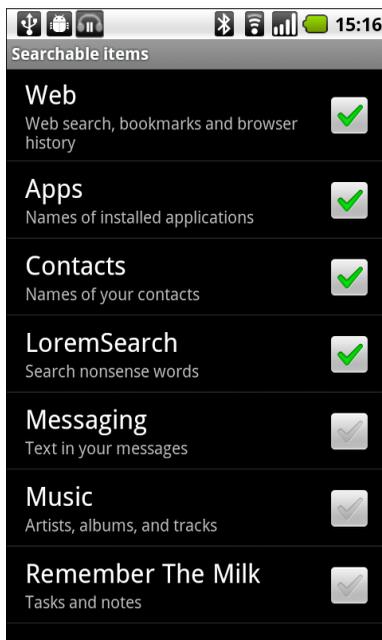
Figure 85. The Searchable Items settings screen

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:
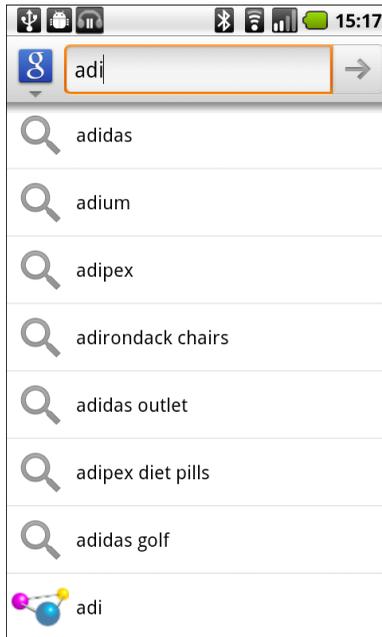
Figure 86. The Quick Search Box, showing application-supplied suggestions

On versions of Android prior to 2.2, to actually see your suggestions, the user also needs to click the arrow to "fold open" the actual suggestions:
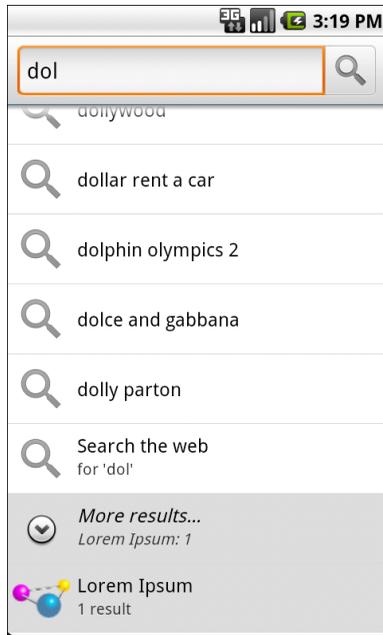
Figure 87. The Quick Search Box, showing another placeholder for application-supplied suggestions

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 88. 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.

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

419

ACTION_SEND support in your own application, so you can appear as an option when some other application allows its users to "send" things. Next, we look at how to get control when the user clicks on certain links in Web browsers, such as a for a certain MIME type or Web site. 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

421

- 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:

```java
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

425

- 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:
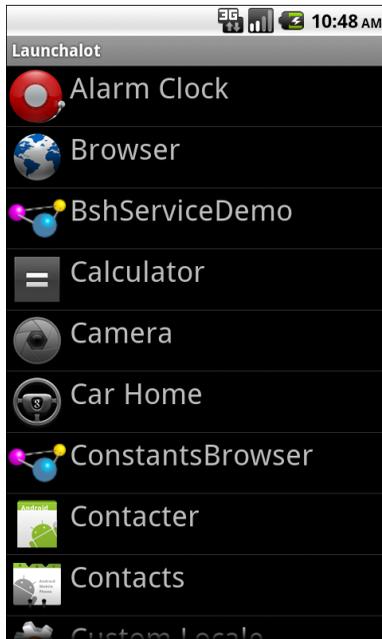
Figure 89. The Launchalot sample application

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

427

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

---

428

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

429

- 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/FauxSender`. This is a trivial implementation of an `ACTION_SEND` responder, in the form of an activity that just raises a Toast with the message to be "sent".

Our application will have two activities:

1. The main activity (`FauxSender`) that supports `ACTION_SEND`

2. A test activity that sends a message via `ACTION_SEND` and `createChooser()`, so our `FauxSender` will be an option

The Manifest

First, let's take a peek at the project's `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.fsender"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.INTERNET" />
```

```
  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
<application android:icon="@drawable/cw"
             android:label="@string/app_name">
  <activity android:label="@string/test_name"
            android:name="FauxSenderTest">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
  <activity android:label="@string/app_name"
            android:name="FauxSender"
            android:theme="@android:style/Theme.NoDisplay">
    <intent-filter android:label="@string/app_name">
      <action android:name="android.intent.action.SEND" />
      <data android:mimeType="text/plain" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
  </activity>
</application>
</manifest>
```

The test activity has the normal "please show me in the Launcher" configurations.

The other activity – `FauxSender` – 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 `FauxSender` to work with `ACTION_SEND` `Intent` objects of the type we aim to support.

The Main Activity

`FauxSender` is almost trivial:

```
package com.commonsware.android.fsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class FauxSender extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String msg=getIntent().getStringExtra(Intent.EXTRA_TEXT);

    if (TextUtils.isEmpty(msg)) {
      msg=getIntent().getStringExtra(Intent.EXTRA_SUBJECT);
    }

    if (TextUtils.isEmpty(msg)) {
      Toast
        .makeText(this, "No message supplied!", Toast.LENGTH_LONG)
        .show();
    }
    else {
      Toast
        .makeText(this, msg, 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 display a Toast with the text of the message.

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.

Obviously, a production-grade `ACTION_SEND` implementation would be more involved, including probably sending the message (using an `IntentService`) over the Internet via some protocol to some destination.

The Test Activity

Our test activity – `FauxSenderTest` – just fires off a pre-defined message using `ACTION_SEND`, using the `createChooser()` technique described earlier in this chapter:

```java
package com.commonsware.android.fsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.widget.Toast;

public class FauxSenderTest extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    sendIt("This is a test of FauxSender");

    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

Running `FauxSenderTest` will bring up a chooser to pick which means you want to use to send the message:

433
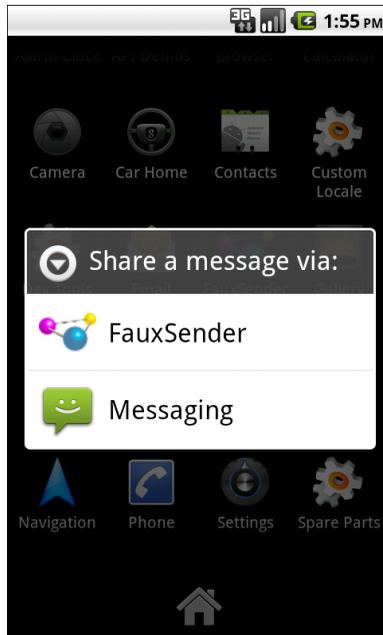
Figure 90. The ACTION_SEND chooser

If you choose FauxSender, you will get a Toast with the test message:
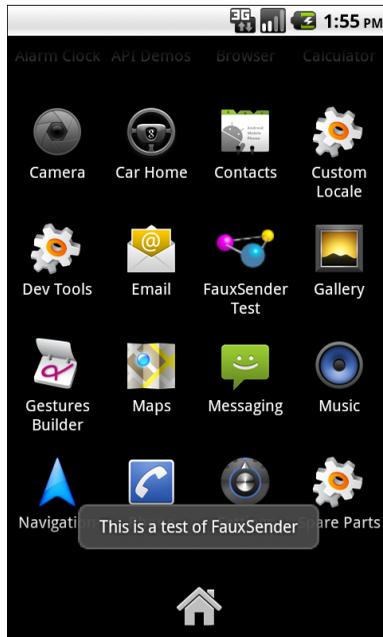
Figure 91. The result of sending via FauxSender

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
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.qsender"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <supports-screens android:largeScreens="true"
                     android:normalScreens="true"
                     android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="QuickSender">
      <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

436

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:
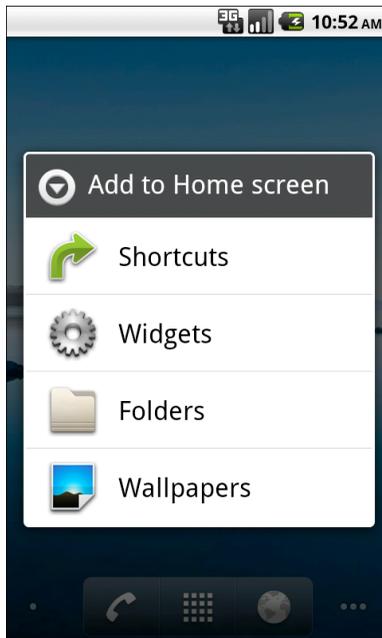
Figure 92. The home screen customization options list

Choose Shortcuts, and scroll down to find QuickSender in the list:

438
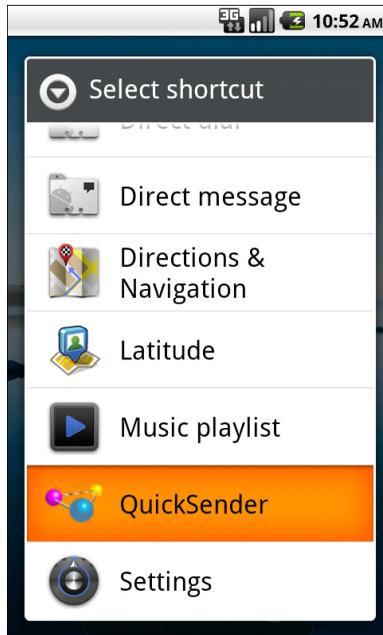
Figure 93. The available types of shortcuts

Click the `QuickSender` entry, which will bring up our activity with the form
to define what to send:
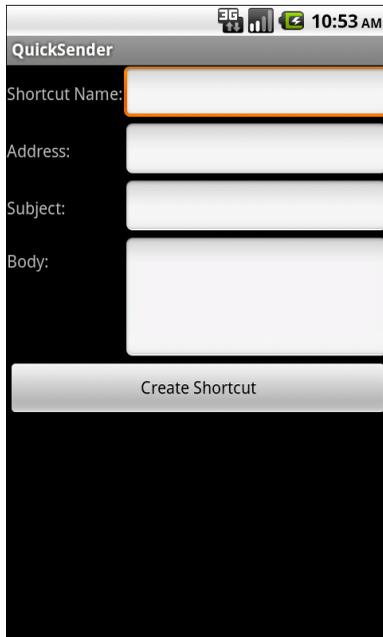
Figure 94. The QuickSender configuration activity

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:
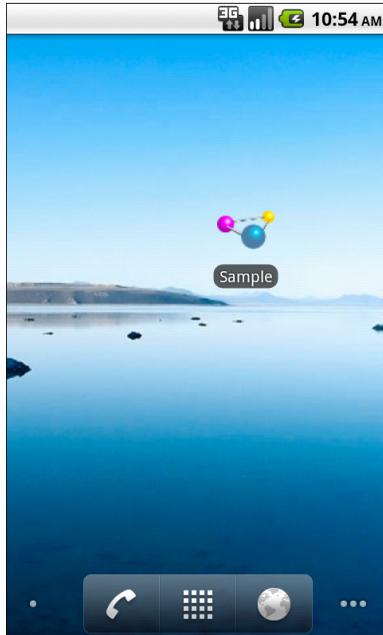
Figure 95. The QuickSender-defined shortcut, labeled Shortcut

If you launch that shortcut, and if there is more than one application on the device set up to handle ACTION_SEND, 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:
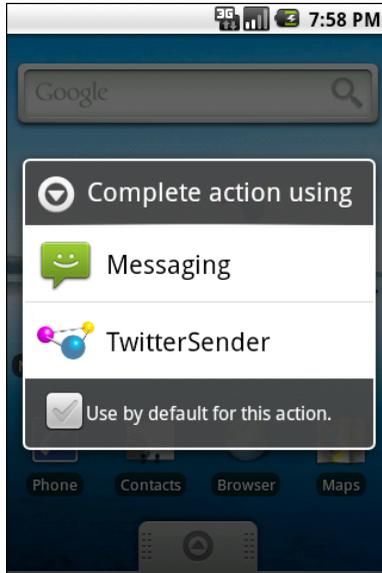
Figure 96. The ACTION_SEND request, as triggered by the shortcut

Depending on what you choose, of course, will dictate how the message actually gets sent.

Your Own Private URL

You may have noticed that Android supports a `market:` URL scheme. Web pages can use such URLs so that, if they are viewed on an Android device's browser, the user can be transported to an Android Market page, perhaps for a specific app or a list of apps for a publisher.

Fortunately, that mechanism is not limited to Android's code – you can get control for various other types of links as well. You do this by adding certain entries to an activity's `<intent-filter>` for an `ACTION_VIEW` Intent.

Manifest Modifications

First, any `<intent-filter>` designed to respond to browser links will need to have a `<category>` element with a name of

442

android.intent.category.BROWSABLE. Just as the LAUNCHER category indicates an activity that should get an icon in the launcher, the BROWSABLE category indicates an activity that wishes to respond to browser links.

You will then need to further refine which links you wish to respond to, via a ‹data› element. This lets you describe the URL and/or MIME type that you wish to respond to. For example, here is the AndroidManifest.xml file from the Introspection/URLHandler project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.urlhandler"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <supports-screens android:largeScreens="true"
                     android:normalScreens="true"
                     android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="URLHandler">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:mimeType="application/pdf" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="http" android:host="www.this-so-does-not-
exist.com" android:path="/something" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.android.MY_ACTION" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Here, we have four ‹intent-filter› elements for our one activity:

443

1. The first is a standard "put an icon for me in the launcher, please" filter, with the LAUNCHABLE category

2. The second claims that we handle PDF files (MIME type of application/pdf), and that we will respond to browser links (BROWSABLE category)

3. The third claims that we will handle any HTTP request (scheme of "http") for a certain Web site (host of "www.this-so-does-not-exist.com"), and that we will respond to browser links (BROWSABLE category)

4. The last is a custom action, for which we will generate a URL that Android will honor, and that we will respond to browser links (BROWSABLE category)

Note that the last one also requires the DEFAULT category in order to work.

Creating a Custom URL

Responding to MIME types makes complete sense…if we implement something designed to handle such a MIME type.

Responding to certain schemes, hosts, paths, or file extensions is certainly usable, but other than perhaps the file extension approach, it makes your application a bit fragile. If the site changes domain names (even a sub-domain) or reorganizes its site with different URL structures, your code will break.

If the goal is simply for you to be able to trigger your own application from your own Web pages, though, the safest approach is to use an intent: URL. These can be generated from an Intent object by calling toUri(Intent.URI_INTENT_SCHEME) on a properly-configured Intent, then calling toString() on the resulting Uri.

For example, the intent: URL for the fourth <intent-filter> from above is:

```
intent:#Intent;action=com.commonsware.android.MY_ACTION;end
```

This is not an official URL scheme, any more than `market:` is, but it works for Android devices. When the Android built-in Browser encounters this URL, it will create an `Intent` out of the URL-serialized form and call `startActivity()` on it, thereby starting your activity.

Reacting to the Link

Your activity can then examine the `Intent` that launched it to determine what to do. In particular, you will probably be interested in the `Uri` corresponding to the link – this is available via the `getData()` method. For example, here is the `URLHandler` activity for this sample project:

```java
package com.commonsware.android.urlhandler;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

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

    TextView uri=(TextView)findViewById(R.id.uri);

    if (Intent.ACTION_MAIN.equals(getIntent().getAction())) {
      String intentUri=(new Intent("com.commonsware.android.MY_ACTION"))
                          .toUri(Intent.URI_INTENT_SCHEME)
                          .toString();

      uri.setText(intentUri);
      Log.w("URLHandler", intentUri);
    }
    else {
      Uri data=getIntent().getData();

      if (data==null) {
        uri.setText("Got com.commonsware.android.MY_ACTION Intent");
      }
      else {
        uri.setText(getIntent().getData().toString());
      }
    }
```

```
  }

  public void visitSample(View v) {
    startActivity(new Intent(Intent.ACTION_VIEW,
                        Uri.parse("http://commonsware.com/sample")));
  }
}
```

This activity's layout has a `TextView` (uri) for showing a `Uri` and a `Button` to launch a page of links, found on the CommonsWare site (`http://commonsware.com/sample`). The `Button` is wired to call `visitSample()`, which just calls `startActivity()` on the aforementioned URL to display it in the Browser.

When the activity starts up, though, it first loads up the `TextView`. What goes in there depends on how the activity was launched:

- If it was launched via the launcher (e.g., the action is `MAIN`), then we display in the `TextView` the `intent:` URL shown in the previous section, generated from an `Intent` object designed to trigger our fourth `<intent-filter>`. This also gets dumped to LogCat, and is how the author got this URL in the first place to put on the sample Web page of links.

- If it was not launched via the launcher, it was launched from a Web link. If the `Uri` from the launching `Intent` is `null`, though, that means the activity was launched via the custom `intent:` URL (which only has an action string), so we put a message in the `TextView` to match.

- Otherwise, the `Uri` from the launching `Intent` will have something we can use to process the link request. For the PDF file, it will be the local path to the downloaded PDF, so we can open it. For the `www.this-so-does-not-exist.com` URL, it will be the URL itself, so we can process it our own way.

Note that for the PDF case, clicking the PDF link in the Browser will download the file in the background, with a Notification indicating when it is complete. Tapping on the entry in the notification drawer will then trigger the `URLHandler` activity.

Also, bear in mind that the device may have multiple handlers for some URLs. For example, a device with a real PDF viewer will give the user a choice of whether to launch the downloaded PDF in the real view or `URLHandler`.

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.

Tapjacking

On the whole, Android's security is fairly good for defending an app from another app. Between using Linux users and filesystems for protecting an application's files from other apps, to the use of custom permissions to control access to public interfaces, an application would seem to be relatively protected.

However, there is one attack vector that exists that, to date, has only been lightly addressed by the operating system: tapjacking. This chapter outlines what tapjacking is and what you can do about it to protect your app's users.

What is Tapjacking?

Tapjacking refers to another program intercepting and inspecting touch events that are delivered to your foreground activity (or related artifacts, such as the input method editor). At its worst, tapjackers could intercept passwords, PINs, and other private data.

The term "tapjacking" seems to have been coined by Lookout Mobile Security, in a blog post that originally demonstrated this issue.

You might be wondering how this is possible. There are a handful of approaches to implementing this. The Lookout blog post cited perhaps the least useful approach: making a transparent `Toast`. The

449

`Tapjacking/Jackalope` sample application will illustrate a far more troublesome implementation.

World War Z (Axis)

You may recall that there are three axes to consider with Android user interfaces. The X and Y axes are the ones you typically think about, as they control the horizontal and vertical positioning of widgets in an activity. The Z axis – effectively "coming out the screen towards the user's eyes" – can be used in applications for sophisticated techniques, such as the pop-up panel used in the maps samples presented earlier in this book.

Normally, you think of the Z axis within the scope of your activity and its widgets. However, there are ways to display "system alerts" – widgets that can float over top of any activity. A `Toast` is the one you are familiar with, most likely. A `Toast` displays something on the screen, yet touch events on the `Toast` itself will be passed through to the underlying activity. Lookout demonstrated that it is possible to create a fully-transparent `Toast`. However, the lifetime of a `Toast` is limited (3.5 seconds maximum), which would limit how long it can try to grab touch events.

However, any application holding the `SYSTEM_ALERT_WINDOW` permission can display their own "system alerts" with custom look and custom duration. By making one that is fully transparent and lives as long as possible, a tapjacker can obtain touch events for any application in the system, including lock screens, home screens, and any standard activity.

Enter the Jackalope

To demonstrate this, let's take a look at the Jackalope sample application. It consists of a tiny activity and a service, with the service doing most of the work.

The activity employs `Theme.NoDisplay`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      android:versionCode="1"
      android:versionName="1.0" package="com.commonsware.android.tj.jackalope">
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
    <application android:label="Jackalope">
        <activity android:name=".Jackalope"
                    android:theme="@android:style/Theme.NoDisplay">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".Tapjacker" />
    </application>
</manifest>
```

The activity then just starts up the service and finishes:

```
package com.commonsware.android.tj.jackalope;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class Jackalope extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    startService(new Intent(this, Tapjacker.class));
    finish();
  }
}
```

The visible effect is... nothing. Tapping the icon in the launcher appears to have no effect, but it does actually start up the tapjacker. You just cannot see it.

The `Tapjacker` service does its evil work in a handful of lines of code:

```
package com.commonsware.android.tj.jackalope;

import android.app.Service;
import android.content.Intent;
import android.graphics.PixelFormat;
import android.os.IBinder;
import android.util.Log;
import android.view.Gravity;
```

451

```
import android.view.MotionEvent;
import android.view.View;
import android.view.WindowManager;

public class Tapjacker extends Service implements View.OnTouchListener {
  private View v=null;
  private WindowManager mgr=null;

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

    v=new View(this);
    v.setOnTouchListener(this);
    mgr=(WindowManager)getSystemService(WINDOW_SERVICE);

    WindowManager.LayoutParams params
      =new WindowManager.LayoutParams(
        WindowManager.LayoutParams.FILL_PARENT,
        WindowManager.LayoutParams.FILL_PARENT,
        WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,
        WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH,
        PixelFormat.TRANSPARENT);

    params.gravity=Gravity.FILL_HORIZONTAL|Gravity.FILL_VERTICAL;
    mgr.addView(v, params);

    // stopSelf(); - uncomment for "component-less" operation
  }

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

  @Override
  public void onDestroy() {
    mgr.removeView(v);  // comment out for "component-less" operation

    super.onDestroy();
  }

  public boolean onTouch(View v, MotionEvent event) {
    Log.w("Tapjacker",
          String.valueOf(event.getX())+":"+String.valueOf(event.getY()));

    return(false);
  }
}
```

In onCreate(), we create an invisible View in Java code. Note that while you normally create a widget by passing in the Activity to the constructor, any Context will work, and so here we use the Tapjacker service itself.

Then, we access the `WindowManager` system service and add the invisible `View` to the system. To do this, we need to supply a `WindowManager.LayoutParams` object, much like you might use `LinearLayout.LayoutParams` or `RelativeLayout.LayoutParams` when putting a `View` inside of one of those containers. In this case, we:

- Say that the `View` is to fill the screen

- Indicates that the `View` is to be treated as a "system overlay" (`TYPE_SYSTEM_OVERLAY`), which will be at the top of the Z axis, floating above anything else (activities, dialogs, etc.)

- Indicates that we are to receive touch events that are beyond the `View` itself (`FLAG_WATCH_OUTSIDE_TOUCH`), such as on the system bar in API Level 11+ devices

We attach the `Tapjacker` service itself as the `OnTouchListener` to the `View`, and simply log all touch events to LogCat. In `onDestroy()`, we remove the system overlay `View`.

The result is that every screen tap results in an entry in LogCat – including data entry via the soft keyboard – even though the user is unaware that anything might be intercepting these events.

Note, though, that this does not intercept regular key events, including those from hardware keyboards. Also note that this does not magically give the malware author access to data entered before the tapjacker was set up. Hence, even if the tapjacker can sniff a password, if they do not know the account name, the user may still be safe.

Thinking Like a Malware Author

So, you have touch events. On the surface, this might not seem terribly useful, since the View cannot see what is being tapped upon.

However, a savvy malware author would identify what activity is in the foreground and log that information along with the tap details and the screen size, periodically dumping that information to some server. The

453

malware author can then scan the touch event dumps to see what interesting applications are showing up. With a minor investment – and possibly collaboration with other malware authors – the author can know what touch events correspond to what keys on various input method editors, including the stock keyboards used by a variety of devices. Loading a pirated version of the APK on an emulator can indicate which activity has the password, PIN, or other secure data. Then, it is merely a matter of identifying the touch events applied to that activity and matching them up with the soft keyboard to determine what the user has entered. Over time, the malware author can perhaps develop a script to help automate this conversion.

Hence, the on-device tapjacker does not have to be very sophisticated, other than trying to avoid detection by the user. All of the real work to leverage the intercepted touch events can be handled offline.

Detecting Potential Tapjackers

Tapjacking seems bad.

This begs the question: can we identify when a tapjacker is running? That would allow users and developers to "route around the damage", such as uninstalling the tapjacker application.

Unfortunately, this does not appear to be possible. There is no obvious way for an application – or the user – to determine if some other application has employed `WindowManager` to add a `TYPE_SYSTEM_OVERLAY` `View` to the screen. Even if there were, there is no way to determine if this `View` represents a tapjacker or somebody exploiting this capability for other, less nefarious ends.

All we can do is identify applications that might pose a problem.

Who Holds a Permission?

The biggest identifier of a possible tapjacker is the SYSTEM_ALERT_WINDOW permission. This is required to add a TYPE_SYSTEM_OVERLAY View to the screen. Relatively few applications request this, since built-in system alerts, like Toast, do not require the permission.

Also, a tapjacker probably needs the INTERNET permission, to deliver the results to the malware author. In principle, the tapjacker could be split into two applications, one with SYSTEM_ALERT_WINDOW and one with INTERNET. However, this adds to deployment complexity and therefore may be avoided by malware authors.

An end user can use programs like RL Permissions to examine the applications that have these permissions. A developer can use PackageManager to enumerate the installed applications and see which ones hold these permissions. We will examine some code for doing this later in this chapter.

Who is Running?

Of course, a tapjacker is only a threat if it is actually running in the background. Applications might use those two permissions just in the course of normal activity-centric operations, not with an everlasting service trying to maintain the interception View.

We can use ActivityManager to enumerate the running processes and what packages' code are in each. Any package that holds the permission combination from the previous section and is running in a process is a possible tapjacking threat. We will examine some code for doing this in the next section.

Note that it is important to examine running processes, not running services. For example, the Tapjacker service from earlier in this chapter could add the interception View and immediately exit. You can see this in action by adjusting the code as indicated in the comments in onCreate()

and `onDestroy()`. The interception `View` will remain intact (with the `Tapjacker` service object leaked) until the process is terminated. That process might be terminated quickly or slowly, depending on what all is going on with the device. A sophisticated malware author might try to run without a running service to increase stealthiness, at the cost of occasionally losing some data.

Combining the Two: TJDetect

To see these techniques in action, take a look at the `Tapjacking/TJDetect` sample project. This consists of a single `ListActivity`, whose list is populated with the applications that hold both `SYSTEM_ALERT_WINDOW` and `INTERNET` permissions and are presently running:

```java
package com.commonsware.android.tj.detect;

import android.app.ActivityManager;
import android.app.ListActivity;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import java.util.ArrayList;
import java.util.HashSet;

public class TJDetect extends ListActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ActivityManager am=(ActivityManager)getSystemService(ACTIVITY_SERVICE);
    HashSet<CharSequence> runningPackages=new HashSet<CharSequence>();

    for (ActivityManager.RunningAppProcessInfo proc :
        am.getRunningAppProcesses()) {
      for (String pkgName : proc.pkgList) {
        runningPackages.add(pkgName);
      }
    }

    PackageManager mgr=getPackageManager();
    ArrayList<CharSequence> scary=new ArrayList<CharSequence>();

    for (PackageInfo pkg :
        mgr.getInstalledPackages(PackageManager.GET_PERMISSIONS)) {
      if (PackageManager.PERMISSION_GRANTED==
            mgr.checkPermission(android.Manifest.permission.SYSTEM_ALERT_WINDOW,
```

```
                                pkg.packageName)) {
      if (PackageManager.PERMISSION_GRANTED==
          mgr.checkPermission(android.Manifest.permission.INTERNET,
                            pkg.packageName)) {
        if (runningPackages.contains(pkg.packageName)) {
          scary.add(mgr.getApplicationLabel(pkg.applicationInfo));
        }
      }
    }
  }

  setListAdapter(new ArrayAdapter(this,
                      android.R.layout.simple_list_item_1,
                      scary));
  }
}
```

To find the unique set of packages that are running across all processes, we iterate over the RunningAppProcessInfo objects returned by ActivityManager from a call to getRunningAppProcesses(). One public data member of RunningAppProcessInfo is a list of all the packages whose code runs in this process (pkgList). We use a simple HashSet to come up with the unique set of packages.

Then, we find all installed packages via a call to getInstalledPackages() on PackageManager. For each package, we use checkPermission() on PackageManager to see if the package in question holds a permission. Packages that pass those two tests are then checked against the HashSet of running packages, and those that are running are recorded in an ArrayList, later wrapped in an ArrayAdapter.

If you run TJDetect, it will not detect Jackalope, since Jackalope lacks the INTERNET permission. And, particularly on production hardware, it will detect several packages that may not be tapjackers at all, but rather are system applications installed in the firmware by the device manufacturer.

Defending Against Tapjackers

OK, so users and developers cannot reliably detect tapjackers. Surely there must be something in the OS that helps defend users and developers against tapjacking, right?

The answer is "yes", for a generous definition of the term "defend" and an equally generous definition of "users and developers".

Filtering Touch Events

The only "defense" directly provided by Android is to allow applications to filter out touch events that had been intercepted by a tapjacker, Toast, or any other form of system overlay or alert. Those touch events are simply dropped, never delivered to the underlying activity.

Implementing the Filter

The simplest way to implement the touch event filter is to add the `android:filterTouchesWhenObscured` attribute to a widget or container, setting it to true. The equivalent Java setter method on `View` is `setFilterTouchesWhenObscured()`.

For example, take a look at the `res/layout/main.xml` file in the `Tapjacking/RelativeSecure` sample project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:filterTouchesWhenObscured="true">
  <TextView android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="URL:"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"/>
  <EditText
    android:id="@id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/label"
    android:layout_alignParentTop="true"/>
  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
```

```
    android:layout_alignRight="@id/entry"
    android:text="OK" />
  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Cancel" />
</RelativeLayout>
```

Here, we have `android:filterTouchesWhenObscured="true"` on the `RelativeLayout` at the root of the layout resource. This property cascades to a container's children, and so if a tapjacker (or Toast or whatever) is above any of the widgets in the `RelativeLayout`, none of the touch events will be processed.

More fine-grained control can be achieved in custom widgets by overriding `onFilterTouchEventForSecurity()`, which gets control before the regular touch event methods. You can determine if a touch event had been intercepted by looking for the `FLAG_WINDOW_IS_OBSCURED` flag in the `MotionEvent` passed to `onFilterTouchEventForSecurity()`, and you can make the decision of how to handle this on an event-by-event basis.

The User Experience and the Hoped-For Security

Normally, the user will not see a difference when interacting with widgets that have this attribute set. However, if a tapjacker is intercepting these events, the user will not see any reaction from the widgets when they are tapped. For example, clicking a `Button` will have no visual effect (e.g., orange flash).

The hope is that users will realize that the UI is not responding to their touch events and therefore will not complete whatever it is they are doing. For example, they might not complete their PIN entry after realizing that the number pad supplied by the app is not responding to their taps.

For some users and some apps, this will be an effective defense. However, there will be some users who will remain oblivious until after completing the attempt to enter the private information.

The Flaws

The user can still use the soft keyboard to enter data into `EditText` widgets. While the soft keyboard will not automatically appear in portrait mode (since the `EditText` will not respond to the tap), if it has the focus, the user can long-press the MENU button to raise the soft keyboard and enter data that way.

Similarly, if the user is in landscape mode and gets the full-screen soft keyboard, since this is not the `EditText` widget defended by the touch event filtering, everything works normally – including interception by tapjacking. Developers could try to prevent this by adding `flagNoFullscreen` to the `android:imeOptions` attribute on the `EditText` in the layout XML, though this may not be honored by all soft keyboards. Developers could also try to prevent this by locking the activity into portrait mode (`android:screenOrientation="portrait"`), but this would be bad for users with side-slider keyboards, Google TV devices, etc.

And, most importantly, the tapjacking still happens. If users keep trying to enter their credentials despite the lack of UI feedback, they may eventually enter the whole thing and therefore become vulnerable to having that information used for ill ends.

Availability

Filtering touch events when the activity is obscured is supported in API Level 9 and above – in other words, Android 2.3 and newer. At the time of this writing, that leaves out 90% of active Android devices, based on the June 1, 2011 published edition of the platform versions data from Google.

By 2013, a majority of devices should run API Level 9 or higher, and by 2014 or 2015, only a tiny percentage will run older versions of Android. However, this does users and developers little good in the near term.

Detect-and-Warn

You can use the tapjacker detection logic illustrated earlier in this chapter. It is not particularly accurate, but you may feel it is worthwhile.

To minimize hassle for the user, your application should maintain a "whitelist" of approved packages. Any time you detect a package that is not on the approved list, you would raise an `AlertDialog` (or the equivalent) to let the user know of the potential tapjacker. If they elect to continue onward in your app, add the new package(s) to the whitelist, so you do not bother the user again for the same package.

Why Is This Being Discussed?

Some of you are by this time wondering why this book has a chapter on this subject.

Google's security team indicated to the author that `android:filterTouchesWhenObscured` is sufficient security. If so, developers need to realize when to use it, and for that, developers need to understand what tapjacking is to start with. The code to implement tapjacking is sufficiently trivial that "security by obscurity" of the code seems pointless.

It is eminently possible that `android:filterTouchesWhenObscured` is not sufficient security, despite Google's claim. In that case, developers may be able to help inform the public about the dangers of applications that request the `SYSTEM_ALERT_WINDOW` permission.

There are legitimate uses for tapjacking techniques. Some apps use this to provide a universal gesture interface, for example, to get control no matter

what application is presently in the foreground. Whether the value that such apps provide is worth the risks inherent in tapjacking is up for debate.

If you feel that tapjacking is a problem and that `android:filterTouchesWhenObscured` is inadequate, you may wish to let Google know when you have the opportunity to interact with Google engineers at conferences and similar events. If you come up with other ways to detect and/or prevent tapjacking, you may wish to distribute that knowledge, so other developers can learn from your discovery.

Working With SMS

SMS and Android is a frustrating experience.

While Android devices have reasonable SMS capability, much of that is out of the reach of developers following the official SDK. For various reasons – some defensible, others less so – there is no officially-supported way to create an SMS client, receive SMS data messages on specified ports, and so forth. Eventually, perhaps, this situation will be improved.

This chapter starts with the one thing you can do – send an SMS, either directly or by invoking the user's choice of SMS client. The chapter ends with a discussion of the various unsanctioned aspects of SMS that you may see other developers using, and why you may not want to follow suit.

Sending Out an SOS, Give or Take a Letter

While much of Android's SMS capabilities are not in the SDK, sending an SMS is. You have two major choices for doing this:

1. Invoke the user's choice of SMS client application, so they can compose a message, track its progress, and so forth using that tool

2. Send the SMS directly yourself, bypassing any existing client

Which of these is best for you depends on what your desired user experience is. If you are composing the message totally within your

463

application, you may want to just send it. However, as we will see, that comes at a price: an extra permission.

Sending Via the SMS Client

Sending an SMS via the user's choice of SMS client is very similar to the use of `ACTION_SEND` described in the previous chapter. You craft an appropriate `Intent`, then call `startActivity()` on that `Intent` to bring up an SMS client (or allow the user to choose between clients).

The `Intent` differs a bit from the `ACTION_SEND` example:

- You use `ACTION_SENDTO`, rather than `ACTION_SEND`
- Your `Uri` needs to begin with `smsto:`, followed by the mobile number you want to send the message to
- Your text message goes in an `sms_body` extra on the `Intent`

For example, here is a snippet of code from the `SMS/Sender` sample project:

```
Intent sms=new Intent(Intent.ACTION_SENDTO,
                      Uri.parse("smsto:"+c.getString(2)));

sms.putExtra("sms_body", msg.getText().toString());

startActivity(sms);
```

Here, our phone number is coming out of the third column of a `Cursor`, and the text message is coming from an `EditText` – more on how this works later in this section, when we review the `Sender` sample more closely.

Sending SMS Directly

If you wish to bypass the UI and send an SMS directly, you can do so through the `SmsManager` class, in the `android.telephony` package. Unlike most Android classes ending in `Manager`, you obtain an `SmsManager` via a static `getDefault()` method on the `SmsManager` class. You can then call `sendTextMessage()`, supplying:

- The phone number to send the text message to

- The "service center" address – leave this `null` unless you know what you are doing

- The actual text message

- A pair of `PendingIntent` objects to be executed when the SMS has been sent and delivered, respectively

If you are concerned that your message may be too long, use `divideMessage()` on `SmsManager` to take your message and split it into individual pieces. Then, you can use `sendMultipartTextMessage()` to send the entire `ArrayList` of message pieces.

For this to work, your application needs to hold the `SEND_SMS` permission, via a child element of your `<manifest>` element in your `AndroidManifest.xml` file.

For example, here is code from `Sender` that uses `SmsManager` to send the same message that the previous section sent via the user's choice of SMS client:

```
SmsManager
  .getDefault()
  .sendTextMessage(c.getString(2), null,
                   msg.getText().toString(),
                   null, null);
```

Inside the Sender Sample

The `Sender` example application is fairly straightforward, given the aforementioned techniques.

The manifest has both the `SEND_SMS` and `READ_CONTACTS` permissions, because we want to allow the user to pick a mobile phone number from their list of contacts, rather than type one in by hand:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:installLocation="preferExternal"
          android:versionCode="1"
```

```
            android:versionName="1.0"
            package="com.commonsware.android.sms.sender"
            xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.SEND_SMS" />
  <uses-sdk android:minSdkVersion="4"
            android:targetSdkVersion="8" />
  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="Sender">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

If you noticed the `android:installLocation` attribute in the root element, that is to allow this application to be installed onto external storage, such as an SD card – this will be covered in greater detail in an upcoming chapter.

The layout has a `Spinner` (for a drop-down of available mobile phone numbers), a pair of `RadioButton` widgets (to indicate which way to send the message), an `EditText` (for the text message), and a "Send" `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
>
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <RadioGroup android:id="@+id/means"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    >
      <RadioButton android:id="@+id/client"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:checked="true"
        android:text="Via Client" />
      <RadioButton android:id="@+id/direct"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Direct" />
  </RadioGroup>
  <EditText
    android:id="@+id/msg"
    android:layout_width="match_parent"
    android:layout_height="0px"
    android:layout_weight="1"
    android:singleLine="false"
    android:gravity="top|left"
  />
  <Button
    android:id="@+id/send"
    android:text="Send!"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="sendTheMessage"
  />
</LinearLayout>
```

Sender uses the same technique for obtaining mobile phone numbers from our contacts as is seen in the chapter on contacts. To support Android 1.x and Android 2.x devices, we implement an abstract class and two concrete implementations, one for the old API and one for the new. The abstract class then has a static method to get at an instance suitable for the device the code is running on:

```
package com.commonsware.android.sms.sender;

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

abstract class ContactsAdapterBridge {
  abstract SpinnerAdapter buildPhonesAdapter(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());
```

```
    }
}
```

The Android 2.x edition uses `ContactsContract` to find just the mobile numbers:

```
package com.commonsware.android.sms.sender;

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

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] { Contacts._ID,
                                       Contacts.DISPLAY_NAME,
                                       Phone.NUMBER
                                     };
    String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Phone.CONTENT_URI,
                            PROJECTION, Phone.TYPE+"=?",
                            ARGS, Contacts.DISPLAY_NAME);

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

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

...while the Android 1.x edition uses the older `Contacts` provider to find the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
```

468

```
import android.provider.Contacts;
import android.widget.SpinnerAdapter;
import android.widget.SimpleCursorAdapter;

class OldContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] { Contacts.Phones._ID,
                                       Contacts.Phones.NAME,
                                       Contacts.Phones.NUMBER
                                     };
    String[] ARGS={String.valueOf(Contacts.Phones.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Contacts.Phones.CONTENT_URI,
                            PROJECTION,
                            Contacts.Phones.TYPE+"=?", ARGS,
                            Contacts.Phones.NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.Phones.NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

For more details on how those providers work, please see the chapter on contacts.

The activity then loads up the Spinner with the appropriate list of contacts. When the user taps the Send button, the sendTheMessage() method is invoked (courtesy of the android:onClick attribute in the layout). That method looks at the radio buttons, sees which one is selected, and routes the text message accordingly:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.app.PendingIntent;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
```

```java
import android.os.Bundle;
import android.telephony.SmsManager;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.Spinner;

public class Sender extends Activity {
  Spinner contacts=null;
  RadioGroup means=null;
  EditText msg=null;

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

    contacts=(Spinner)findViewById(R.id.spinner);

    contacts.setAdapter(ContactsAdapterBridge
                          .INSTANCE
                          .buildPhonesAdapter(this));

    means=(RadioGroup)findViewById(R.id.means);
    msg=(EditText)findViewById(R.id.msg);
  }

  public void sendTheMessage(View v) {
    Cursor c=(Cursor)contacts.getSelectedItem();

    if (means.getCheckedRadioButtonId()==R.id.client) {
      Intent sms=new Intent(Intent.ACTION_SENDTO,
                          Uri.parse("smsto:"+c.getString(2)));

      sms.putExtra("sms_body", msg.getText().toString());

      startActivity(sms);
    }
    else {
      SmsManager
        .getDefault()
        .sendTextMessage(c.getString(2), null,
                      msg.getText().toString(),
                      null, null);
    }
  }
}
```

You Can't Get There From Here

The Android SDK is vast. It, however, does not cover everything. Many Android capabilities are not part of the SDK, though they can be accessed via indirect means. Doing so is dangerous, for two reasons:

1. Things not in the SDK and not part of the Compatibility Definition Document might well be replaced by device manufacturers. For example, even though the Android open source project has a stock SMS client, device manufacturers could replace it. Your application, therefore, may work on some devices but not others.

2. Things not in the SDK are subject to modification by the core Android team, and if you fail to react to those modifications (or cannot react, as the case may be), your application will fail on future versions of Android.

Developers are strongly encouraged to stick within the limits of the SDK. That being said, let us take a look at a pair of SMS capabilities that are beyond the SDK, still get used by developers, and what risks you will encounter by mirroring their techniques.

Receiving SMS

It is possible for an application to receive an incoming SMS message…if you are willing to listen on the undocumented `android.provider.Telephony.SMS_RECEIVED` broadcast `Intent`. That is sent by Android whenever an SMS arrives, and it is up to an application to implement a `BroadcastReceiver` to respond to that `Intent` and do something with the message. The Android open source project has such an application – Messaging – and device manufacturers can replace it with something else.

The `BroadcastReceiver` can then turn around and use the `SmsMessage` class, in the `android.telephony` package, to get at the message itself, through the following undocumented recipe:

- Given the received `Intent` (`intent`), call `intent.getExtras().get("pdus")` to get an `Object[]` representing the raw portions of the message

- For each of those "pdus" objects, call `SmsMessage.createFromPdu()` to convert the `Object` into an `SmsMessage` – though to make this work, you need to cast the `Object` to a `byte[]` as part of passing it to the `createFromPdu()` static method

The resulting `SmsMessage` object gets you access to the text of the message, the sending phone number, etc.

The `SMS_RECEIVED` broadcast `Intent` is broadcast a bit differently than most others in Android. It is an "ordered broadcast", meaning the `Intent` will be delivered to one `BroadcastReceiver` at a time. This has two impacts of note:

1. In your receiver's `<intent-filter>` element, you can have an `android:priority` attribute. Higher priority values get access to the broadcast `Intent` earlier than will lower priority values. The standard Messaging application has the default priority (undocumented, appears to be `0` or `1`), so you can arrange to get access to the SMS before the application does.

2. Your `BroadcastReceiver` can call `abortBroadcast()` on itself to prevent the `Intent` from being broadcast to other receivers of lower priority. In effect, this causes your receiver to consume the SMS – the Messaging application will not receive it.

However, just because the Messaging application has the default priority does not mean all SMS clients will, and so you cannot reliably intercept SMS messages this way. That, plus the undocumented nature of all of this, means that applications you write to receive SMS messages are likely to be fragile in production, breaking on various devices due to device manufacturer-installed apps, third-party apps, or changes to Android itself in the future.

Working With Existing Messages

When perusing the Internet, you will find various blog posts and such referring to the SMS inbox `ContentProvider`, represented by the `content://sms/inbox` Uri.

This `ContentProvider` is undocumented and is not part of the Android SDK, because it is not part of the Android OS.

Rather, this `ContentProvider` is used by the aforementioned Messaging application, for storing saved SMS messages. And, as noted, this application may or may not exist on any given Android device. If a device manufacturer replaces Messaging with their own application, there may be nothing on that device that responds to that `Uri`, or the schemas may be totally different. Plus, Android may well change or even remove this `ContentProvider` in future editions of Android.

For all those reasons, developers should not be relying upon this `ContentProvider`.

More on the Manifest

If you come to this book from *The Busy Coder's Guide to Android Development*, you will already have done a fair number of things with your project's `AndroidManifest.xml` file:

- Used it to define components, like activities, services, content providers, and manifest-registered broadcast receivers

- Used it to declare permissions your application requires, or possibly to define permissions that other applications need in order to integrate with your application

- Used it to define what SDK level, screen sizes, and other device capabilities your application requires

In this chapter, we continue looking at things the manifest offers you, starting with a discussion of controlling where your application gets installed on a device, and wrapping up with a bit of information about activity aliases.

Just Looking For Some Elbow Room

On October 22, 2008, the HTC Dream was released, under the moniker of "T-Mobile G1", as the first production Android device.

Complaints about the lack of available storage space for applications probably started on October 23rd.

475

The Dream, while a solid first Android device, offered only 70MB of on-board flash for application storage. This storage had to include:

- The Android application (APK) file

- Any local files or databases the application created, particularly those deemed unsafe to put on the SD card (e.g., privacy)

- Extra copies of some portions of the APK file, such as the compiled Dalvik bytecode, which get unpacked on installation for speed of access

It would not take long for a user to fill up 70MB of space, then have to start removing some applications to be able to try others.

Users and developers alike could not quite understand why the Dream had so little space compared to the available iPhone models, and they begged to at least allow applications to install to the SD card, where there would be more room. This, however, was not easy to implement in a secure fashion, and it took until Android 2.2 for the feature to become officially available.

Now that it is available, though, let's see how to use it.

Configuring Your App to Reside on External Storage

Indicating to Android that your application can reside on the SD card is easy...and necessary, if you want the feature. If you do not tell Android this is allowed, Android will *not* install your application to the SD card, nor allow the user to move the application to the SD card. Hence, once Android 2.2 becomes available on a substantial number of devices, you will be pressured by your user base to enable this feature, more so if your application is large.

All you need to do is add an `android:installLocation` attribute to the root `<manifest>` element of your `AndroidManifest.xml` file. There are three possible values for this attribute:

1. `internalOnly`, the default, meaning that the application cannot be installed to the SD card

2. `preferExternal`, meaning the application would like to be installed on the SD card

3. `auto`, meaning the application can be installed in either location

If you use `preferExternal`, then your application will be initially installed on the SD card in most cases. Android reserves the right to still install your application on internal storage in cases where that makes too much sense, such as there not being an SD card installed at the time.

If you use `auto`, then Android will make the decision as to the installation location, based on a variety of factors. In effect, this means that `auto` and `preferExternal` are functionally very similar – all you are doing with `preferExternal` is giving Android a hint as to your desired installation destination.

Because Android decides where your application is initially installed, and because the user has the option to move your application between the SD card and on-board flash, you cannot assume any given installation spot. The exception is if you choose `internalOnly`, in which case Android will honor your request, at the potential cost of not allowing the installation at all if there is no more room in on-board flash.

For example, here is the manifest from the `SMS/Sender` application, profiled in another chapter, showing the use of `preferExternal`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:installLocation="preferExternal"
          android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.sms.sender"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.SEND_SMS" />
  <uses-sdk android:minSdkVersion="4"
            android:targetSdkVersion="8" />
  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
```

```
                      android:smallScreens="false" />
  <application android:icon="@drawable/cw"
              android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="Sender">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Since this feature only became available in Android 2.2, to support older versions of Android, just have your build tools target API level 8 (e.g., `target=android-8` in `default.properties` for those of you building via Ant) while having your `minSdkVersion` attribute in the manifest state the lowest Android version your application supports overall. Older versions of Android will ignore the `android:installLocation` attribute. So, for example, in the above manifest, the `Sender` application supports API level 4 and above (Android 1.6 and newer), but still can use `android:installLocation="preferExternal"`, because the build tools are targeting API level 8.

What the User Sees

For an application that wound up on the SD card, courtesy of your choice of `preferExternal` or `auto`, the user will have an option to move it to the phone's internal storage. This can be done by choosing the application in the Manage Applications list in the Settings application, then clicking the "Move to phone" button:
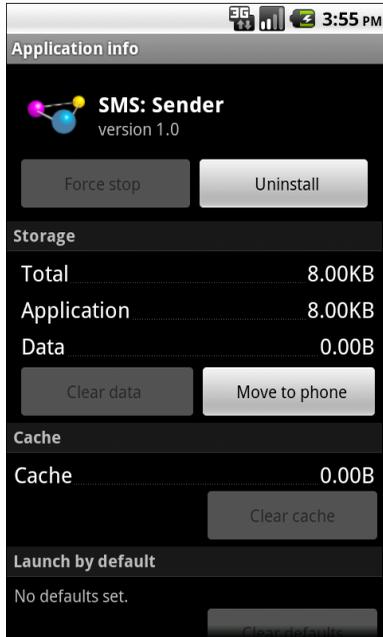
Figure 97. An application, installed on the SD card

Conversely, if your application is installed in on-board flash, and it is movable to external storage, they will be given that option:

Figure 98. An application, installed on the on-board flash but movable to the SD card

What the Pirate Sees

Ideally, the pirate sees nothing at all.

One of the major concerns with installing applications to the SD card is that the SD card is usually formatted FAT32 (`vfat`), offering no protection from prying eyes. The concern was that pirates could then just pluck the APK file off the SD card and distribute it, even for paid apps from the Android Market.

Apparently, they solved this problem.

To quote the Android developer documentation:

> *The unique container in which your application is stored is encrypted with a randomly generated key that can be de-*

480

> *crypted only by the device that originally installed it. Thus, an application installed on an SD card works for only one device.*

Moreover, this "unique container" is not normally mounted when the user mounts external storage on their host machine. The user mounts `/mnt/sdcard`; the "unique container" is `/mnt/asec`.

What Your App Sees...When the Card is Removed

So far, this has all seemed great for users and developers. Developers need to add a single attribute to the manifest, and Android 2.2 users gain the flexibility of where the app gets stored.

Alas, there is a problem, and it is a big one: either the host PC or the device can have access to the SD card, but not both. As a result, if the user makes the SD card available to the host PC, by plugging in the USB cable and mounting the SD card as a drive via a `Notification` or other means, that SD card becomes unavailable for running applications.

So, what happens?

- First, your application is terminated forcibly, as if your process was being closed due to low memory. Notably, your activities and services will not be called with `onDestroy()`, and instance state saved via `onSaveInstanceState()` is lost.

- Second, your application is unhooked from the system. Users will not see your application in the launcher, your `AlarmManager` alarms will be canceled, and so on.

- When the user makes the SD card available to the phone again, your application will be hooked back into the system and will be once again available to the user (for example, your icon will reappear in the launcher)

The upshot: if your application is simply a collection of activities, otherwise not terribly connected to Android, the impact on your application is no different than if the user reboots the phone, kills your process via a so-called "task killer" application, etc. If, however, you are doing more than that, the impacts may be more dramatic.

Perhaps the most dramatic impact, from a user's standpoint, will be if your application implements app widgets. If the user has your app widget on her home screen, that app widget will be removed when the SD card becomes unavailable to the phone. Worse, your app widget cannot be re-added to the home screen until the phone is rebooted (a limitation that hopefully will be lifted sometime after Android 2.2).

The user is warned about this happening, at least in general:
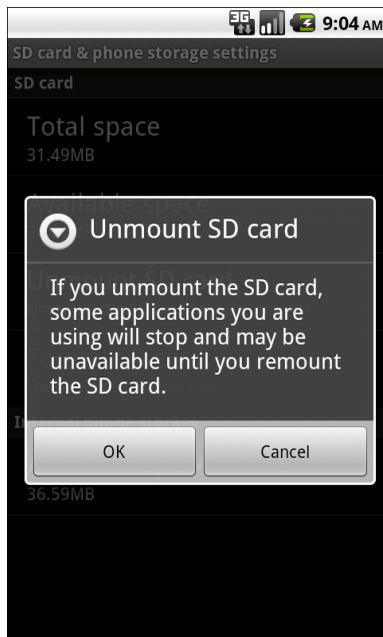
Figure 99. Warning when unmounting the SD card

Two broadcast Intents are sent out related to this:

1. ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE, when the SD card (and applications installed upon it) become unavailable

2. ACTION_EXTERNAL_APPLICATIONS_AVAILABLE, when the SD card and its applications return to normal

Note that the documentation is unclear as to whether your own application, that had been on the SD card, can receive ACTION_EXTERNAL_APPLICATIONS_AVAILABLE once the SD card is back in action. There is an outstanding issue on this topic in the Android issue tracker.

Also note that all of these problems hold true for longer if the user physically removes the SD card from the device. If, for example, they replace the card with a different one – such as one with more space – your application will be largely lost. They will see a note in their applications list for your application, but the icon will indicate it is on an SD card, and the only thing they can do is uninstall it:
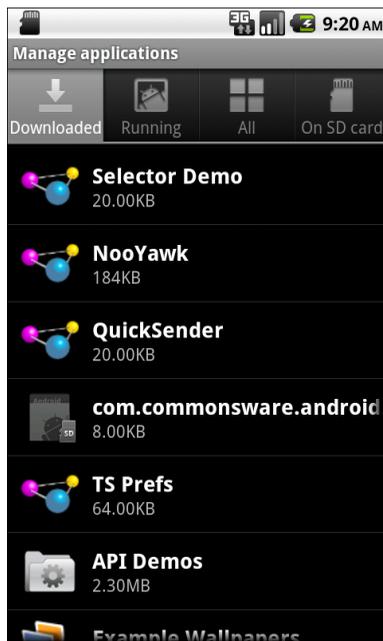
Figure 100. The Manage Applications list, with an application shown from a removed SD card

Choosing Whether to Support External Storage

Given the huge problem from the previous section, the question of whether or not your application should support external storage is far from clear.

As the Android developer documentation states:

> *Large games are more commonly the types of applications that should allow installation on external storage, because games don't typically provide additional services when inactive. When external storage becomes unavailable and a game process is killed, there should be no visible effect when the storage becomes available again and the user restarts the game (assuming that the game properly saved its state during the normal Activity lifecycle).*

Conversely, if your application implements any of the following features, it may be best to not support external storage:

- Polling of Web services or other Internet resources via a scheduled alarm
- Account managers and their corresponding sync adapters, for custom sources of contact data
- App widgets, as noted in the previous section
- Device administration extensions
- Live folders
- Custom soft keyboards ("input method engines")
- Live wallpapers
- Custom search providers

Using an Alias

As was mentioned in the chapter on integration, you can use the `PackageManager` class to enable and disable components in your application. This works at the component level, meaning you can enable and disable activities, services, content providers, and broadcast receivers. It does not support enabling or disabling individual `<intent-filter>` stanzas from a given component, though.

Why might you want to do this?

- Perhaps you have an activity you want to be available for use, but not necessarily available in the launcher, depending on user configuration or unlocking "pro" features or something

- Perhaps you want to add browser support for certain MIME types, but only if other third-party applications are not already installed on the device

While you cannot control individual `<intent-filter>` stanzas directly, you can have a similar effect via an activity alias.

An activity alias is another manifest element – `<activity-alias>` – that provides an alternative set of filters or other component settings for an already-defined activity. For example, here is the `AndroidManifest.xml` file from the `Manifest/Alias` project:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.alias"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <supports-screens android:largeScreens="true"
                     android:normalScreens="true"
                     android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
```

```
      </intent-filter>
    </activity>
    <activity-alias android:label="@string/app_name2"
                    android:name="ThisIsTheAlias"
                    android:targetActivity="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity-alias>
  </application>
</manifest>
```

Here, we have one `<activity>` element, with an `<intent-filter>` to put the activity in the launcher. We also have an `<activity-alias>` element...which puts a second icon in the launcher for the same activity implementation.

An activity alias can be enabled and disabled independently of its underlying activity. Hence, you can have one activity class have several independent sets of intent filters and can choose which of those sets are enabled at any point in time.

For testing purposes, you can also enable and disable these from the command line. Use the `adb shell pm disable` command to disable a component:

```
adb shell pm disable
com.commonsware.android.alias/com.commonsware.android.alias.ThisIsTheAlias
```

...and the corresponding `adb shell pm enable` command to enable a component:

```
adb shell pm enable
com.commonsware.android.alias/com.commonsware.android.alias.ThisIsTheAlias
```

In each case, you supply the package of the application (`com.commonsware.android.alias`) and the class of the component to enable or disable (`com.commonsware.android.alias.ThisIsTheAlias`), separated by a slash.

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

487

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.

Push Notifications with C2DM

C2DM – short for "cloud to device messaging" – is Google's new framework for asynchronously delivering notifications from the Internet ("cloud") to Android devices. Rather than the device waking up and polling on a regular basis at the behest of your app, your app can register for notifications and then wait for them to arrive. C2DM is engineered with power savings in mind, aiming to minimize the length of time 3G radios are exchanging data.

The proper use of C2DM means better battery life for your users. It can also reduce the amount of time your code runs, which helps you stay out of sight of users looking to pounce on background tasks and eradicate them with task killers.

C2DM is beta technology as of the time of this writing. It is available on an invitation basis only and is likely to undergo some revisions before it is widely available. Hence, more so than most chapters in this book, please bear in mind that the material presented here may well have changed by the time you get around to using C2DM. Also,note that C2DM is only available on Android 2.2 and higher. And, if you intend to use the Android 2.2 emulator, you will need to register a Google account on the emulator, via the Settings application.

Pieces of Push

C2DM has a lot of parts that need to connect together to allow your servers to asynchronously deliver messages to your Android applications.

The Account

You will need a Google account to represent the server from which the messages are delivered. The Android client application will register for messages from this account, and the server will send messages to Google for delivery using this account.

This account can be a pure Google account (e.g., @gmail.com) or one that is set up for your own domain using Google Apps. However, it is probably a good idea to use an account that you will not be using for anything else or likely to need to change. Considering that this account name will be "baked into" your Android application (in simple implementations, anyway), changing it may not be that easy.

During the C2DM beta period, this account is the one you will use on the C2DM signup form.

The Android App

Obviously, there is your Android app – without this, having a chapter on C2DM in this book would be rather silly. Your Android application will need at least one new class, some other additional Java code, and some manifest modifications to be able to participate in C2DM.

Your Server

Something has to send messages to the Android apps by way of Google. This is generally called "the server application", though technically it does not need to run on a server. Whatever it is, it will have a reason to send data asynchronously to your Android applications, and it will need to have

494

the ability to send HTTP requests to Google's servers to actually send that data.

Google's Server

Your server is not directly communicating with the Android apps. Instead, you send the messages to Google, who queues them up and will deliver them as soon as is practical. That may be nearly immediately, but it may take some time, particularly depending on how the message is configured and whether the device is on.

Google's On-Device Code

The reason that Android 2.2 is required is that 2.2 is the first release containing Google's code for managing its side of the C2DM connection. In effect, Google's on-device code maintains an open socket with its servers. Messages, when they arrive at the servers, are delivered over this open socket.

Google's Client Code

Google has created some client-side code to help you manage your C2DM registrations and messages, handling a lot of the boilerplate logic for you. As of the time of this writing, that code is part of the chrome2phone sample application. Google has indicated that it will be pulling that code out into a separate library, and this chapter demonstrates the use of that code.

Getting From Here to There

So, how does this all work?

First, your Android application will tell Google's on-device code that it wants to register for messages from your Google account. Using the Google C2DM client code, this is a single call to a static method on a class – under

the covers, it packages the information in an Intent and sends it to the Google on-device code.

When the registration occurs, you will be notified by a broadcast `Intent`, containing a registration ID. Google's C2DM client code will route that to an `IntentService`, where you can do whatever is necessary. A typical thing to do would be to make a Web service call to your server, supplying the registration ID, so the server knows how to send messages to your application on this device.

At this point, given the registration ID, the server is able to send messages to your app. It will do this by first getting a valid set of authentication credentials – effectively turning the Google account name and password into a long-lived authentication token. Then, your server can do an HTTP POST to the Google C2DM servers, supplying that authentication token, the registration ID of the app, and whatever data should be passed along.

Once Google's servers receive that POST, your app will receive the message at the next available opportunity. This could be in a matter of seconds. It could be in a matter of days, if the user is traveling and has their phone on "airplane mode". It could be anywhere in between. And, if the user does not pick up the message within a reasonable period of time, Google may drop the message.

Assuming the message makes it to the device, it will be routed to you via a broadcast `Intent`, perhaps handled by the same `IntentService` you set up for registration notices.

Your app can unregister whenever it wishes, to invalidate the registration ID and stop receiving messages.

Permissions for Push

C2DM uses Android permissions in a somewhat more sophisticated fashion than do most applications. That sophistication will require you to do a few things in your manifest above and beyond the norm.

First, you will need to request the INTERNET permission. Technically, this is only required if you are using the Internet (e.g., a Web service) to send the registration ID to your server. However, that will be a fairly typical pattern.

Next, you will need to request the com.google.android.c2dm.permission.RECEIVE permission. This allows your application to receive messages from the C2DM engine that forms the core of Google's on-device C2DM code.

You also will want to define a custom permission – C2D_MESSAGE, prefixed by your application's package – and declare that you use that permission. This will be used to help prevent other applications from spoofing you with fake C2DM messages.

If you are using the Google C2DM client code, as is shown in the sample project for this chapter, you will also need to request the WAKE_LOCK permission, as the C2DM client code uses a WakeLock to help ensure that the device stays awake long enough for you to handle incoming messages, much like the WakefulIntentService shown elsewhere in this book.

From the Push/C2DM sample project, here are the permission-related elements from AndroidManifest.xml corresponding to the preceding points:

```
<uses-permission
android:name="com.commonsware.android.c2dm.permission.C2D_MESSAGE" />
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

Registering an Interest

Now, let's start taking a closer look at some code, to get C2DM going in an application. Again, all source code listings are coming from the Push/C2DM sample project.

In a production application, you would probably register for messages from your server on first run of the app, such as after the user has launched it

from the launcher and clicked through any license agreement you might have. For the Push/C2DM sample, though, we have you type in your Google account name in an EditText, then click a Button to perform the registration:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <EditText android:id="@+id/account"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="4dip"
  />
  <Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Register!"
    android:onClick="registerAccount"
  />
</LinearLayout>
```

Using the Google C2DM client code, all you need to do to register for messages is call C2DMessaging.register(), supplying a Context (e.g., your Activity) and the Google account name:

```java
package com.commonsware.android.c2dm;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import com.google.android.c2dm.C2DMessaging;

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

  public void registerAccount(View v) {
    EditText acct=(EditText)findViewById(R.id.account);

    C2DMessaging.register(this, acct.getText().toString());
  }
}
```

To get your registration ID, and to receive messages later on, you will need to receive the broadcasts sent out by Google's on-device C2DM code. If you are using Google's C2DM client code, you can do this by implementing a class named `C2DMReceiver`, as a subclass of `C2DMBaseReceiver`:

```java
package com.commonsware.android.c2dm;

import android.content.Context;
import android.content.Intent;
import android.util.Log;
import com.google.android.c2dm.C2DMBaseReceiver;

public class C2DMReceiver extends C2DMBaseReceiver {
  public C2DMReceiver() {
    super("this.is.not@real.biz");
  }

  @Override
  public void onRegistered(Context context, String registrationId) {
    Log.w("C2DMReceiver-onRegistered", registrationId);
  }

  @Override
  public void onUnregistered(Context context) {
    Log.w("C2DMReceiver-onUnregistered", "got here!");
  }

  @Override
  public void onError(Context context, String errorId) {
    Log.w("C2DMReceiver-onError", errorId);
  }

  @Override
  protected void onMessage(Context context, Intent intent) {
    Log.w("C2DMReceiver", intent.getStringExtra("payload"));
  }
}
```

You must override the `onMessage()` and `onError()` methods, as they are declared abstract in `C2DMBaseReceiver`. `onMessage()` will be called when a message arrives; `onError()` will be called if there is some problem. Typically, you will also override `onRegistered()`, where you will get your registration ID and can pass that along to your Web service...or just dump it to LogCat, as shown above. You might also consider overriding `onUnregistered()`, which will be called if you call `C2DMessaging.unregister()` at some point to retract your interest in messages from this Google account. Also, `C2DMBaseReceiver` requests that you supply the Google account in the constructor. The sample application hard-wires in a fake value, because the

real Google account is being supplied via the `EditText` – your production code can probably hard-wire in the proper account name. Reportedly, this is only used for logging purposes at this time.

We will explain a bit more about how you interpret received messages later in this chapter.

You also need to add a few things to your manifest, above and beyond the permissions cited in the previous section.

First, you need to add your `C2DMReceiver` service, just as an ordinary `<service>` element, with no `<intent-filter>` required:

```
<service android:name=".C2DMReceiver" />
```

Then, you need to add `C2DMBroadcastReceiver`, via a `<receiver>` element, to your manifest. This class, supplied by the Google C2DM client code, will receive the C2DM broadcasts and will route them to your `C2DMReceiver` class. The `<receiver>` element is a little unusual:

```
<receiver android:name="com.google.android.c2dm.C2DMBroadcastReceiver"
          android:permission="com.google.android.c2dm.permission.SEND">

  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.RECEIVE" />
    <category android:name="com.commonsware.android.c2dm" />
  </intent-filter>
  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
    <category android:name="com.commonsware.android.c2dm" />
  </intent-filter>
</receiver>
```

Specifically:

- The `android:name` attribute has to specify the full class name, including package, since this is a class from Google's C2DM client code, not your own package

- For your protection, you should have the `android:permission="com.google.android.c2dm.permission.SEND"` attribute, to require the broadcaster of the `Intent` to hold that

permission, to further limit the ability for other applications to spoof messages from your app

- You need `<intent-filter>` elements for the `com.google.android.c2dm.intent.RECEIVE` and `com.google.android.c2dm.intent.REGISTRATION` actions, where the category for those filters is your application's package (`com.commonsware.android.c2dm` in this sample) – this ensures that the broadcasts will only go to your application, not to anyone else's

This is all largely boilerplate, except for the custom category values.

If you do all of that and register a Google account, you will get a registration ID back. This is a 120 character cryptic string that your server will need to send messages to this specific app on this specific device.

While this all seems a little bit complicated, the Google C2DM client code wraps up most of the ugliness – your code could be even *more* complicated!

Push It Real Good

Your server need to get the registration IDs from instances of your app, then send messages to those IDs when appropriate. Sending a message is a matter of doing 1 or 2 HTTP POST requests, and therefore can be accomplished by any serious server-side programming environment. You do not even strictly need a server for this – the `Push/C2DM` sample project will demonstrate sending a message using the curl command-line HTTP client.

Getting Authenticated

Before you can send a message, you need to authenticate yourself with Google's C2DM servers. This involves an HTTP POST request, where you supply your account credentials and get in return an authentication token. This uses the same basic logic that is used to log into any Google server for any of their exposed APIs, and there are client libraries for Google authentication available for many programming languages.

Here is the `auth.sh` script from the `Push/C2DM` project, showing how to perform an authentication request using `curl`:

```
curl https://www.google.com/accounts/ClientLogin -d Email=$1 -d "Passwd=$2" -d
accountType=GOOGLE -d source=Google-cURL-Example -d service=ac2dm
```

This script expects two command-line parameters: your Google account name (e.g., `foo@gmail.com`) and its password. The `curl` command supplies those two values with three others in a request to `ClientLogin`:

1. The `accountType`, which is `GOOGLE` if your account is a plain Google account or `HOSTED` if your account comes from one managed by Google App for your domain

2. The `source`, which apparently is an arbitrary string identifying what is making the authentication request

3. The `service`, which must be `ac2dm` for this to work

The result will be text response with three long strings, named `LID`, `SID`, and `Auth`. You will need the `Auth` value. This is a 160-character string, representing a token showing that you have been authenticated. This token will be good for several days, so you do not need to request a fresh `Auth` token on each message. Ideally, you do not even store the Google account information on the server, lest your server be hacked and that account be put to ill use. Instead, store the `Auth` token somewhere on the server and refresh it periodically. Also, a request to send a message will include an `Update-Client-Auth` header with a fresh `Auth` token if the Google C2DM servers determine that your existing token will expire soon.

Sending a Notification

Given the 120-character registration ID and the 160-character `Auth` token, you can now send a message to the app. This involves doing an HTTP POST to the C2DM servers themselves, as shown in the `post.sh` `curl` script:

```
curl --header "Authorization: GoogleLogin auth=$1"
"https://android.apis.google.com/c2dm/send" -d registration_id=$2 -d
"data.payload=$3" -d collapse_key=something
```

This script expects three command-line parameters:

1. The `Auth` token

2. The registration ID

3. The "payload" – a simple string that will be sent to the app

The `Auth` token goes in a `GoogleAuth Authorization` HTTP header. The registration ID is supplied as a parameter on the POST request, along with:

- Your specified payload, as a POST parameter named `data.payload`

- The `collapse_key`, which will be explained later in this chapter

About the Message

You can pass up to 1,024 characters' worth of data in your message, spread across one or more values. Each POST parameter prefixed with `data.` will be considered part of the message and will be put into the Intent sent to your C2DMReceiver class as a String extra (minus the `data.` prefix). Hence, the sample `post.sh` script uses `data.payload` for your message, and the sample C2DMReceiver implementation retrieves that via the `payload` Intent extra. While this sample only shows a single value being sent, you can provide several `data.` POST parameters if you wish, so long as they combine to be under 1,024 characters.

A Controlled Push

Of course, the Push/C2DM sample project is a simplified look at the entire push notification process. When you start dealing with thousands of users and thousands of messages, things get a wee bit more complicated. Here are a couple of control points you should be aware of as you think about applying these techniques to a production application.

Message Parameters

Devices may not be in position to receive messages right away. While the delay may be temporary, it could be of indefinite duration. Somebody having their phone turned off, or on "airplane mode", for an extended period is an obvious example. Even if the phone is on and operating normally, though, it may be that the socket connection between the device and the C2DM servers has been interrupted, and the power-optimized on-device C2DM code may be a bit slow to re-establish the connection. At the same time, you are going to be sending out messages typically based on your own schedule, such as in response to external data sources, ignorant of what is going on with any given device.

Google is not considering C2DM to be a guaranteed store-and-forward queue system. In particular, Google reserves the right to try to coalesce messages, in part to reduce storage demands, but also so as not to flood the device when a connection is re-established.

Key to this is the `collapse_key` parameter on the message request. If a device is unavailable, and during that time you send two or more messages with the same `collapse_key`, the C2DM servers may elect to only send one of those messages – typically the last one, though not necessarily. You can use this to your advantage, to minimize processing you need to do on the client. For example, if your use of C2DM is to alert your custom email application that "you've got mail", you can use a consistent `collapse_key` with messages telling the client how many unread emails are in their inbox. That can be used by the client to update a Notification and, eventually, cajole the user into actually reading her mail.

A related optional parameter you can include in your messages is `delay_while_idle`. If you specify this as a POST parameter, that will indicate to the C2DM servers that, while you want the message to be delivered, it is not important enough to wake up the device. C2DM will hold onto the message (or the last one if several are sent with the same `collapse_key`), but it will not push it to the device until it knows the device is awake (perhaps due to another C2DM message for that device lacking this parameter). You

can think of this as being akin to choosing an `AlarmManager` alarm type lacking the `_WAKEUP` suffix. The goal is to minimize battery consumption.

Notable Message Responses

When you send a message, you should get a `200 OK` response from the C2DM servers. If everything went well, you will get back a body of the form `id=...`, where `...` is some unique ID of the message. If, however, you get a body of `Error=...`, that means something went wrong.

Some errors, like `MissingCollapseKey`, will probably be found and fixed during development. Some errors, like `MessageTooBig`, are hopefully found during stress testing. Others, though, may legitimately happen during normal operations. In particular, here are four to watch for:

1. `QuotaExceeded` and `DeviceQuotaExceeded` will be returned if you have sent too many messages too quickly, either in general (`QuotaExceeded`) or to a specific device (`DeviceQuotaExceeded`). Google would appreciate it if you would try again later, perhaps using some sort of exponential back-off algorithm.

2. `InvalidRegistration` means that the registration ID you supplied is incorrect. This suggests there is some form of corruption in the channel by which you got that registration ID to the server.

3. `NotRegistered`, for a registration ID that used to work, means that the user has unregistered that ID, and it should no longer be used. If you get `NotRegistered` from the beginning, there may be a problem with your C2DM setup. In particular, during this beta period, it may mean there are problems with your Google ID that was added to the beta test whitelist.

The Right Way to Push

Google recommends that you use C2DM not to deliver data, but to deliver a wakeup call to your application, which then goes and pulls the data. C2DM is not a guaranteed store-and-forward engine – that, coupled with the `collapse_key` concept, means that not every one of your messages will make

it through to the device. If you put "real data" in the C2DM message, that data may be lost. Also, this means you will (hopefully) never run into the 1,024-byte cap on message length.

You may also need to do push by some means *other* than C2DM, in all likelihood. C2DM has two key limitations:

1. It only works on devices running Android 2.2 and higher, which at the time of this writing is a very small percentage of the market

2. It requires some of the infrastructure that powers the Android Market, and so may not be available on devices lacking the Market

The first limitation will fall away in time; how much the second limitation impacts you will be determined by the mix of devices your users are using. If a significant number are using older or non-Market devices, you will need some separate solution: polling, WebSockets, etc.

NFC

NFC, courtesy of high-profile boosters like Google Wallet, is poised to be a significant new capability in Android devices. While at the time of this writing, only the Samsung Nexus S has NFC built in, other handsets are slated to be NFC-capable in the coming months. Google is hoping that developers will write NFC-aware applications to help further drive adoption of this technology by device manufacturers.

This, of course, begs the question: what is NFC? Besides being where the Green Bay Packers play, that is?

(For those of you from outside of the United States, that was an American football joke. We now return you to your regularly-scheduled chapter.)

What Is NFC?

NFC stands for Near-Field Communications. It is a wireless standard for data exchange, aimed at very short range transmissions – on the order of a couple of centimeters. NFC is in wide use today, for everything from credit cards to passports. Typically, the NFC data exchange is for simple data – contact information, URLs, and the like.

In particular, NFC tends to be widely used where one side of the communications channel is "passive", or unpowered. The other side (the "initiator") broadcasts a signal, which the passive side converts into power

enough to send back its response. As such, NFC "tags" containing such passive targets can be made fairly small and can be embedded in a wide range of containers, from stickers to cards to hats.

The objective is "low friction" interaction – no pairing like with Bluetooth, no IP address shenanigans as with WiFi. The user just taps and goes.

...Compared to RFID?

NFC is often confused with or compared to RFID. It is simplest to think of RFID as being an umbrella term, under which NFC falls. Not every RFID technology is NFC, but many things that you hear of being "RFID" may actually be NFC-compliant devices or tags.

...Compared to QR Codes?

In many places, NFC will be used in ways you might consider using QR codes. For example, a restaurant could use either technology, or both, on a sign to lead patrons to the restaurant's Yelp page, as a way of soliciting reviews. Somebody with a capable device could either tap the NFC tag on the sign to bring up Yelp or take a picture of the QR code and use that to bring up Yelp.

NFC's primary advantage over QR codes is that it requires no user intervention beyond physically moving their device in close proximity to the tag. QR codes, on the other hand, require the user to launch a barcode scanning application, center the barcode in the viewfinder, and then get the results. The net is that NFC will be faster.

QR's advantages include:

- No need for any special hardware to generate the code, as opposed to needing a tag and something to write information into the tag for NFC

- The ability to display QR codes in distant locations (e.g., via Web sites), whereas NFC requires physical proximity

To NDEF, Or Not to NDEF

RFID is a concept, not a standard. As such, different vendors created their own ways of structuring data on these tags or chips, making one vendor's tags incompatible with another vendor's readers or writers. While various standards bodies, like ISO, have gotten involved, it's still a bit of a rat's nest of conflicting formats and approaches.

The NFC offshoot of RFID has had somewhat greater success in establishing standards. NFC itself is an ISO and ECMA standard, covering things like transport protocols and transfer speeds. And a consortium called the NFC Forum created NDEF – the NFC Data Exchange Format – for specifying the content of tags.

However, not all NFC tags necessarily support NDEF. NDEF is much newer than NFC, and so lots of NFC tags are out in the wild that were distributed before NDEF even existed.

You can roughly divide NFC tags into three buckets:

1. Those that support NDEF "out of the box"

2. Those that can be "formatted" as NDEF

3. Those that use other content schemes

Android has some support for non-NDEF tags, such as the MIFARE Classic. However, the hope and expectation going forward is that NFC tags will coalesce around NDEF.

NDEF, as it turns out, maps neatly to Android's `Intent` system, as you will see as we proceed through this chapter.

NDEF Modalities

Most developers interested in NFC will be interested in reading NFC tags and retrieving the NDEF data off of them. In Android, tapping an NDEF tag

with an NFC-capable device will trigger an activity to be started, based on a certain `IntentFilter`.

Some developers will be interested in writing to NFC tags, putting URLs, vCards, or other information on them. This may or may not be possible for any given tag.

And while the "traditional" thinking around NFC has been that one side of the communication is a passive tag, Android will help promote the "peer-to-peer" approach – having two Android devices exchange data via NFC and NDEF. Basically, putting the two devices back-to-back will cause each to detect the other device's "tag", and each can read and write to the other via this means.

The peer-to-peer approach exists in Android 2.3.4 (Gingerbread) and Android 3.0/3.1 (Honeycomb), but it is supposed to be given a significant boost in the Ice Cream Sandwich (ICS) release slated for later in 2011. ICS will also NFC-enable many of the built-in apps, to offer "zero-click sharing" – for example, viewing a contact will automatically push that contact to another NFC-capable phone if they are touching.

Right now, this chapter will focus more on interfacing NFC-capable Android devices with NDEF-capable tags. Peer-to-peer will be covered in a future edition of this book.

Of course, all of these are only available on hardware. At the present time, there is no emulator for NFC, nor any means of accessing a USB NFC reader or writer from the emulator.

NDEF Structure and Android's Translation

NDEF is made up of messages, themselves made up of a series of records. From Android's standpoint, each tag consists of one such message.

Each record consists of a binary (`byte[]`) payload plus metadata to describe the nature of the payload. The metadata primarily consists of a type and a subtype. There are quite a few combinations of these, but the big three for new Android NFC uses are:

1. A type of `TNF_WELL_KNOWN` and a subtype of `RTD_TEXT`, indicating that the payload is simply plain text

2. A type of `TNF_WELL_KNOWN` and a subtype of `RTD_URI`, indicating that the payload is a URI, such as a URL to a Web page

3. A type of `TNF_MIME_MEDIA`, where the subtype is a standard MIME type, indicating that the payload is of that MIME type

When Android scans an NDEF tag, it will use this information to construct a suitable `Intent` to use with `startActivity()`. The action will be `android.nfc.action.NDEF_DISCOVERED`, to distinguish the scanned-tag case from, say, something simply asking to view some content. The MIME type in the Intent will be `text/plain` for the first scenario above or the supplied MIME type for the third scenario above. The data (`Uri`) in the Intent will be the supplied URI for the second scenario above. Once constructed, Android will invoke `startActivity()` on that `Intent`, bringing up an activity or an activity chooser, as appropriate.

NFC-capable Android devices have a Tags application pre-installed that will handle any NFC tag not handled by some other app. So, for example, an NDEF tag with an HTTP URL will fire up the Tags application, which in turn will allow the user to open up a Web browser on that URL.

The Reality of NDEF

The enthusiasm that some have with regards to Android and NFC technology needs to be tempered by the reality of NDEF, NFC tags in general, and Android's support for NFC. It is easy to imagine all sorts of possibilities that may or may not be practical when current limitations are reached.

Some Tags are Read-Only

Some tags come "from the factory" read-only. Either you arrange for the distributor to write data onto them (e.g., blast a certain URL onto a bunch of NFC stickers to paste onto signs), or they come with some other pre-established data. Touchatag, for example, distributes NFC tags that have Touchatag URLs on them – they then help you set up redirects from their supplied URL to ones you supply.

While these tags will be of interest to consumers and businesses, they are unlikely to be of interest to Android developers, since their use cases are already established and typically do not need custom Android application support. Android developers seeking customizable tags will want ones that are read-write, or at least write-once.

Some Tags Can't Be Read-Only

Conversely, some tags lack any sort of read-only flag. An ideal tag for developers is one that is write-once: putting an NDEF message on the tag and flagging it read-only in one operation. Some tags do not support this, or making the tag read-only at any later point. The MIFARE Classic 1K tag is an example – while technically it can be made read-only, it requires a key known only to the tag manufacturer.

Some Tags Need to be Formatted

The MIFARE Classic 1K NFC tag is NDEF-capable, but must be "formatted" first, supplying the initial NDEF message contents. You have the option of formatting it read-write or read-only (turning the Classic 1K a write-once tag).

This is not a problem – in fact, the write-once option may be compelling. However, it is something to keep in mind.

Also, note that the MIFARE Classic 1K, while it can be formatted as NDEF, uses a proprietary protocol "under the covers". Not all Android devices will support the Classic 1K, as the device manufacturers elect not to pay the licensing fee. Where possible, try to stick to tags that are natively NDEF-compliant (so-called "NFC Forum Tag Types 1-4").

Tags Have Limited Storage

The "1K" in the name "MIFARE Classic 1K" refers to the amount of storage on the tag: 1 kilobyte of information.

And that's far larger than other tags, such as the MIFARE Ultralight C, some of which have ~64 bytes of storage.

Clearly, you will not be writing an MP3 file or JPEG photo to these tags. Rather, the tags will tend to either be a "launcher" into something with richer communications (e.g., URL to a Web site) or will use the sorts of data you may be used to from QR codes, such as a vCard or iCalendar for contact and event data, respectively.

NDEF Data Structures Are Documented Elsewhere

The Android developer documentation is focused on the Android classes related to NFC and on the Intent mechanism used for scanned tags. It does not focus on the actual structure of the payloads.

For `TNF_MIME_MEDIA` and `RTD_TEXT`, the payload is whatever you want. For `RTD_URI`, however, the byte array has a bit more structure to it, as the NDEF specification calls for a single byte to represent the URI prefix (e.g., `http://www.` versus `http://` versus `https://www.`). The objective, presumably, is to support incrementally longer URLs on tags with minuscule storage. Hence, you will need to convert your URLs into this sort of byte array if you are writing them out to a tag.

Generally speaking, the rules surrounding the structure of NDEF messages and records is found at the NFC Forum site.

Availability of NFC-Capable Android Devices

At the time of this writing, there is one NFC-capable Android device – the Samsung Nexus S. While ostensibly available for consumers, this device has largely been marketed at developers and Android aficionados (e.g., its firmware can be replaced readily).

Reports indicate that more NFC-capable devices will be showing up in stores in the coming months. However, right now, NFC features in an app will not be very widely used.

Sources of Tags

NFC tags are not the sort of thing you will find on your grocer's shelves. In fact, few, if any, mainstream firms sell them today.

Here are some online sites from which you can order rewritable NFC tags, listed here in alphabetical order:

- Buy NFC Stickers
- Buy NFC Tags
- Smartcard Focus
- tagstand

Note that not all may ship to your locale.

Writing to a Tag

So, let's see what it takes to write an NDEF message to a tag, formatting it if needed. The code samples shown in this chapter are from the NFC/URLTagger sample application. This application will set up an activity to respond to

`ACTION_SEND` activity `Intents`, with an eye towards receiving a URL from a browser, then waiting for a tag and writing the URL to that tag. The idea is that this sort of application could be used by non-technical people to populate tags containing URLs to their company's Web site, etc.

Getting a URL

First, we need to get a URL from the browser. As we saw in the chapter on integration, the standard Android browser uses `ACTION_SEND` of `text/plain` contents when the user chooses the "Share Page" menu. So, we have one activity, `URLTagger`, that will respond to such an `Intent`:

```
<activity android:name="URLTagger"
          android:label="@string/app_name">
    <intent-filter android:label="@string/app_name">
      <action android:name="android.intent.action.SEND" />
      <data android:mimeType="text/plain" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Of course, lots of other applications support `ACTION_SEND` of `text/plain` contents that are not URLs. A production-grade version of this application would want to validate the `EXTRA_TEXT` `Intent` extra to confirm that, indeed, this is a URL, before putting in an NDEF message claiming that it is a URL.

Detecting a Tag

When the user shares a URL with our application, our activity is launched. At that point, we need to go into "detect a tag" mode – the user should then tap their device to a tag, so we can write out the URL.

First, in `onCreate()`, we get access to the `NfcAdapter`, which is our gateway to much of the NFC functionality in Android:

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

```
  nfc=NfcAdapter.getDefaultAdapter(this);
}
```

We use a boolean data member – `inWriteMode` – to keep track of whether or not we are set up to write to a tag. Initially, of course, that is set to be false. Hence, when we are first launched, by the time we get to `onResume()`, we can go ahead and register our interest in future tags:

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

  if (!inWriteMode) {
    IntentFilter discovery=new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    IntentFilter[] tagFilters=new IntentFilter[] { discovery };
    Intent i=new Intent(this, getClass())
              .addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP|
                        Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

    inWriteMode=true;
    nfc.enableForegroundDispatch(this, pi, tagFilters, null);
  }
}
```

When an NDEF-capable tag is within signal range of the device, Android will invoke `startActivity()` for the `NfcAdapter.ACTION_TAG_DISCOVERED` Intent action. However, it can do this in one of two ways:

1. Normally, it will use a chooser (via `Intent.createChooser()`) to allow the user to pick from any activities that claim to support this action.

2. The foreground application can request via `enableForegroundDispatch()` for it to handle all tag events while it is in the foreground, superseding the normal `startActivity()` flow. In this case, while Android still will invoke an activity, it will be our activity, not any other one.

We want the second approach right now, so the next tag brought in range is the one we will try writing to.

To do that, we need to create an array of `IntentFilter` objects, identifying the NFC-related actions that we want to capture in the foreground. In this

516

case, we only care about `ACTION_TAG_DISCOVERED` – if we were supporting non-NDEF NFC tags, we might also need to watch for `ACTION_TECH_DISCOVERED`.

We also need a `PendingIntent` identifying the activity that should be invoked when such a tag is encountered while we are in the foreground. Typically, this will be the current activity. By adding `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to the `Intent` as flags, we ensure that our current specific instance of the activity will be given control again via `onNewIntent()`.

Armed with those two values, we can call `enableForegroundDispatch()` on the `NfcAdapter` to register our request to process tags via the current activity instance.

In `onPause()`, if the activity is finishing, we call `disableForegroundDispatch()` to undo the work done in `onResume()`:

```java
@Override
public void onPause() {
  if (isFinishing()) {
    nfc.disableForegroundDispatch(this);
    inWriteMode=false;
  }

  super.onPause();
}
```

We have to see if we are finishing, because even though our activity never leaves the screen, Android still calls `onPause()` and `onResume()` as part of delivering the `Intent` to `onNewInent()`. Our approach, though, has flaws – if the user presses HOME, for example, we never disable the NFC dispatch logic. A production-grade application would need to handle this better.

For any of this code to work, we need to hold the NFC permission via an appropriate line in the manifest:

```xml
<uses-permission android:name="android.permission.NFC" />
```

Also note that if you have several activities that the user can reach while you are trying to also capture NFC tag events, you will need to call `enableForegroundDispatch()` in each activity – it's a per-activity request, not a per-application request.

Reacting to a Tag

Once the user brings a tag in range, `onNewIntent()` will be invoked with the `ACTION_TAG_DISCOVERED` Intent action:

```
@Override
protected void onNewIntent(Intent intent) {
  if (inWriteMode &&
      NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
    Tag tag=intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    byte[] url=buildUrlBytes();
    NdefRecord record=new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
                                     NdefRecord.RTD_URI,
                                     new byte[] {}, url);
    NdefMessage msg=new NdefMessage(new NdefRecord[] {record});

    new WriteTask(this, msg, tag).execute();
  }
}
```

If we are in write mode and the delivered Intent is indeed an `ACTION_TAG_DISCOVERED` one, we can get at the `Tag` object associated with the user's NFC tag via the `NfcAdapter.EXTRA_TAG Parcelable` extra on the `Intent`.

Writing an NDEF message to the tag, therefore, is a matter of crafting the message and actually writing it. An NDEF message consists of one or more records (though, typically, only one record is used), with each record wrapping around a byte array of payload data.

Getting the Shared URL

We did not do anything to get the URL out of the `Intent` back in `onCreate()`, when our activity was first started up. Now, of course, we need that URL. You might think it is too late to get it, since our activity was effectively started again due to the tag and `onNewIntent()`.

However, `getIntent()` on an `Activity` always returns the `Intent` used to create the activity in the first place. The `getIntent()` value is not replaced when `onNewIntent()` is called.

Hence, as part of the `buildUrlBytes()` method to create the binary payload, we can go and call `getIntent().getStringExtra(Intent.EXTRA_TEXT)` to retrieve the URL.

Creating the Byte Array

Given the URL, we need to convert it into a byte array suitable for use in a `TNF_WELL_KNOWN`, `RTD_URI` NDEF record. Ordinarily, you would just call `toByteArray()` on the `String` and be done with it. However, the byte array we need uses a single byte to indicate the URL prefix, with the rest of the byte array for the characters after this prefix.

This is efficient. This is understandable. This is annoying.

First, we need the roster of prefixes, defined in `URLTagger` as a static data member cunningly named `PREFIXES`:

```
static private final String[] PREFIXES={"http://www.", "https://www.",
                                        "http://", "https://",
                                        "tel:", "mailto:",
                                        "ftp://anonymous:anonymous@",
                                        "ftp://ftp.", "ftps://",
                                        "sftp://", "smb://",
                                        "nfs://", "ftp://",
                                        "dav://", "news:",
                                        "telnet://", "imap:",
                                        "rtsp://", "urn:",
                                        "pop:", "sip:", "sips:",
                                        "tftp:", "btspp://",
                                        "btl2cap://", "btgoep://",
                                        "tcpobex://",
                                        "irdaobex://",
                                        "file://", "urn:epc:id:",
                                        "urn:epc:tag:",
                                        "urn:epc:pat:",
                                        "urn:epc:raw:",
                                        "urn:epc:", "urn:nfc:"};
```

Then, in buildUrlBytes(), we need to find the prefix (if any) and use it:

```
private byte[] buildUrlBytes() {
  String raw=getIntent().getStringExtra(Intent.EXTRA_TEXT);
  int prefix=0;
  String subset=raw;

  for (int i=0;i<PREFIXES.length;i++) {
    if (raw.startsWith(PREFIXES[i])) {
      prefix=i+1;
      subset=raw.substring(PREFIXES[i].length());

      break;
    }
  }

  byte[] subsetBytes=subset.getBytes();
  byte[] result=new byte[subsetBytes.length+1];

  result[0]=(byte)prefix;
  System.arraycopy(subsetBytes, 0, result, 1, subsetBytes.length);

  return(result);
}
```

We iterate over the PREFIXES array and find a match, if any. If there is a match, we record the NDEF value for the first byte (our PREFIXES index plus one) and create a subset string containing the characters after the prefix. If there is no matching prefix, the prefix byte is 0 and we will include the full URL.

Given that, we construct a byte array containing our prefix byte in the first slot, and the rest taken up by the byte array of the subset of our URL.

Creating the NDEF Record and Message

Given the result of buildUrlBytes(), our onNewIntent() implementation creates a TNF_WELL_KNOWN, RTD_URI NdefRecord object, and pours that into an NdefMessage object.

The third parameter to the NdefRecord constructor is a byte array representing the optional "ID" of this record, which is not necessary here.

Finally, we delegate the actual writing to a `WriteTask` subclass of `AsyncTask`, as writing the `NdefMessage` to the `Tag` is... interesting.

Writing to a Tag

Here is the aforementioned `WriteTask` static inner class:

```
static class WriteTask extends AsyncTask<Void, Void, Void> {
    Activity host=null;
    NdefMessage msg=null;
    Tag tag=null;
    String text=null;

    WriteTask(Activity host, NdefMessage msg, Tag tag) {
        this.host=host;
        this.msg=msg;
        this.tag=tag;
    }

    @Override
    protected Void doInBackground(Void... arg0) {
        int size=msg.toByteArray().length;

        try {
            Ndef ndef=Ndef.get(tag);

            if (ndef==null) {
                NdefFormatable formatable=NdefFormatable.get(tag);

                if (formatable!=null) {
                    try {
                        formatable.connect();

                        try {
                            formatable.format(msg);
                        }
                        catch (Exception e) {
                            text="Tag refused to format";
                        }
                    }
                    catch (Exception e) {
                        text="Tag refused to connect";
                    }
                    finally {
                        formatable.close();
                    }
                }
                else {
                    text="Tag does not support NDEF";
                }
```

```
        }
      else {
        ndef.connect();

        try {
          if (!ndef.isWritable()) {
            text="Tag is read-only";
          }
          else if (ndef.getMaxSize()<size) {
            text="Message is too big for tag";
          }
          else {
            ndef.writeNdefMessage(msg);
          }
        }
        catch (Exception e) {
          text="Tag refused to connect";
        }
        finally {
          ndef.close();
        }
      }
    }
    catch (Exception e) {
      Log.e("URLTagger", "Exception when writing tag", e);
      text="General exception: "+e.getMessage();
    }

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    if (text!=null) {
      Toast.makeText(host, text, Toast.LENGTH_SHORT).show();
    }

    host.finish();
  }
}
```

In doInBackground(), after making note of how big the message is in bytes, we first try to get the Ndef aspect of the Tag object, by calling the static get() method on the Ndef class. If the tag is an NDEF tag, this should return an Ndef instance. If it does not, we try to get an NdefFormatable aspect by calling get() on the NdefFormatable class. If the tag is not NDEF now but can be formatted as NDEF, this should give us an NdefFormatable object. If both aspect attempts fail, we bail out, displaying a Toast to let the user know that while the tag they used is NFC, it is not NDEF-compliant.

If the tag turned out to be `NdefFormatable`, to put the `NdefMessage` on it, we first `connect()` to the tag, then `format()` it, supplying the message. `NdefFormatable` also supports `formatReadOnly()` for tags that support that mode – this will write the message on the tag, then block it from further updates. When we are done, we `close()` the connection.

If the tag turned out to be `Ndef` already, we `connect()` to it, then see if it is writable and has enough room. If it meets both of those criteria, we can emit the message via `writeNdefMessage()`, which overwrites the NDEF message that had already existed on the tag (if any). If the tag supported it, a call to `makeReadOnly()` would block further updates to the tag. Again, when we are done, we `close()` the connection.

All of the actual NFC I/O is performed in `doInBackground()`, because this I/O may take some time, and we do not want to block the main application thread while doing it.

Responding to a Tag

Writing to a tag is a bit complicated. Responding to an NDEF message on a tag is significantly easier.

If the foreground activity is not consuming NFC events – as `URLTagger` does in write mode – then Android will use normal `Intent` resolution with `startActivity()` to handle the tag. To respond to the tag, all you need to do is have an activity set up to watch for an `android.nfc.action.NDEF_DISCOVERED` `Intent`. To get control ahead of the built-in Tags application, also have a `<data>` element that describes the sort of content or URL you are expecting to find on the tag.

For example, suppose you used the Android browser to visit http://commonsware.com/nfctest, and you wrote that to a tag using `URLTagger`. The URLTagger application has another activity, `URLHandler`, that will respond when you tap the newly-written tag from the home screen or anywhere else. It accomplishes this via a suitable `<intent-filter>`:

```
<activity android:name="URLHandler"
          android:label="@string/app_name">
    <intent-filter android:label="@string/app_name">
      <action android:name="android.nfc.action.NDEF_DISCOVERED" />
      <data android:scheme="http"
            android:host="commonsware.com"
            android:path="/nfctest"
      />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

The URLHandler activity can then use getIntent() to retrieve the key pieces of data from the tag itself, if needed. In particular, the EXTRA_NDEF_MESSAGES Parcelable array extra will return an array of NdefMessage objects. Typically, there will only be one of these. You can call getRecords() on the NdefMessage to get at the array of NdefRecord objects (again, typically only one). Methods like getPayload() will allow you to get at the individual portions of the record.

The nice thing is that the URL still works, even if URLTagger is not on the device. In that case, the Tags application would react to the tag, and the user could tap on it to bring up a browser on this URL. A production application might create a Web page that tells the user about this great and wonderful app the can install, and provide links to the Android Market (or elsewhere) to go get the app.

Expected Pattern: Bootstrap

Tags tend to have limited capacity. Even in peer-to-peer settings, the effective bandwidth of NFC is paltry compared to anything outside of dial-up Internet access.

As a result, NFC will be used infrequently as the complete communications solution between a publisher and a device. Sometimes it will, when the content is specifically small, such as a contact (vCard) or event (iCalendar). But, for anything bigger than that, NFC will serve more as a convenient bootstrap for more conventional communications options:

- Embedding a URL in a tag, as the previous sample showed, allows an installed application to run or a Web site to be browsed

- Embedding an Android Market URL in a tag allows for easy access to some specialized app (e.g., menu for a restaurant)

- A multi-player game might use peer-to-peer NFC to allow local participants to rapidly connect into the same shared game area, where the game is played over the Internet or Bluetooth

- And so on.

Mobile Devices are Mobile

Reading and writing NFC tags is a relatively slow process, mostly due to low bandwidth. It may take a second or two to actually complete the operation.

Users, however, are not know for their patience.

If a user moves their device out of range of the tag while Android is attempting to read it, Android simply will skip the dispatch. If, however, the tag leaves the signal area of the device while you are writing to it, you will get an `IOException`. At this point, the state of the tag is unknown.

You may wish to incorporate something into your UI to let the user know that you are working with the tag, encouraging them to leave the phone in place until you are done.

Additional Resources

To help make sense of the tags that you are trying to use with your app, you may wish to grab the NFC TagInfo application off of the Android Market. This application simply scans a tag and allows you to peruse all the details of that tag, including the supported technologies (e.g., does it support NDEF? is it `NdefFormatable`?), the NDEF records, and so on.

To learn more about NFC on Android – beyond this chapter or the Android developer documentation – this Google I|O 2011 presentation is recommended.

PART IV – Scripting Languages

The Role of Scripting Languages

A scripting language, for the purpose of this book, has two characteristics:

- It is interpreted from source and so does not require any sort of compilation step

- It cannot (presently) be used to create a full-fledged Android application without at least some form of custom Java-based stub, and probably much more than that

In this part of the book, we will look at scripting languages on Android and what you can accomplish with them, despite any limitations inherent in their collective definition.

All Grown Up

Interpreted languages have been a part of the programming landscape for decades. The language most associated with the desktop computer revolution – BASIC – was originally an interpreted language. However, the advent of MS-DOS and the IBM PC (and clones) led developers in the direction of C for "serious programming", for reasons of speed. While interpreted languages continued to evolve, they tended to be described as "scripting" languages, used to glue other applications together. Perl, Python, and the like were not considered "serious" contenders for application development.

The follow-on revolution, for the Internet, changed all of that. Most interactive Web sites were written as CGI scripts using these "toy" languages, Perl first and foremost. Even in environments where Perl was unpopular, such as Windows, Web applications were still written using scripting languages, such as VBScript in Active Server Pages (ASP). While some firms developed Web applications using C/C++, scripting languages ruled the roost. That remains to this day, where you are far more likely to find people writing Web applications in PHP or Ruby than you will find them writing in C or C++. The most likely compiled language for Web development – Java – is still technically an interpreted language, albeit not usually considered a scripting language.

Nowadays, writing major components of an application using a scripting language is not terribly surprising. While this is still most common with Web applications, you can find scripting languages used in the browser (Javascript), games (Lua), virtual worlds (LSL), and so on. Even though these languages execute more slowly than there C/C++ counterparts, they offer much greater flexibility, and faster CPUs make the performance of scripts less critical.

Following the Script

Scripting languages are not built into Android, beyond the Javascript interpreter in the WebKit Web browser. Despite this, there is quite a bit of interest in scripting on Android, and the biggest reasons for this come down to experience and comfort level.

Your Expertise

Perhaps you have spent your entire career writing Python scripts, or you cut your teeth on Perl CGI programs, or you have gotten seriously into Ruby development.

Maybe you used Java in previous jobs and hate it with the fiery passion of a thousand suns.

Regardless of the cause, your expertise may lie outside the traditional Android realm of Java-based development. Perhaps you would never touch Android if you had to write in Java, or maybe you feel you would just be significantly more productive in some other language. How much that productivity gain is real versus "in your head" is immaterial – if you want to develop in some other language, you owe it to yourself to try.

Your Users' Expertise

Maybe you are looking to create a program where not only you can write scripts, but so can your users. This might be a utility, or a game, or rulesets for email management, or whatever.

In that case, you need:

- Something interpreted, so you can execute what the user types in

- Something embeddable, so your larger application (typically written in Java, of course) is capable of executing those scripts

- Something your users will be comfortable using for scripting

The last criterion is perhaps the toughest, as non-developers typically have limited experience in writing scripts in any language, let alone one that runs on Android. Perhaps the most popular such language is Basic, in the form of VBA and VBScript on Windows...but there are no interpreters for those languages for Android at this time.

Crowd-Developing

Perhaps your users will not only be entering scripts for their own benefit, but for others' benefit as well.

Many platforms have been improved by power users and amateur developers alike. Browser users gain from those writing GreaseMonkey scripts. Bloggers benefit from those writing WordPress themes. And so on.

To facilitate this sort of work, not only do you need an interpreted, embeddable, user-familiar scripting environment, but you need some means for users to publish their scripts and download the scripts of others. Fortunately, with Android having near-continuous connectivity, your challenge will lie more on organizing and hosting the scripts, more so than getting them on and off of devices.

Going Off-Script

Scripting languages on Android have their fair share of issues. It is safe to say that while Android does not prohibit the use of scripting languages, its architecture does not exactly go out of its way to make them easy to use, either.

Security

For a scripting language to do much that is interesting, it is going to need some amount of privileges. A script cannot access the Internet unless its process has that right. A script cannot modify the user's contacts unless its process has that right. And so on.

For scripts you write, so long as those scripts cannot be modified readily by malware authors, security is whatever you define it to be. If your script-based application needs Internet access, so be it.

For scripts your users write, things get a bit more challenging, since permissions cannot be modified on the fly by applications. Many interpreters will tend to request (or otherwise have access to) permissions that are broader than any individual user might need, because those permissions are needed by somebody. However, the risk is still minimal to the user, so long as they are careful with the scripts they write.

For scripts your users might download, written by others, security becomes a big problem. If the interpreter has a wide range of permissions, downloaded scripts can easily host malware that exploits those permissions for nefarious ends. An interpreter with both Internet access and the right to

read the user's contacts means that any script the user might download and run could copy the user's contact data and send it to spammers or identity thieves.

Performance

Java, as interpreted by the Dalvik virtual machine, is reasonably fast, particularly on Android 2.2 and newer versions. C/C++, through the NDK, is far faster.

Scripting languages are a mixed bag.

Some scripting languages for Android have interpreters that are implemented in C code. Those interpreters' performance is partly a function of how well they were written and ported over to the chipsets Android runs on. However, if those interpreters expose Android APIs to the language, that can add considerable overhead. For example, the Scripting Layer for Android (SL4A) makes Android APIs available to scripting languages via a tiny built-in Java Web server and a Web service API. While convenient for language integration, converting simple Java calls into Web service calls slows things down quite a bit.

Some scripting languages have interpreters that themselves are written in Java and run on the virtual machine. Those are likely to perform worse on an Android device than when they are run on a desktop or server, simply because of the performance differences between the standard Java VMs and the Dalvik VM. However, they will have quicker access to the Java class libraries that make up much of Android than will C-based interpreters.

Cross-Platform Compatibility

Most of the scripting languages for Android are ports from versions that run across multiple platforms. This is one of their big benefits – that is where you and your users may have gained experience with those languages. However, just as, say, Perl and Python run a bit differently on Windows than on Linux or OS X, there will be some differences in how

those languages run on Android. The Android operating system is not a traditional Linux environment, and so file paths, environment variables, available pre-installed programs, and the like will not be the same. Some of those may, in turn, impact how the scripting languages operate. You may need to make some modification to any existing scripts for those languages that you attempt to run on Android.

Maturity...On Android

Some scripting languages that have been ported to Android are rather old, like Perl and Python. Others are old and somewhat abandoned for traditional development, like BeanShell. Yet others are fairly new to the programming scene altogether, like JRuby.

However, none of them have a long track record on Android, simply because Android itself has not been around very long. This has several implications:

- There is more likely to be bugs in newer ports of a language than older ports

- Fewer people will have experience in supporting these languages on Android (compared to supporting them on Linux, for example)

- The number of production applications built using these languages on Android is minuscule compared to their use on more traditional environments

The Scripting Layer for Android

When it comes to scripting languages on Android, the first stop should always be the Scripting Layer for Android (SL4A). Led by Damon Kohler, this project is rather popular, both among hardcore Android developers and those people looking to automate a bit more of their Android experience.

The Role of SL4A

What started as an experiment to get Python and Lua going on Android, back in late 2008, turned into a more serious endeavor in June 2009, when the Android Scripting Environment (now called the Scripting Layer for Android, or SL4A) was announced on the Google Open Source blog and the Google Code site for it was established. Since then, SL4A has been a magnet for people interested in getting their favorite language working on Android or advancing its support.

On-Device Development

Historically, the primary role of SL4A was as a tool to allow people to put together scripts, often written on the device itself, to take care of various chores. This appealed to developers who were looking for something lightweight compared to the Android SDK and Java. For those used to tinkering with scripts on other mobile Linux platforms (e.g., the Nokia N800 running Maemo), SL4A promised a similar sort of capability.

Over time, SL4A's scope in this area has grown, including preliminary support for SL4A scripts packaged as APK files, much like an Android application written in Java or any of the alternative frameworks described in this book.

Getting Started with SL4A

SL4A is a bit more difficult to install than is the average Android application, due to the various interpreters it uses and their respective sizes. That being said, none of the steps involved with getting SL4A set up are terribly difficult, and most are just part of the application itself.

Installing SL4A

At the time of this writing, SL4A is not distributed via the Android Market. Instead, you can download it to your device off of the SL4A Web site. Perhaps the easiest way to do that is to scan the QR code on the SL4A home page using Barcode Scanner or a similar utility.

Installing Interpreters

When you first install SL4A, the only available scripting language is for shell scripts, as that is built into Android itself. If you want to work with other interpreters, you will need to download those. That is why the base SL4A download is so small (~200KB) – most of the smarts are separate downloads, largely due to size.

To add interpreters, launch SL4A from the launcher, then choose View > Interpreters from the option menu. You will be presented with the (presently short) list of installed interpreters:

Figure 101. The initial list of installed SL4A interpreters

Then, to install additional interpreters, choose Add from the option menu. You will be given a roster of SL4A-compatible interpreters to choose from:

Figure 102. The list of available SL4A interpreters

Click on one of the interpreters, and this will trigger the download of an APK file for that specific interpreter. Slide down the notification drawer and click on that APK file to continue the installation process. When the APK itself is installed, open up that interpreter (e.g., click the [Open] button when the install is done). That will bring up an activity to let you download the rest of the interpreter binaries:

Figure 103. Downloading the Python SL4A interpreter, continued

Click the Install button, and SL4A will download and install the interpreter's component parts:

Figure 104. Downloading the Python SL4A interpreter

This may take one or several downloads, depending on the interpreter. When done, and after a few progress dialogs' worth of unpacking, the interpreter will appear in the list of interpreters:
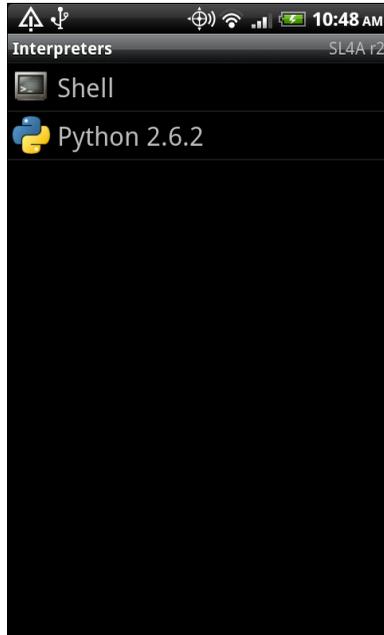
Figure 105. The updated list of installed SL4A interpreters

Note that the interpreters will be installed on your device's "external storage" (typically some flavor of SD card), due to their size. You will find an SL4A/ directory on that card with the interpreters and scripts.

Running Supplied Scripts

Back on the Scripts activity (e.g., what you see when you launch SL4A from the launcher), you will be presented with a list of the available scripts. Initially, these will be ones that shipped with the interpreters, as examples for how to write SL4A scripts in that language:
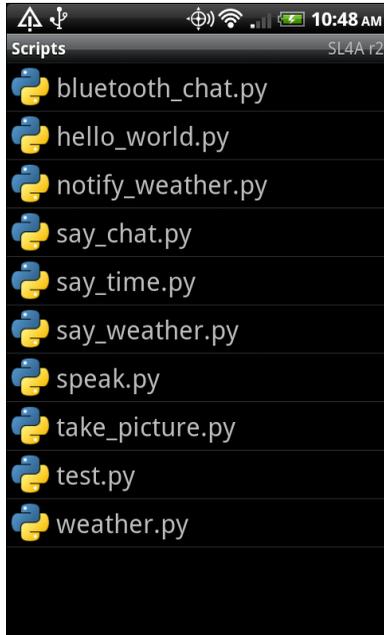
Figure 106. The list of SL4A scripts

Tapping on any of these scripts will bring up a "quick actions" balloon:
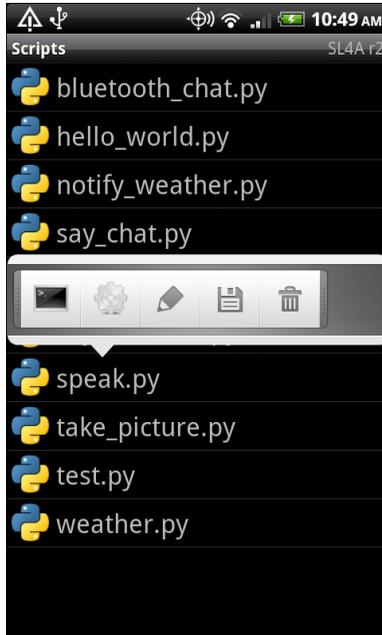
Figure 107. Quick actions for the speak.py script

Click the little shell icon to run it, showing its terminal output along the way:

543
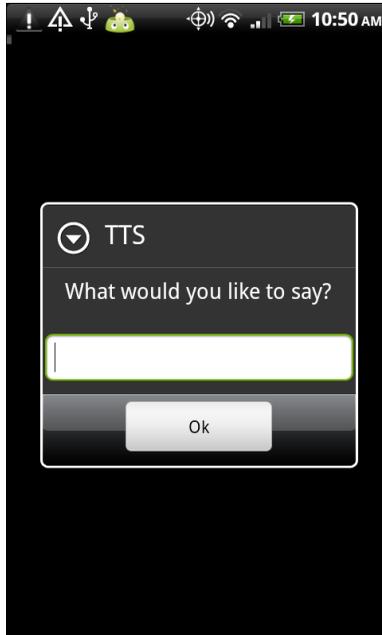
Figure 108. The visual results of running the speak.py SL4A script

Writing SL4A Scripts

While the scripts supplied with the interpreters are...entertaining, they only scratch the surface of what an SL4A script can accomplish. Of course, to go beyond what is there, you will need to start writing some scripts.

Editing Options

Since scripts are stored on your SD card (or whatever the "external storage" is for your device), you can create scripts using some other computer – one with fancy things like "mice" and "ergonomic keyboards" – and transfer it over via USB, like you would transfer over an MP3 file. This eases typing, but it will make for an awkward development cycle, since your computer and the Android device cannot both have access to the SD card simultaneously. The mount/unmount process may get a bit annoying. On the other hand, this is a great way to transfer over a script you obtained from somebody else.

Another option is to edit your scripts on the device. SL4A has a built in script editor designed for this purpose. Of course, the screen may be a bit small and the keyboard may be a bit...soft, but this is a great answer for small scripts.

To add a new script, from the Scripts activity, choose Add from the option menu. This will bring up a roster of available scripting languages and other items (e.g., add a folder):
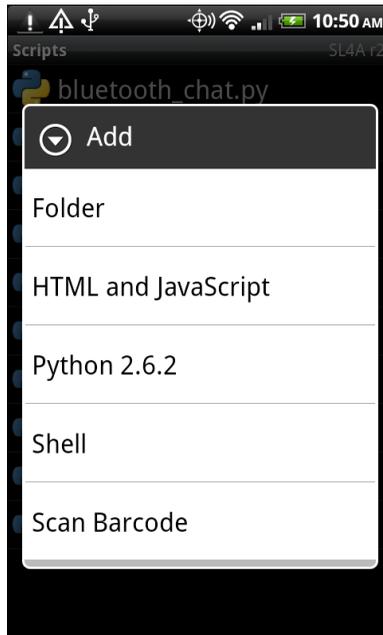
Figure 109. The add-script language selection dialog

(the "Scan Barcode" option gives you an easy route to install a third-party script, one encoded in a QR code)

Tap the language you want, and you will be taken into the script editor:

Figure 110. The script editor

The field at the top is for the script name, and the large text area at the bottom is for the script itself. A file extension and boilerplate code will be supplied for you automatically.

In fact, that boilerplate code is rather important, as you will see momentarily.

To edit an existing script, long-tap on the script in the list and choose Edit from the context menu.

To save your changes to a new or existing script, choose the Save option from the script editor option menu. You can also "Save and Run" to test the script immediately.

Calling Into Android

In the real world, Perl knows nothing about Android. Neither does Python, BeanShell, or most of the other scripting languages available for SL4A. This would be rather limiting, as most of what you would want a script to do will have to deal with the device to some level: collect input, get a location, say some text using speech synthesis, dial the phone, etc.

Fortunately, SL4A has a solution, one of those "so crazy, it just might work" sorts of solutions: SL4A has a built-in RPC server. While implementing a server on a smartphone is not something one ordinarily does, it provides an ingenious bridge from the scripting language to the device itself.

Each scripting language is given a local object proxy that works with the RPC server. For example, here is a Python script that speaks the current time:

Figure 111. The script editor, showing the say_time.py script

The `import android` and `droid=android.Android()` statements establish a connection between the Python interpreter and the SL4A RPC server. From that point, the `droid` object is available for use to access Android capabilities – in this case, speaking a message.

Python does not strictly realize that it is accessing local functionality. It simply makes RPC calls, ones that just so happen to be fulfilled on the device rather than via some remote RPC server accessed over the Internet.

Browsing the API

Therefore, SL4A effectively exposes an API to each of its scripting languages, via this RPC bridge. While the API is not huge, it accomplishes a lot and is ever-growing.

If you are editing scripts on the device, you can browse the API by choosing the API Browser option menu from the script editor. This brings up a list of available methods on your RPC proxy (e.g., `droid`) that you can call:

Figure 112. The script editor's API browser

Tapping on any item in the list will "unfold" it to provide more details, such as the parameter list. Long-tapping on an item brings up a context menu where you can:

- insert a template call to the method into your script at the cursor position

- "prompt" you for the parameter values for the method, then insert the completed method call into your script

It is also possible to browse the API in a regular Web browser, if you are developing scripts off-device.

Running SL4A Scripts

Scripts are only useful if you run them, of course. We have seen two options for running scripts: tapping on them in the scripts list, or choosing "Save & Run" from the script editor. Those are not your only options, however.

Background

If you long-tap on a script in the script list, you will see a context menu option to "Start in Background". As the name suggests, this kicks off the script in the background. Rather than seeing the terminal window for the script, the script just runs. A notification will appear in the status bar, with the SL4A icon, indicating that the RPC server is in operation and that script(s) may be running.

Shortcuts

Rather than have to open up SL4A every time, you can set up shortcuts on your home screen to run individual scripts. Just long-tap on the home screen background and choose Shortcuts from the context menu, then Scripts from the available shortcuts. This brings up the scripts list, but this time, when you choose a script, you are presented with a quick actions balloon for how to start it: in a terminal or in the background:
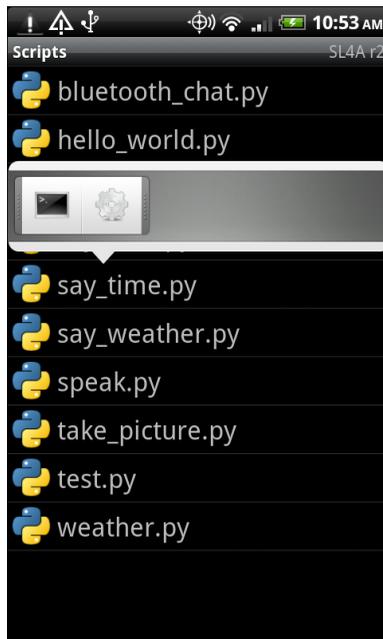
Figure 113. Configuring an SL4A shortcut

Choose one, and at this point, a shortcut, with the interpreter's icon and the name of the script, will appear on your home screen. Tapping it runs the script.

Other Alternatives

Users of Locale – an application designed to trigger events at certain times or when you get to certain locations – can trigger SL4A scripts in addition to invoking standard built-in tools.

In addition, there is preliminary support in SL4A for packaging scripts as APK files for wider distribution.

Potential Issues

As the SL4A Web site indicates, SL4A is "alpha-quality". It is not without warts. How much those warts are an issue for you, in terms of crafting and running utility scripts, is up to you.

Security...From Scripts

SL4A itself holds a long list of Android permissions, including:

- The ability to read your contact data
- The ability to call phone numbers and place SMS messages
- Access to your location
- Access to your received SMS/MMS messages
- Bluetooth access
- Internet access
- The ability to write to the SD card
- The ability to record audio and take pictures
- The ability to keep your device awake

- The ability to retrieve the list of running applications and restart other applications

- And so on

Hence, its scripts – via the RPC-based API – can perform all of those actions. For example, a script you download from a third party could read all your contacts and send that information to a spammer. Hence, you should only run scripts that you trust, since SL4A effectively "wires open" many aspects of Android's standard security protections.

Security...From Other Apps

Originally, the on-device Web service supplying the RPC-based API was wide open. Any program that could find the port could connect to that Web service and invoke operations. That would not necessarily be all that bad...except that the Web service runs in its own process with its own permissions, and it may have permissions that other applications lack (e.g., right to access the Internet or to read contacts). Given that, malware could use SL4A to do things that it, by itself, could not do, allowing it to sneak onto more devices.

SL4A now uses a token-based authentication mechanism for using the Web service, to help close this loophole. In principle, only SL4A scripts should be able to use the RPC server.

JVM Scripting Languages

The Java virtual machine (JVM) is a remarkably flexible engine. While it was originally developed purely for Java, it has spawned its own family of languages, just as Microsoft's CIL supports multiple languages for the Windows platform. Some languages targeting the JVM as a runtime will work on Android, since the regular Java VM and Android's Dalvik VM are so similar.

Languages on Languages

Except for the handful of early language interpreters and compilers hand-constructed in machine code, every programming language is built atop earlier ones. C and C++ are built atop assembly language. Many other languages, such as Java itself, are built atop C/C++.

Hence, it should not come as much of a surprise that an environment as popular as Java has spawned another generation of languages whose implementations are in Java.

There are a few flavors of these languages. Some, like Scala and Clojure, are compiled languages whose compilers created JVM bytecodes, no different than would a Java compiler. These do not strictly qualify as a "scripting language", however, since they typically compile their source code to bytecode ahead of time.

Some Java-based scripting languages use fairly simple interpreters. These interpreters convert scripting code into parsed representations (frequently so-called "abstract syntax trees", or ASTs), then execute the scripts from their parsed forms. Most scripting languages at least start here, and some, like BeanShell, stick with this implementation.

Other scripting languages try to bridge the gap between a purely interpreted language and a compiled one like Scala or Clojure. These languages turn the parsed scripting code into JVM bytecode, effectively implementing their own just-in-time compiler (JIT). Since many Java runtimes themselves have a JIT to turn bytecode into machine code ("opcode"), languages with their own JIT can significantly outperform their purely-interpreted counterparts. JRuby and Rhino are two languages that have taken this approach.

A Brief History of JVM Scripting

Back in the beginning, the only way to write for the JVM was in Java itself. However, since writing language interpreters is a common pastime, it did not take long for people to start implementing interpreters in Java. These had their niche audiences, but there was only modest interest in the early days – interpreters made Java applets too large to download, for example.

Things got a bit more interesting in 1999, when IBM released the Bean Scripting Framework (BSF). This offered a uniform API for scripting engines, meaning that a hosting Java application could write to the BSF API, then plug in arbitrary interpreters at runtime. It was even possible, with a bit of extra work, to allow new interpreters to be downloaded and used on demand, rather than having to be pre-installed with the application. BSF also standardized how to inject Java objects into the scripting engines themselves, for access by the scripts. This allowed scripts to work with the host application's objects, such as allowing scripts to manipulate the contents of the jEdit text editor.

This spurred interest in scripting. In addition to some IBM languages (e.g., NetREXX) supporting BSF natively, other languages, like BeanShell, created

BSF adapters to allow their languages to participate in the BSF space. On the consumer side, various Web frameworks started supporting BSF scripting for dynamic Web content generation, and so forth.

Interest was high enough that Apache took over stewardship of BSF in 2003. Shortly thereafter, Sun and others started work on JSR-223, which added the `javax.script` framework to Java 6. The `javax.script` framework advanced the BSF concept and standardized it as part of Java itself.

At this point, most JVM scripting languages that are currently maintained support `javax.script` integration, and may also support integration with the older BSF API as well. There is a long list of available `javax.script`-compatible scripting languages.

Android does not include `javax.script` as part of its subset of the Java SE class library from the Apache Harmony project. This certainly does not preclude integrating scripting languages into Android applications, but it does raise the degree of difficulty a bit.

Limitations

Of course, JVM scripting languages do not necessarily work on Android without issue. There may be some work to get a JVM language going on Android, above and beyond the challenges for scripting languages in general on Android.

Android SDK Limits

Android is not Java SE, or Java ME, or even Java EE. While Android has many standard Java classes, it does not have a class library that matches any traditional pattern. As such, languages built assuming Java SE, for example, may have some dependency issues.

For languages where you have access to the source code, removing these dependencies may be relatively straightforward, particularly if they are

ancillary to the operation of the language itself. For example, the language may come with miniature Swing IDEs, support for scripted servlets, or other capabilities that are not particularly relevant on Android and can be excised from the source code.

Wrong Bytecode

Android runs Dalvik bytecode, not Java bytecode. The conversion from Java bytecode to Dalvik bytecode happens at compile time. However, the conversion tool is rather finicky – it wants bytecode from Sun/Oracle's Java 1.5 or 1.6, nothing else. This can cause some problems:

- You may encounter a JAR that is old enough to have been compiled with Java 1.4.2

- You may encounter JARs compiled using other compilers, such as the GNU Compiler for Java (GCJ), common on Linux distributions

- Eventually, when Java 7 ships, there may be bytecode differences that preclude Java 7-compiled JARs from working with Android

- Languages that have their own JIT compilers will have problems, because their JIT compilers will be generating Java bytecodes, not Dalvik bytecodes, meaning that the JIT facility needs to be rewritten or disabled

Again, if you have the source code, recompiling on an Android-friendly Java compiler should be a simple process.

Age

The heyday of some JVM languages is in the past. As such, you may find that support for some languages will be limited, simply because few people are still interested in them. Finding people interested in those languages on Android – the cross-section of two niches – may be even more of a problem.

SL4A and JVM Languages

SL4A supports three JVM languages today:

- BeanShell

- JRuby

- Rhino (Javascript)

You can use those within your SL4A environment no different than you can any other scripting language (e.g., Perl, Python, PHP). Hence, if what you are looking for is to create your own personal scripts, or writing small applications, SL4A saves you a lot of hassle. If there is a JVM scripting language you like but is not supported by SL4A, adding support for new interpreters within SL4A is fairly straightforward, though the APIs may change as SL4A is undergoing a fairly frequent set of revisions.

Embedding JVM Languages

While SL4A will drive end users towards writing their own scripts or miniature applications using JVM languages, another use of these languages is for embedding in a full Android application. Scripting may accelerate development, if the developers are more comfortable with the scripted language than with Java. Also, if the scripts are able to be modified or expanded by users, an ecosystem may emerge for user-contributed scripts.

Architecture for Embedding

Embedding a scripting language is not something to be undertaken lightly, even on a desktop or server application. Mobile devices running Android will have similar issues.

Asynchronous

One potential problem is that a script may take too long to execute. Android's architecture assume that work triggered by buttons, menus, and the like will either happen very quickly or will be done on background threads. Particularly for user-generated scripts, the script execution time is unknowable in advance – it might be a few milliseconds, or it might be several seconds. Hence, any implementation of a scripting extension for an Android application needs to consider executing all scripts in a background thread. This, of course, raises its own challenges for reflecting those scripts' results on-screen, since GUI updates cannot be done on a background thread.

Security

Scripts in Android inherit the security restrictions of the process that runs the script. If an application has the right to access the Internet, so will any scripts run in that application's process. If an application has the right to read the user's contacts, so will any scripts run in that application's process. And so on. If the scripts in question are created by the application's authors, this is not a big deal – the rest of the application has those same permissions, after all. But, if the application supports user-authored scripts, it raises the potential of malware hijacking the application to do things that the malware itself would otherwise lack the rights to do.

Inside the InterpreterService

One way to solve both of those problems is to isolate the scripting language in a self-contained low-permission APK – "sandboxing" the interpreter so the scripts it executes are less able to cause harm. This APK could also arrange to have the interpreter execute its scripts on a background thread. An even better implementation would allow the embedding application to decide whether or not the "sandbox" is important – applications with a controlled source of scripts may not need the extra security or the implementation headaches it causes.

With that in mind, let us take a look at the `JVM/InterpreterService` sample project, one possible implementation of the strategy described above.

The Interpreter Interface

The `InterpreterService` can support an arbitrary number of interpreters, via a common interface. This interface provides a simplified API for having an interpreter execute a script and return a result:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public interface I_Interpreter {
  Bundle executeScript(Bundle input);
}
```

As you can see, it is *very* simplified, offering just a single `executeScript()` method. That method accepts a `Bundle` (a key-value store akin to a Java `HashMap`) as a parameter – that `Bundle` will need to contain the script and any other objects needed to execute the script.

The interpreter will return another `Bundle` from `executeScript()`, containing whatever data it wants the script's requester to have access to.

For example, here is the implementation of `EchoInterpreter`, which just returns the same `Bundle` that was passed in:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public class EchoInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    return(input);
  }
}
```

A somewhat more elaborate sample is the `SQLiteInterpreter`:

```
package com.commonsware.abj.interp;

import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import java.util.ArrayList;

public class SQLiteInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    Bundle result=new Bundle(input);
    String script=input.getString(InterpreterService.SCRIPT);

    if (script!=null) {
      SQLiteDatabase db=SQLiteDatabase.create(null);
      Cursor c=db.rawQuery(script, null);

      c.moveToFirst();

      for (int i=0;i<c.getColumnCount();i++) {
        result.putString(c.getColumnName(i), c.getString(i));
      }

      c.close();
      db.close();
    }

    return(result);
  }
}
```

This class accepts a script, in the form of a SQLite database query. It extracts the script from the Bundle, using a pre-defined key (InterpreterService.SCRIPT):

```
String script=input.getString(InterpreterService.SCRIPT);
```

Assuming there is such a script, it creates an empty in-memory database and executes the SQLite query against that database:

```
SQLiteDatabase db=SQLiteDatabase.create(null);
Cursor c=db.rawQuery(script, null);
```

The results come back in the form of a Cursor – itself a key-value store. SQLiteInterpreter takes those results and pours them into a Bundle to be returned:

```
c.moveToFirst();

for (int i=0;i<c.getColumnCount();i++) {
  result.putString(c.getColumnName(i), c.getString(i));
}

c.close();
db.close();
```

The `Bundle` being returned starts from a copy of the input `Bundle`, so the script requester can embed in the input `Bundle` any identifiers it needs to determine how to handle the results from executing this script.

`SQLiteInterpreter` is not terribly flexible, but you can use it for simple numeric and string calculations, such as the following script:

```
SELECT 1+2 AS result, 'foo' AS other_result, 3*8 AS third_result;
```

This would return a `Bundle` containing a key of `result` with a value of `3`, a key of `other_result` with a value of `foo`, and a key of `third_result` with a value of `24`.

Of course, it would be nice to support more compelling interpreters, and we will examine a pair of those later in this chapter.

Loading Interpreters and Executing Scripts

Of course, having a nice clean interface to the interpreters does nothing in terms of actually executing them on a background thread, let alone sandboxing them. The `InterpreterService` class itself handles that.

`InterpreterService` is an `IntentService`, which automatically routes incoming `Intent` objects (from calls to `startService()`) to a background thread via a call to `onHandleIntent()`. `IntentService` will queue up `Intent` objects if needed, and `IntentService` even automatically shuts down if there is no more work to be done.

Here is the implementation of `onHandleIntent()` from `InterpreterService`:

```
@Override
protected void onHandleIntent(Intent intent) {
  String action=intent.getAction();
  I_Interpreter interpreter=interpreters.get(action);

  if (interpreter==null) {
    try {
      interpreter=(I_Interpreter)Class.forName(action).newInstance();
      interpreters.put(action, interpreter);
    }
    catch (Throwable t) {
      Log.e("InterpreterService", "Error creating interpreter", t);
    }
  }

  if (interpreter==null) {
    failure(intent, "Could not create interpreter: "+intent.getAction());
  }
  else {
    try {
      success(intent, interpreter.executeScript(intent.getBundleExtra(BUNDLE)));
    }
    catch (Throwable t) {
      Log.e("InterpreterService", "Error executing script", t);

      try {
        failure(intent, t);
      }
      catch (Throwable t2) {
        Log.e("InterpreterService",
              "Error returning exception to client",
              t2);
      }
    }
  }
}
```

We keep a cache of interpreters, since initializing their engines may take some time. That cache is keyed by the interpreter's class name, and that key comes in to the service by way of the action on the Intent that was used to start the service. In other words, the script requester tells us, by way of the Intent used in startService(), which interpreter to use.

Those interpreters are created using reflection:

```
try {
  interpreter=(I_Interpreter)Class.forName(action).newInstance();
  interpreters.put(action, interpreter);
}
catch (Throwable t) {
```

```
   Log.e("InterpreterService", "Error creating interpreter", t);
}
```

This way, `InterpreterService` has no compile-time knowledge of any given interpreter class. Interpreters can come and go, but `InterpreterService` remains the same.

Assuming an interpreter was found (either cached or newly created), we have it execute the script, with the input `Bundle` coming from an "extra" on the `Intent`. Methods named `success()` and `failure()` are then responsible for getting the results to the script requester...as will be seen in the next section.

Delivering Results

Script requesters can get the results of the script back – in the form of the interpreter's output `Bundle` – in one of two ways.

One option is a private broadcast `Intent`. This is a broadcast `Intent` where the broadcast is limited to be delivered only to a specific package, not to any potential broadcast receiver on the device.

The other option is to supply a `PendingIntent` that will be sent with the results. This could be used by an `Activity` and `createPendingIntent()` to have control routed to its `onActivityResult()` method. Or, an arbitrary `PendingIntent` could be created, to start another activity, for example.

The implementations of `success()` and `failure()` in `InterpreterService` simply build up an `Intent` containing the results to be delivered:

```
private void success(Intent intent, Bundle result) {
   Intent data=new Intent();

   data.putExtras(result);
   data.putExtra(RESULT_CODE, SUCCESS);

   send(intent, data);
}
```

```
private void failure(Intent intent, String message) {
  Intent data=new Intent();

  data.putExtra(ERROR, message);
  data.putExtra(RESULT_CODE, FAILURE);

  send(intent, data);
}

private void failure(Intent intent, Throwable t) {
  Intent data=new Intent();

  data.putExtra(ERROR, t.getMessage());
  data.putExtra(TRACE, getStackTrace(t));
  data.putExtra(RESULT_CODE, FAILURE);

  send(intent, data);
}
```

These, in turn, delegate the actual sending logic to a `send()` method that delivers the result `Intent` via a private broadcast or a `PendingIntent`, as indicated by the script requester:

```
private void send(Intent intent, Intent data) {
  String broadcast=intent.getStringExtra(BROADCAST_ACTION);

  if (broadcast==null) {
    PendingIntent pi=(PendingIntent)intent.getParcelableExtra(PENDING_RESULT);

    if (pi!=null) {
      try {
        pi.send(this, Activity.RESULT_OK, data);
      }
      catch (PendingIntent.CanceledException e) {
        // no-op – client must be gone
      }
    }
  }
  else {
    data.setPackage(intent.getStringExtra(BROADCAST_PACKAGE));
    data.setAction(broadcast);

    sendBroadcast(data);
  }
}
```

Packaging the InterpreterService

There are three steps for integrating `InterpreterService` into an application.

First, you need to decide what APK the `InterpreterService` goes in – the main one for the application (no sandbox) or a separate low-permission one (sandbox).

Second, you need to decide what interpreters you wish to support, writing `I_Interpreter` implementations and getting the interpreters' JARs into the project's `libs/` directory.

Third, you need to add the source code for `InterpreterService` along with a suitable `<service>` entry in `AndroidManifest.xml`. This entry will need to support `<intent-filter>` elements for each scripting language you are supporting, such as:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.abj.interp"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <service android:exported="false"
             android:name=".InterpreterService">
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.EchoInterpreter" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.SQLiteInterpreter" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.BshInterpreter" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.RhinoInterpreter" />
      </intent-filter>
    </service>
  </application>
</manifest>
```

From there, it is a matter of adding in appropriate `startService()` calls to your application wherever you want to execute a script, and processing the results you get back.

Using the InterpreterService

To use the `InterpreterService`, you need to first determine which `I_Interpreter` engine you are using, as that forms the action for the `Intent` to be used with the `InterpreterService`. Create an `Intent` with that action, then add in an `InterpreterService.BUNDLE` extra for the script and other data to be supplied to the interpreter. Also, you can add an `InterpreterService.BROADCAST_ACTION`, to be used by `InterpreterService` to send results back to you via a broadcast `Intent`. Finally, call `startService()` on the `Intent`, and the results will be delivered to you asynchronously.

For example, here is a test method from the `EchoInterpreterTests` test case:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public class EchoInterpreterTests extends InterpreterTestCase {
  protected String getInterpreterName() {
    return("com.commonsware.abj.interp.EchoInterpreter");
  }

  public void testNoInput() {
    Bundle results=execServiceTest(new Bundle());

    assertNotNull(results);
    assert(results.size()==0);
  }

  public void testWithSomeInputJustForGrins() {
    Bundle input=new Bundle();

    input.putString("this", "is a value");

    Bundle results=execServiceTest(input);

    assertNotNull(results);
    assertEquals(results.getString("this"), "is a value");
  }
}
```

The echo "interpreter" simply echoes the input `Bundle` into the output. The `execServiceTest()` method is inherited from the `InterpreterTestCase` base class:

```
package com.commonsware.abj.interp;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.test.AndroidTestCase;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import junit.framework.TestCase;

abstract public class InterpreterTestCase extends AndroidTestCase {
  abstract protected String getInterpreterName();

  private static String ACTION="com.commonsware.abj.interp.InterpreterTestCase";
  private CountDownLatch latch=new CountDownLatch(1);
  private Bundle results=null;

  protected void setUp() throws Exception {
    super.setUp();

    getContext().registerReceiver(onBroadcast, new IntentFilter(ACTION));
  }

  protected void tearDown() {
    getContext().unregisterReceiver(onBroadcast);
  }

  protected Bundle execServiceTest(Bundle input) {
    Intent i=new Intent(getInterpreterName());

    i.putExtra(InterpreterService.BUNDLE, input);
    i.putExtra(InterpreterService.BROADCAST_ACTION, ACTION);

    getContext().startService(i);

    try {
      latch.await(5000, TimeUnit.MILLISECONDS);
    }
    catch (InterruptedException e) {
      // just keep rollin'
    }

    return(results);
  }

  private BroadcastReceiver onBroadcast=new BroadcastReceiver() {
    @Override
    public void onReceive(Context ctxt, Intent i) {
      results=i.getExtras();
      latch.countDown();
    }
```

```
  };
}
```

The `execServiceTest()` method uses a `CountDownLatch` to wait on the interpreter to do its work before proceeding (or 5000 milliseconds, whichever comes first). The broadcast `Intent` containing the results, registered to watch for `com.commonsware.abj.interp.InterpreterTestCase` broadcasts, stuffs the output `Bundle` in a results data member and drops the latch, allowing the main test thread to continue.

BeanShell on Android

What if Java itself were a scripting language? What if you could just execute a snippet of Java code, outside of any class or method? What if you could still import classes, call static methods on classes, create new objects, as well?

That was what BeanShell offered, back in its heyday. And, since BeanShell does not use sophisticated tricks with its interpreter – like JIT compilation of scripting code – BeanShell is fairly easy to integrate into Android.

What is BeanShell?

BeanShell is Java on Java.

With BeanShell, you can write scripts in loose Java syntax. Here, "loose" means:

- In addition to writing classes, you can execute Java statements outside of classes, in a classic imperative or scripting style

- Data types are optional for variables

- Not every language feature is supported, particularly things like annotations that did not arrive until Java 1.5

- Etc.

BeanShell was originally developed in the late 1990's by Pat Niemeyer. It enjoyed a fair amount of success, even being considered as a standard interpreter to ship with Java (JSR-274). However, shortly thereafter, BeanShell lost momentum, and it is no longer being actively maintained. That being said, it works quite nicely on Android...once a few minor packaging issues are taken care of.

Getting BeanShell Working on Android

BeanShell has two main problems when it comes to Android:

1. The publicly-downloadable JAR was compiled for Java 1.4.2, and Android requires Java 5 or newer

2. The source code includes various things, like a Swing-based GUI and a servlet, that have no real place in an Android app and require classes that Android lacks

Fortunately, with BeanShell being open source, it is easy enough to overcome these challenges. You could download the source into an Android library project, then remove the classes that are not necessary (e.g., the servlet), and use that library project in your main application. Or, you could use an Android project for creating a JAR file that was compiled against the Android class library, so you are certain everything is supported.

However, the easiest answer is to use SL4A's BeanShell JAR, since they have solved those problems already. The JAR can be found in the SL4A source code repository, though you will probably need to check out the project using Mercurial, since JARs cannot readily be downloaded from the Google Code Web site.

Integrating BeanShell

The BeanShell engine is found in the `bsh.Interpreter` class. Wrapping one of these in an `I_Interpreter` interface, for use with `InterpreterService`, is fairly simple:

569

```
package com.commonsware.abj.interp;

import android.os.Bundle;
import bsh.Interpreter;

public class BshInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    Interpreter i=new Interpreter();
    Bundle output=new Bundle(input);
    String script=input.getString(InterpreterService.SCRIPT);

    if (script!=null) {
      try {
        i.set(InterpreterService.BUNDLE, input);
        i.set(InterpreterService.RESULT, output);

        Object eval_result=i.eval(script);

        output.putString("result", eval_result.toString());
      }
      catch (Throwable t) {
        output.putString("error", t.getMessage());
      }
    }

    return(output);
  }
}
```

BeanShell interpreters are fairly inexpensive objects, so we create a fresh
Interpreter for each script, so one script cannot somehow access results
from prior scripts. After setting up the output Bundle and extracting the
script from the input Bundle, we inject both Bundle objects into BeanShell
itself, where they can be accessed like global variables, named _bundle and
_result.

At this point, we evaluate the script, using the eval() method on the
Interpreter object. If all goes well, we convert the object returned by the
script into a String and tuck it into the output Bundle, alongside anything
else the script may have put into the Bundle. If there is a problem, such as a
syntax error in the script, we put the error message into the output Bundle.

So long as the InterpreterService has an <intent-filter> for the
com.commonsware.abj.interp.BshInterpreter action, and so long as we have a
BeanShell JAR in the project's libs/ directory, InterpreterService is now
capable of executing BeanShell scripts as needed.

For example, here are a couple of test methods from the `BshInterpreterTests` test case:

```java
public void testSimpleResult() {
   Bundle input=new Bundle();

   input.putString(InterpreterService.SCRIPT, "1+2");

   Bundle output=execServiceTest(input);

   assertNull(output.getString("error"));
   assert(output.size()==2);
   assertEquals(output.getString("result"), "3");
}

public void testComplexResult() {
   Bundle input=new Bundle();

   input.putString(InterpreterService.SCRIPT, "_result.putInt(\"foo\", 1+2);");

   Bundle output=execServiceTest(input);

   assertNull(output.getString("error"));
   assert(output.size()==3);
   assertEquals(output.getInt("foo"), 3);
}
```

With our inherited `execServiceTest()` method handling invoking the `InterpreterService` and waiting for responses, we can "simply" put our script as the `InterpreterService.SCRIPT` value in the input `Bundle`, and see what we get out. The first test script returns a simple value; the second test script directly calls methods on the output `Bundle` to return its results.

Rhino on Android

Javascript arrived on the language scene hot on the heels of Java itself. The name was chosen for marketing purposes more so than for any technical reason. Java and Javascript had little to do with one another, other than both adding interactivity to Web browsers. And while Java has largely faded from mainstream browser usage, Javascript has become more and more of a force on the browser, and even now on Web servers.

And, along the way, the Mozilla project put Javascript on Java and gave us Rhino.

What is Rhino?

If BeanShell is Java in Java, Rhino is Javascript in Java.

As part of Netscape's failed "Javagator" attempt to create a Web browser in Java, they created a Javascript interpreter for Java, code-named Rhino after the cover of O'Reilly Media's JavaScript: The Definitive Guide. Eventually, Rhino was made available to the Mozilla Foundation, which has continued maintaining it. At the present time, Rhino implements Javascript 1.7, so it does not support the latest and greatest Javascript capabilities, but it is still fairly full-featured.

Interest in Rhino has ticked upwards, courtesy of interest in using Javascript in places other than Web browsers, such as server-side frameworks. And, of course, it works nicely with Android.

Getting Rhino Working on Android

Similar to BeanShell, Rhino has a few minor source-level incompatibilities with Android. However, these can be readily pruned out, leaving you with a still-functional Javascript interpreter. However, once again, it is easiest to use SL4A's Rhino JAR, since all that work is done for you.

Integrating Rhino

Putting an `I_Interpreter` facade on Rhino is incrementally more difficult than it is for BeanShell, but not by that much:

```
package com.commonsware.abj.interp;

import android.os.Bundle;
import org.mozilla.javascript.*;

public class RhinoInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    String script=input.getString(InterpreterService.SCRIPT);
    Bundle output=new Bundle(input);

    if (script!=null) {
```

```
    Context ctxt=Context.enter();

    try {
      ctxt.setOptimizationLevel(-1);

      Scriptable scope=ctxt.initStandardObjects();
      Object jsBundle=Context.javaToJS(input, scope);
      ScriptableObject.putProperty(scope, InterpreterService.BUNDLE,
jsBundle);

      jsBundle=Context.javaToJS(output, scope);
      ScriptableObject.putProperty(scope, InterpreterService.RESULT,
jsBundle);
      String result=Context.toString(ctxt.evaluateString(scope, script,
                                                 "<script>", 1,
                                                 null));

      output.putString("result", result);
    }
    finally {
      Context.exit();
    }
  }

  return(output);
  }
}
```

As with `BshInterpreter`, `RhinoInterpreter` sets up the output `Bundle` and extracts the script from the input `Bundle`. Assuming there is a script, `RhinoInterpreter` then sets up a Rhino `Context` object, which is roughly analogous to the BeanShell `Intepreter` object. One key difference is that you need to clean up the `Context`, by calling a static `exit()` method on the `Context` class, whereas with a BeanShell `Interpreter`, you just let garbage collection deal with it.

Rhino has a JIT compiler, one that unfortunately will not work with Android, since it generates Java bytecode, not Dalvik bytecode. However, Rhino lets you turn that off, by calling `setOptimizationLevel()` on the `Context` object with a value of `-1` (meaning, in effect, disable all optimizations).

After that, we:

- Create a language scope for our script and inject standard Javascript global objects into that scope

---

- Wrap our two `Bundle` objects with Javascript proxies via calls to `javaToJS()`, then injecting those objects into the scope as `_bundle` and `_result` via `putProperty()` calls

- Execute the script via a call to `evaluateString()` on the `Context` object, converting the resulting object into a `String` and pouring it into the output `Bundle`

If our `InterpreterService` has an `<intent-filter>` for the `com.commonsware.abj.interp.RhinoInterpreter` action, and so long as we have a Rhino JAR in the project's `libs/` directory, `InterpreterService` can now invoke Javascript.

For example, here are equivalent test methods from `RhinoInterpreterTests` to the ones shown above for `BshInterpreterTests` and BeanShell:

```
public void testSimpleResult() {
   Bundle input=new Bundle();

   input.putString(InterpreterService.SCRIPT, "1+2");

   Bundle output=execServiceTest(input);

   assertNull(output.getString("error"));
   assert(output.size()==2);
   assertEquals(output.getString("result"), "3");
}
public void testComplexResult() {
   Bundle input=new Bundle();

   input.putString(InterpreterService.SCRIPT, "_result.putInt(\"foo\", 1+2);");

   Bundle output=execServiceTest(input);

   assertNull(output.getString("error"));
   assert(output.size()==3);
   assertEquals(output.getInt("foo"), 3);
}
```

Other JVM Scripting Languages

As mentioned previously, there are many languages that, themselves, are implemented in Java and can be ported to Android, with varying degrees of

difficulty. Many of these languages are fairly esoteric. Some, like JRuby, have evolved to the point where they transcend a simple "scripting language" on Android.

However, there are two other languages worth mentioning, as they are fairly well-known in Java circles: Groovy and Jython.

Groovy

Groovy is perhaps the most popular Java-based language that does not have its roots in some other previous language (Java, Javascript, Python, etc.). Designed in some respects to be a "better Java than Java", Groovy gives you access to Java classes while allowing you to write scripts with dynamic typing, closures, and so forth. Groovy has an extensive community, complete with a fair number of Groovy-specific libraries and frameworks, plus some books on the market.

At the time of this writing, it does not appear that Groovy has been successfully ported to work on Android, though.

Jython

Jython is an implementation of a Python language interpreter in Java. It has been around for quite some time, and gives you Python syntax with access to standard Java classes where needed. While the Jython community is not as well-organized as that of Groovy, there are plenty of books covering the use of Jython.

Jython's momentum has flagged a bit in recent months, in part due to Sun's waning interest in the technology and the departure of Sun employees from the project. One attempt to get Jython working with Android has been shut down, with people steered towards SL4A. It is unclear if others will make subsequent attempts.

PART V – Advanced Development

Reusable Components

In the world of Java outside of Android, reusable components rule the roost. Whether they are simple JARs, are tied in via inversion-of-control (IoC) containers like Spring, or rely on enterprise service buses like Mule, reusable Java components are a huge portion of the overall Java ecosystem. Even full-fledged applications, like Eclipse or NetBeans, are frequently made up of a number of inter-locking components, many of which are available for others to use in their own applications.

In an ideal world, Android will evolve similarly, particularly given its reliance upon the Java programming language. This begs the question: what are the best ways to package code into a reusable component? Or, perhaps more basic: what are the possibilities for making reusable components?

Pick Up a JAR

A Java JAR is simplicity incarnate: a ZIP archive of classes compiled to bytecode. While the JAR as a packaging method is imperfect – dealing with dependencies can be no fun – it is still a very easy way to bundle Java logic into a discrete item that can be uploaded, downloaded, installed, integrated, and used.

Android introduces a seemingly vast number of challenges, though.

579

The JAR Itself

Packaging up a set of Java code into a JAR is very straightforward, even if that Java code refers to Android APIs. Whether you use the `jar` command directly, the `<jar>` task in an Ant script, or the equivalent in Eclipse, you just package up the class files as you normally would.

For example, here is an Ant task that creates a JAR for an Android project:

```
<target name="jar" depends="compile">
    <jar
        destfile="bin/CWAC-MergeAdapter.jar"
        basedir="bin/classes"
    />
</target>
```

To create a project that targets a JAR file, just create a regular Android project (e.g., `android create project` or the Eclipse new-project wizard), but ignore the options to build an APK. Just compile the code and put it in the JAR.

Note that the JAR will contain Java class files, meaning Java bytecode. The reuser of your JAR will put your JAR into their project (e.g., in the `libs/` directory), and their project will convert your JAR'd classes into Dalvik bytecode as part of building their APK.

Resources

The JAR can take care of your Java code. And if all you need is Java code, reuse via JAR file is extremely easy.

Android code often uses other things outside of Java code, and that is where the problems crop up. The most prominent of these "other things" are resources: layouts, bitmaps, menus, preferences, custom view attributes, etc.

Android projects expect these resources to be in the project's own `res/` directory, and there is no facility to get these resources from anywhere else. That causes some problems.

Packaging and Installing

First, you are going to need to package up the resources you want to ship and to distribute them along with your JAR. You could try to package them in the JAR itself, or you could put the JAR and resources into some larger container, like a ZIP file.

The people reusing your code will need to not only add the JAR to their projects, but also unpack those shipped resources (in their respective resource sets) and put them in their projects as well.

Naming

The act of unpacking those resources and putting them in a project leads to potential naming conflicts. If you have a layout named `main.xml` and the project already has a layout named `main.xml`, somebody loses.

Hence, when you write your reusable code, you will want to adopt a naming convention that "ensures" all your resource names are going to be unique. Of course, you have no true way of absolutely ensuring that they will be unique, but you can substantially increase your odds. One approach is to prefix all names with something distinctive for your project.

Note that the "names" will be filenames for file-based resources (layouts, drawables, etc.) and the values of `android:name` attributes for element-based resources (strings, dimensions, colors, etc.).

Also note that `android:id` values do not have to be unique, as Android is already set up to support the notion of multiple distinct uses of an ID.

ID Lookup

Complicating matters further is that even if your build process generates an R.java file, the resource identifiers encoded in that file will be different in your project than in the reuser's project. Hence, you cannot refer to resources via R. references, like you would in a regular Android application.

If all your resources have simple integer identifiers, you can use the getIdentifier() method on the Resources class to convert between a string representation of the resource identifier and the actual identifier itself. This uses reflection, and so is not fast. You should strongly consider caching these values to minimize the number of reflection calls.

However, at least one type of resource does not have a simple integer resource identifier – custom view attributes. R.styleable.foo is an int[], not an int. getIdentifier() will only work with an integer resource identifier. Your alternative is to do the reflection directly, or find some existing code that will handle that for you, so you can get at the int[] that you need.

Customizing and Overriding

Bear in mind that the reuser of your project may wish to change some things. Perhaps your bitmaps clash with their desired color scheme. Perhaps you did not ship string resources in all desired translations. Perhaps your context menu needs some more items.

There are two ways you can support such modifications. One is to tell the reusers to modify their copy of the resources they unpacked into their projects. This has the advantage of not requiring any particular code changes on your part. However, it may make support more difficult – perhaps some of the modifications they make accidentally break things, and you may have a tough time answering questions about a modified code base.

The alternative is for you to support setters, custom view attributes, or similar means for reusers to supply their own resource identifiers for you to use. Where they give you one, use it; where they do not, use the resource you shipped. This adds to your project's code but may offer a cleaner customization model for reusers.

Assets

Assets – files in `assets/` in an Android project – will have some of the same issues as do resources:

- You need to package and distribute those assets

- Reusers need to unpack those assets into their projects

- You have to take steps to prevent name collisions (e.g., use a directory in `assets/` likely to be unique to your project)

- Potentially, reusers may want to use a different asset than the one you shipped

Since assets are accessed by a string name, rather than a generated resource ID, at least you do not have to worry about that particular issue, as you would with a raw resource.

Manifest Entries

If your reusable code ships activities, services, content providers, or broadcast receivers, reusers of your code will need to add entries for your components in their `AndroidManifest.xml` file. Similarly, if you require certain permissions, or certain hardware features, you will have other manifest entries (e.g., `<uses-permission>`) that will be needed in a reusing project's manifest.

You can handle this by supplying a sample manifest and providing instructions for what pieces of it need to be merged into the reuser's own manifest.

AIDL Interfaces

If you are shipping a `Service` in your JAR, and if that `Service` is supposed to allow remote access via AIDL, you will need to ship the AIDL file(s) with the JAR itself. Those files will be needed by consumers of the `Service`, even if the developer integrating the JAR itself might not need those files.

This pattern – a JAR containing a remote `Service` – is probably going to be unusual. More likely, a remote `Service` will be packaged as part of an application in an APK file, rather than via a JAR.

Permissions

Your code may require certain Android permissions in order to succeed, such as needing `WAKE_LOCK` to use a `WakeLock`, or needing `INTERNET`, or whatever. Unfortunately, you cannot specify permissions in a JAR file, so you will need to make sure that reusers of your JAR correctly add the permissions you require, or find ways to gracefully degrade what you do when those permissions are missing.

You can see if the hosting project requested your permission by using `checkPermission()` on `PackageManager`:

```
int result=getPackageManager()
           .checkPermission("android.permission.WAKE_LOCK",
                            getPackageName());

if (PackageManager.PERMISSION_DENIED==result) {
  // do something
}
```

If it did not, what you do is up to the way your API is designed and how you want to handle such problems:

- You could throw a `RuntimeException`. Since developers will encounter this problem during development, this should not harm their production application.

- You could return `false` or `null` or some other "didn't work" return value from a method. For example, you could design an API that allows developers to check if a certain feature is available, then return `false` from that method.

- You could ignore the problem and let the Android-generated `RuntimeException` handle it. However, this may not be as friendly to your reusers as might throwing your own `RuntimeException`.

- You could throw a regular checked exception if you prefer (e.g., a custom `PermissionMissingException`), though that requires extra `try/catch` blocks in the reuser's code for what should only be a configuration error in their project's manifest.

Other Source Code

You may have Java source beyond the actual reusable classes themselves, such as sample code demonstrating how to reuse the JAR and related files. You will need to consider how you wish to distribute this code, as part of the actual component package (e.g., ZIP) or via separate means (e.g., `git` repository).

Your API

Your reusable code should be exposing an API for reusing projects to call. Most times, if you are packaging code as a JAR, that API will be in the form of Java classes and methods.

Public versus Non-Public

Those classes and methods will need to be public, as you want the reusing project to reside in its own Java package, not yours.

This means that your black-box test suite (if you have one) and sample code (if you offer any) really should be in separate Java packages as well, so you test and demonstrate the public API. Otherwise, you may accidentally access package-protected classes and methods.

Flexibility versus Maintainability

As with any body of reusable code, you are going to have to consider how much you want to actually implement. The more features and options you provide, the more flexible your reusable code will be for reusers. However, the more features and options you provide, the more complex your reusable code becomes, increasing maintainability costs over time.

This is particularly important when it comes to the public API. Ideally, your public API expands in future releases but does not eliminate or alter the API that came before it. Otherwise, when you ship an updated JAR, your reusers' projects will break, making them unhappy with you and your code.

Documentation

If you are expecting people to reuse your code, you are going to have to tell them how to do that. Usually, these sorts of packages ship documentation with them, sometimes a clone of what is available online. That way, developers can choose the local or hosted edition of the documentation as they wish.

Note that generated documentation (e.g., Javadocs) may still need to be shipped or otherwise supplied to reusers, if you are not providing the source code in the package. Without the source code, reusers cannot regenerate the Javadocs.

Licensing

Your reusable code should be accompanied by adequate licensing information.

Your License

The first license you should worry about is your own. Is your component open source? If so, you will want to ship a license file containing those

terms. If your component is not open source, make sure there is a license agreement shipped with the component that lets the reuser know the terms of use.

Bear in mind that not all of your code necessarily has to have the same license. For example, you might have a proprietary license for the component itself, but have sample code be licensed under Apache License 2.0 for easy copy-and-paste.

Third-Party License Impacts

You may need to include licenses for third party libraries that you have to ship along with your own JAR. Obviously, those licenses would need to give you redistribution rights – otherwise, you cannot ship those libraries in the first place.

Sometimes, the third party licenses will impact your project more directly, such as:

- Incorporating a GPL library may require your project to be licensed under the same license

- Adding support for Facebook data may require you to limit your API or require reusers to supply API access keys, since you probably do not have rights to redistribute Facebook data

A Private Library

The "r6" version of the Android SDK introduced the "library project". This offers a form of reuse, to share a chunk of code between projects. It is specifically aimed at developers or teams creating multiple applications from the same code base. Perhaps the most popular occurrence of this pattern is the "paid/free" application pair: two applications, one offered for free, one with richer functionality that requires a payment. Via a library project, the common portions of those two applications can be consolidated, even if those "common portions" include things like resources.

The library project support is integrated into Eclipse, though you can create library projects for use via Ant as well.

Creating a Library Project

An Android library project, in many respects, looks like a regular Android project. It has source code and resources. It has a manifest. It supports third-party JAR files (e.g., `libs/`).

What it does not do, though, is build an APK file. Instead, it represents a basket of programming assets that the Android build tools know how to blend in with a regular Android projects.

To create a library project in Eclipse, start by creating a normal Android project. Then, in the project properties window (e.g., right-click on the project and choose Properties), in the Android area, check the "Is Library" checkbox. Click [Apply], and you are done.

To create a library project for use with Ant, you can use the `android create lib-project` command. This has the net effect of putting an `android.library=true` entry in your project's `default.properties` file.

Using a Library Project

Once you have a library project, you can attach it to a regular Android project, so the regular Android project has access to everything in the library.

To do this in Eclipse, go into the project properties window (e.g., right-click on the project and choose Properties). Click on the Android entry in the list on the left, then click the [Add] button in the Library area. This will let you browse to the directory where your library project resides. You can add multiple libraries and control their ordering with the [Up] and [Down] buttons, or remove a library with the [Remove] button.

For developing using Ant, you can use `android update lib-project` command. This adds an entry like `android.library.reference.1=...` to your project's `default.properties` file, where `...` is the relative path to your library project. You can add several such libraries, controlling their ordering via the numeric suffix at the end of each property name (e.g., `1` in the previous example).

Now, if you build the main project, the Android build tools will:

- Include the `src/` directories of the main project and all of the libraries in the source being compiled.

- Include all of the resources of the projects, with the caveat that if more than one project defines the same resource (e.g., `res/layout/main.xml`), the highest priority project's resource is included. The main project is top priority, and the priority of the remainder are determined by their order as defined in Eclipse or `default.properties`.

This means you can safely reference `R.` constants (e.g., `R.layout.main`) in your library source code, as at compile time it will use the value from the main project's generated `R` class(es).

Limitations of Library Projects

While library projects are useful for code organization and reuse, they do have their limits, such as:

- As noted above, if more than one project (main plus libraries) defines the same resource, the higher-priority project's copy gets used. Generally, that is a good thing, as it means that the main project can replace resources defined by a library (e.g., change icons). However, it does mean that two libraries might collide. It is important to keep your resource names distinct, a concept touched upon in greater detail later in this chapter.

- While you can define entries in the manifest file for a library, at present, they do not appear to be used.

- AIDL files defined in a library will not be picked up by the main project.

- While resources from libraries are put into the main project's APK, assets defined in a library's `assets/` directory are not.

- One library cannot depend on another library. You can either produce or consume a library, but not both.

Picking Up a Parcel

The author of this book has also started The Android Parcel Project, a set of conventions and tools to help create reusable Android components. The goal is to make it just a bit easier to create an Android library project that can successfully be reused by third party developers, once distributed in some form.

Binary-Only Library Projects

Android library projects are designed for distributing source code. That may or may not be palatable in all cases.

You can create a binary-only library project via the following steps:

1. Create an Android library project, with your source code and such – this is your master project, from which you will create a version of the library project for distribution

2. Compile the Java source (e.g., `ant compile`) and turn it into a JAR file

3. Create a distribution Android library project, with the same resources as the master library project, but no source code

4. Put the JAR file in the distribution Android library project's `libs/` directory

The resulting distribution Android library project will have everything a main project will need, just without the source code.

Note that if you use resources, you will need to take extra steps with a binary-only library project to deal with the resource identifiers, a topic covered in the next section.

Resource Naming Conventions

As mentioned previously, resources in multiple libraries might collide with one another, particularly for obvious names (e.g., `res/layout/main.xml`, the `app_name` string resource). You need to take steps to help minimize the odds of this occurring with your reusable component.

Resource Name Prefixes

The simple answer is to append a prefix to the front of all resource names. Here, "resource names" refers to:

- The filenames of resources where the resource itself is a file (e.g., the filename of a layout resource)
- The names of resources where more than one resource resides in an XML file (e.g., the name attribute of a string resource)

Bear in mind the resource naming limits (letters, numbers, underscores; cannot start with a number).

Hence, instead of `res/layout/main.xml`, you might have `res/layout/cwac_touchlist_main.xml`.

Ideally, we would use a prefix that is based on package-style naming conventions, for minimal chance of collision. However, `com_commonsware_cwac_touchlist_main.xml` would be painful to type. Also, since the "collision space" is merely the world of reusable libraries, not the Android Market, feel free to choose something shorter that is unlikely to be taken by anyone else.

Also, you do not need to prefix `android:id` values (e.g., in layout files), as Android assumes that there may be multiple definitions of those values.

Eventually, the Android Parcel Project will supply a `lint`-style utility to help you validate that you are applying a prefix for all required resources.

Runtime Resource ID Lookups

For library projects distributed with full source code, just applying the prefix to all resources is sufficient.

However, if you are distributing a binary-only library project, you will run into a problem. Your library-compiled code will use a generated ID that will differ, in all likelihood, from the ID generated by the main project from the combined set of resources. Hence, if you simply use `R.layout.cwac_touchlist_main` (or the like) in your code, you will wind up with missing or invalid resources at runtime – the value that the `javac` compiler inlined in your class will be the wrong value.

Instead, we need to look up those resource IDs at runtime. There are two ways to do this:

1. Use the `getIdentifier()` method on a `Resources` object, typically obtained via `getResources()` on some `Context`. This uses reflection under the covers, and since that can be slow, it is a really good idea to cache the results of these lookups. And, `getIdentifier()` does not support all types of identifiers – notably, it does not work with custom attributes in custom widgets.

2. Use the `ParcelHelper` class supplied by the Android Parcel Project.

This is distributed in a plain JAR file (`CWAC-Parcel.jar`), available from a GitHub repository.

You need to create an instance of `ParcelHelper`, supplying your prefix and a `Context`. Then, when you need a resource ID, you can call methods like:

- `getLayoutId()`
- `getMenuId()`
- `getDrawableId()`

- and so forth

These lookups are a bit expensive, since they involve reflection. Hence, ParcelHelper caches them on your behalf, to improve performance.

For example, here is some code to initialize a ParcelHelper, then inflate a layout named main (i.e., res/layout/cwac_colormixer_main.xml):

```
parcel=new ParcelHelper("cwac-colormixer", getContext());

((Activity)getContext())
    .getLayoutInflater()
    .inflate(parcel.getLayoutId("main"), this, true);
```

(note that, for backwards compatibility with a previous edition of the Android Parcel Project, you can supply a dash in a prefix, which will be converted to an underscore automatically)

Parcel Distribution

Android library projects designed to be reusable components (parcels) can just be packaged as ZIP files for distribution, much as Google did with the License Validation Library. The more immediate question is, what should be distributed?

- You will need to decide whether to distribute source code or go the binary-only library project route

- You may wish to consider having some sort of sample project included in the distribution (e.g., a demo/ subproject referencing the parent project as a library)

- You should have some sort of license (e.g., LICENSE file with Apache License 2.0 terms), so developers know the "rules of the game" for reusing your component

- You may wish to include documentation, or perhaps that can just be on a Web site

- Be sure to include any dependent JARs (e.g., in the library project's `libs/` directory), or ensure potential reusers know where to get the JARs your code requires

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 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

From the command line, you use `android create project` to create a regular Android project. To create a project designed to test another project – what we will call a "test project" – you use the `android create test-project` command. From Eclipse, you can create a test project using the appropriate wizard. You will need to tell it which project to test, where you want the test project to reside, etc.

An Android test project is complete set of Android project artifacts: manifest, source directories, resources, etc. Much of its structure is

identical to a regular test project. In fact, the generated 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. For example, the `Contacts/Spinners` project has a `tests/` subdirectory containing a test project set up to test various facets of the `Spinners` application.

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.contacts.spinners` 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.contacts.spinners.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.contacts.spinners;

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.contacts.spinners.ContactsDemoTest \
 *
com.commonsware.android.contacts.spinners.tests/android.test.InstrumentationTest
Runner
 */
public class ContactsDemoTest extends
ActivityInstrumentationTestCase<ContactSpinners> {

    public ContactsDemoTest() {
        super("com.commonsware.android.contacts.spinners",
ContactSpinners.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.contacts.spinners) 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.contacts.spinners.ContactsDemoTest -w
com.commonsware.android.contacts.spinners.tests/android.test.InstrumentationTest
Runner
```

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.contacts.spinners`, 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`:

```java
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.contacts.spinners).

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.

Getting Ready for Production

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 get your application ready to be distributed in a production form, through the Android Market and elsewhere.

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

607

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 114. 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 115. 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 116. 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 117. 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
```

612

```
      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>
```

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.

Also, bear in mind that your package name must be unique across all applications on the Android Market, should you choose to distribute that way.

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.

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.

Keyword Index

623

Command.....................................

629

631

Property...