Figure 5. The MeterDemo application

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

Change of State

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

Changing Button Backgrounds

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

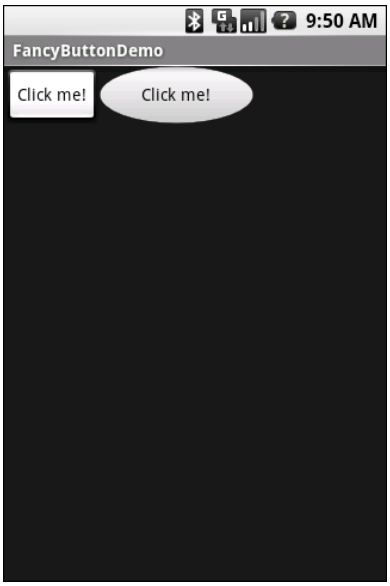

Figure 6. The FancyButton application, showing a normal oval-shaped button

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

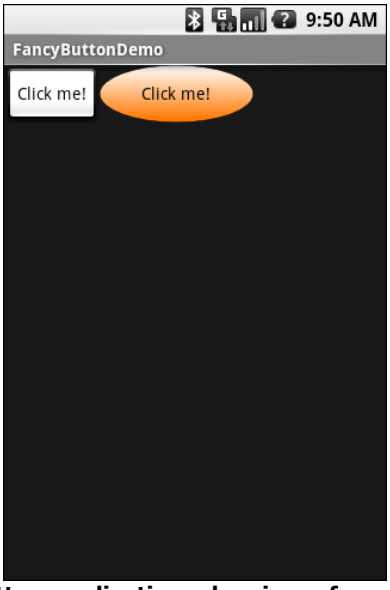

Figure 7. The FancyButton application, showing a focused oval-shaped button

...and it needs to be clickable:

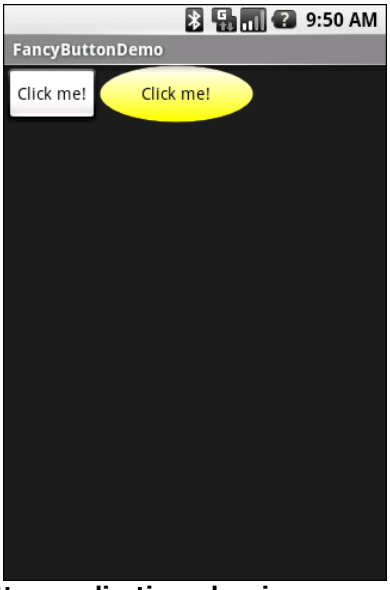

Figure 8. The FancyButton application, showing a pressed oval-shaped button

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

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

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

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

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

There are four states being described in this selector:

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

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
>
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click me!"
  />
  <Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

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

Changing CheckBox States

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

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

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

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

    <!-- Enabled states -->

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

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

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

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

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


    <!-- Disabled states -->

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

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

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

</selector>
```

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

Figure 9. An example of a zoomed CheckBox image

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

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

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

This gives us a look like this:

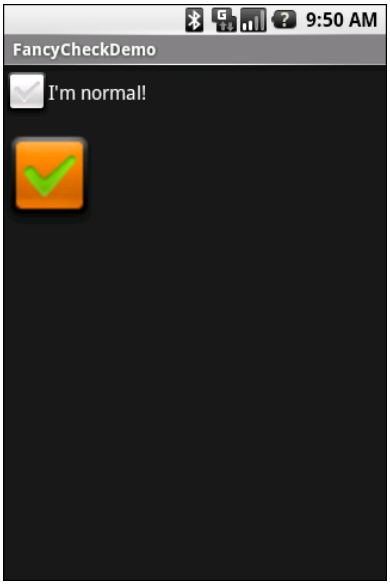

Figure 10. The FancyCheck application, showing a focused and checked CheckBox

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

CHAPTER 3

More Fun With ListViews

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

Giant Economy-Size Dividers

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

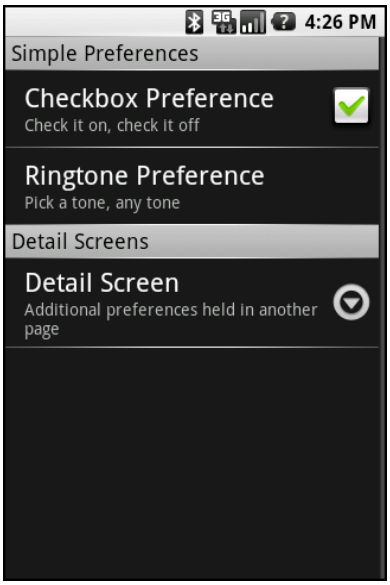

Figure 11. A PreferenceScreen UI

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

If so, you would have been wrong.

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

Choosing What Is Selectable

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

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

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

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

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

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

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

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

Composition for Sections

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

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

The pattern is fairly simple:

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

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

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

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

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

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

import android.view.View;
```

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

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

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

  public SectionedAdapter() {
    super();
  }

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

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

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

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

      position-=size;
    }

    return(null);
  }

  public int getCount() {
    int total=0;

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

    return(total);
  }

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

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

    return(total);
  }

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

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

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

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

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

    return(-1);
  }

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

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

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

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

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

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

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

    return(null);
  }

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

  class Section {
    String caption;
    Adapter adapter;

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

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

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

One thing that `SectionedAdapter` needs to do, though, is ensure that the pool of section header `View` objects is recycled separately from each section's own pool of `View` objects. To do this, `SectionedAdapter` takes advantage of `getViewTypeCount()`, by returning the total number of distinct types of `View` objects from all section `Adapter`s plus one for its own header `View` pool. Similarly, `getItemViewType()` considers the `0`th `View` type to be the header `View` pool, with the pools for each `Adapter` in sequence starting from 1. This pattern requires that each section `Adapter` have its `View` type numbers

starting from 0 and incrementing by 1, but most `Adapter` classes only use one `View` type and do not even implement their own `getViewTypeCount()` or `getItemViewType()`, so this will work most of the time.

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

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

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

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

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

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

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

    Collections.shuffle(list);

    adapter.addSection("Shuffled",
                       new ArrayAdapter<String>(this,
```

```
                        android.R.layout.simple_list_item_1,
                        list));

    list=Arrays.asList(items);

    Collections.shuffle(list);

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

    setListAdapter(adapter);
  }

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

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

      result.setText(caption);

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

The result is much as you might expect:

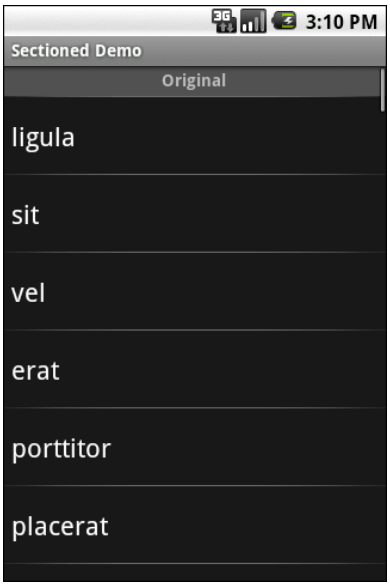

Figure 12. A ListView using a SectionedAdapter, showing one header and part of a list

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

From Head To Toe

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

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

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

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

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

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

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

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

    areWeDeadYet.set(true);
  }

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

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

        Collections.shuffle(list);

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

    return(btn);
  }

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

    updateFooter(txt);

    return(txt);
  }

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

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

    if (!areWeDeadYet.get()) {
      handler.postDelayed(new Runnable() {
        public void run() {

          updateFooter(txt);
        }
      }, 1000);
    }
  }
}
```

Here, we add a header `View` built via `buildHeader()`, returning a `Button` that, when clicked, will shuffle the contents of the list. We also add a footer `View` built via `buildFooter()`, returning a `TextView` that shows how long the activity has been running, updated every second. The list itself is the ever-popular list of *lorem ipsum* words.

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

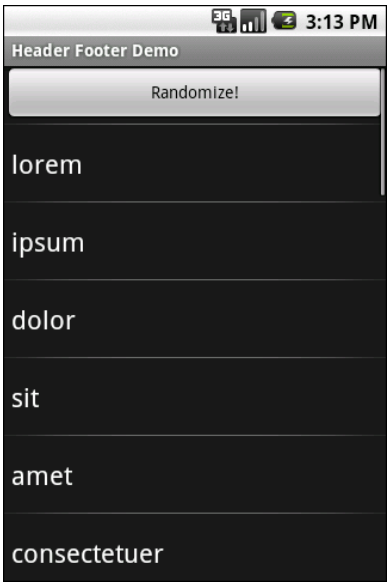

Figure 13. A ListView with a header view shown

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

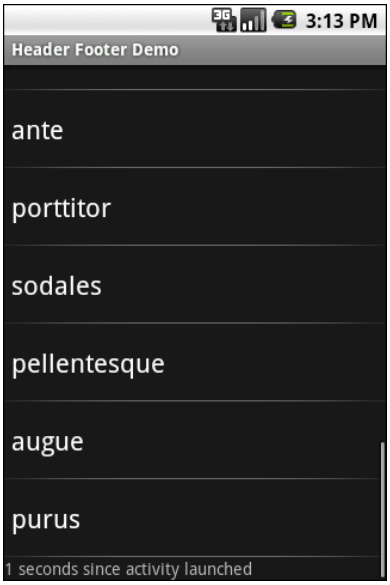

Figure 14. A ListView with a footer view shown

Control Your Selection

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

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

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

Create a Unified Row View

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

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

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

```
  android:layout_width="fill_parent"
  android:layout_height="fill_parent" >
  <View
    android:id="@+id/bar"
    android:background="#FFFF0000"
    android:layout_width="5px"
    android:layout_height="fill_parent"
    android:visibility="invisible"
  />
  <TextView
    android:id="@+id/label"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:textSize="10pt"
    android:paddingTop="2px"
    android:paddingBottom="2px"
    android:paddingLeft="5px"
  />
</LinearLayout>
```

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

Configure the List, Get Control on Selection

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

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

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

```
import android.widget.TextView;

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

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

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

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

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

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

      return(convertView);
    }
  }

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

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

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

      return(label);
    }

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

      return(bar);
    }
  }

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

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

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

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

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

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

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

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

Change the Row

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

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<ListView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:listSelector="#00000000"
/>
```

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

Figure 15. 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.

CHAPTER 4

Show Up At Home

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

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

East is East, and West is West...

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

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

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

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

The Big Picture for a Small App Widget

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

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

`RemoteViews` encapsulates a limited edition of normal widgets, in such a fashion that the `RemoteViews` can be "easily" transported across process boundaries. You configure the `RemoteViews` via your `BroadcastReceiver` and make those `RemoteViews` available to Android. Android in turn delivers the `RemoteViews` to the app widget host (usually the home screen), which renders them to the screen itself.

This architectural choice has many impacts:

1. You do not have access to the full range of widgets and containers. You can use `FrameLayout`, `LinearLayout`, and `RelativeLayout` for containers, and `AnalogClock`, `Button`, `Chronometer`, `ImageButton`, `ImageView`, `ProgressBar`, and `TextView` for widgets.

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

Crafting App Widgets

This will become somewhat easier to understand in the context of some sample code. In the `AppWidget/TwitterWidget` project, you will find an app widget that shows the latest tweet in your Twitter timeline. If you have read *Android Programming Tutorials*, you will recognize the JTwitter JAR we will use for accessing the Twitter Web service.

The Manifest

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

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

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

- Our `<receiver>` has `android:label` and `android:icon` attributes, which are not normally needed on `BroadcastReceiver` declarations. However, in this case, those are used for the entry that goes in the menu of available widgets to add to the home screen. Hence, you will probably want to supply values for both of those, and use appropriate resources in case you want translations for other languages.

- Our `<receiver>` has an `<intent-filter>` for the `android.appwidget.action.APPWIDGET_UPDATE` action. This means we will get control whenever Android wants us to update the content of our app widget. There may be other actions we want to monitor – more on this in a later section.
- Our `<receiver>` also has a `<meta-data>` element, indicating that its `android.appwidget.provider` details can be found in the `res/xml/widget_provider.xml` file. This metadata is described in the next section.
- Our `<activity>` has two `<intent-filter>` elements, the normal "put me in the Launcher" one and one looking for an action of `android.appwidget.action.APPWIDGET_CONFIGURE`.

The Metadata

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

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

Here, we provide four pieces of information:

- The minimum width and height of the app widget (`android:minWidth` and `android:minHeight`). These are approximate – the app widget host (e.g., home screen) will tend to convert these values into "cells" based upon the overall layout of the UI where the app widgets will reside. However, they should be no smaller than the minimums cited here.
- The frequency in which Android should request an update of the widget's contents (`android:updatePeriodMillis`). This is expressed in terms of milliseconds, so a value of `900000` is a 15-minute update cycle.

- An activity class that will be used to configure the widget when it is first added to the screen (`android:configure`). This will be described in greater detail in a later section.

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

The Layout

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

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

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

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

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

The BroadcastReceiver

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

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

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

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

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

The Service

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

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

Here is the onHandleIntent() implementation from UpdateService:

```
@Override
public void onHandleIntent(Intent intent) {
  ComponentName me=new ComponentName(this,
                                     TwitterWidget.class);
  AppWidgetManager mgr=AppWidgetManager.getInstance(this);

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

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

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

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

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

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

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

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

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

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

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

  return(updateViews);
}
```

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

We then use methods like:

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

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

The Configuration Activity

Way back in the manifest, we included an <activity> element for a TWPrefs activity. And, in our widget metadata XML file, we said that TWPrefs was the android:configure attribute value. In our RemoteViews for the widget itself, we connect a configure button to launch TWPrefs when clicked.

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

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

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

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

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

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

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

That being said, here is the implementation of `TWPrefs`:

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

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

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

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

    addPreferencesFromResource(R.xml.preferences);
  }

  @Override
```

```
  public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode==KeyEvent.KEYCODE_BACK) {
      if (CONFIGURE_ACTION.equals(getIntent().getAction())) {
        Intent intent=getIntent();
        Bundle extras=intent.getExtras();

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

          mgr.updateAppWidget(id, views);

          Intent result=new Intent();

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

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

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

So, we have to cheat a bit.

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

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

This minimizes code duplication, but it does mean there is a slight hiccup, where the widget initially appears blank, before the first timeline entry appears. This is largely unavoidable in this case – we cannot wait for Twitter to respond since `onKeyDown()` is called on the UI thread and we need to call `setResult()` now rather than wait for Twitter's response.

Undoubtedly, there are other patterns for handling this situation.

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:

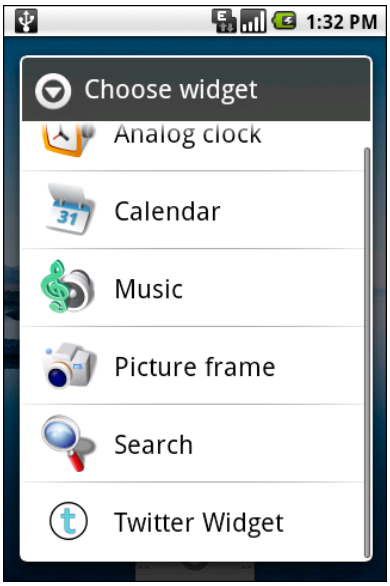

Figure 16. The roster of available widgets

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

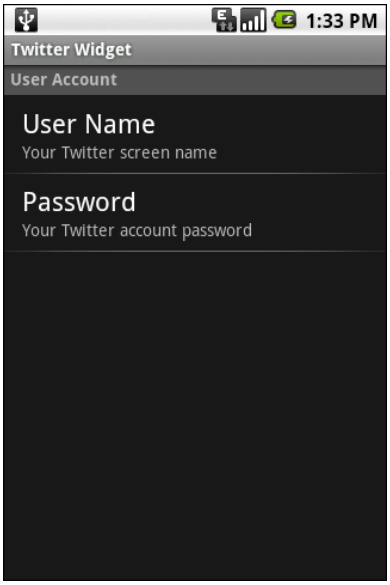

Figure 17. The TwitterWidget configuration activity

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

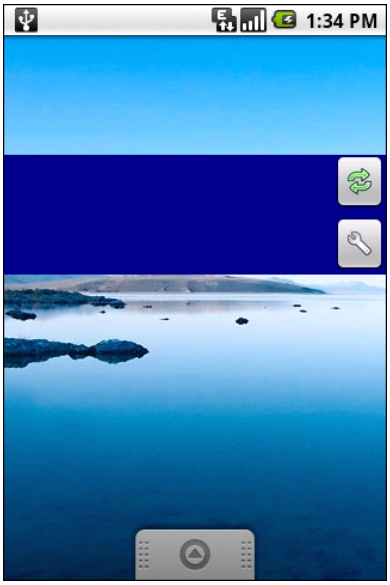

Figure 18. TwitterWidget, immediately after being added

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

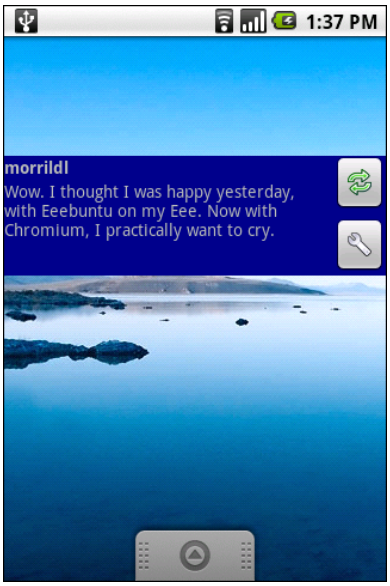

Figure 19. TwitterWidget, with a timeline entry

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

Another and Another

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

If you want to support separate app widget instances, you will need to store your state on a per-app-widget-identifier basis. For example, while `TwitterWidget` uses preferences for the Twitter account details, you might need multiple preference files, or use a SQLite database with an app widget identifier column, or something to distinguish one app widget instance from another. You will also need to use an appropriate version of

`updateAppWidget()` on `AppWidgetManager` when you update the app widgets, one that takes app widget identifiers as the first parameter, so you update the proper app widget instances.

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

App Widgets: Their Life and Times

TwitterWidget overrode two `AppWidgetProvider` methods:

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

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

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

Note, however, that there is a bug in Android 1.5r2, where `onDeleted()` will not be properly called. You will need to implement `onReceive()` and watch for the `ACTION_APPWIDGET_DELETED` action in the received `Intent` and call `onDeleted()` yourself. This should be fixed in a future edition of Android.

Controlling Your (App Widget's) Destiny

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

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

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

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

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