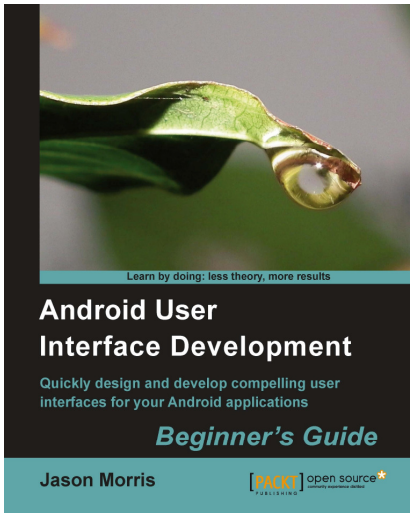
[PACKT] open source
community experience distilled
PUBLISHING

Android User Interface Development: Beginner's Guide

Jason Morris



Chapter No. 3

"Developing with Specialized Android Widgets"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Developing with Specialized Android Widgets"

A synopsis of the book's content

Information on where to buy this book

About the Author

Jason Morris has worked on software as diverse as fruit tracking systems, insurance systems, and travel search and booking engines. He has been writing software for as long as he can remember. He is currently working as a Software Architect for Travelstart in South Africa. He works on multiple front-end and middleware systems, leveraging a variety of Java based technologies.

The people I'd like to thank most for their direct, or indirect help in writing this book are my wife Caron Morris, my father Mike Morris, my mom Jayne Morris, and the rest of my family for their love and support. I'd also like to thank Wayne, Stuart, Angela, and James, and everyone on my team at Travelstart. Finally a very big thanks to Martin Skans for his invaluable input.

Android User Interface Development: Beginner's Guide

On 9th January, 2007, Apple officially launched the iPhone, and the world of user interface design shifted. While tablet PCs had been around for a while, the iPhone was the first device to give so many people a portable touchscreen, and people loved it. Just over a year later, Google and the Open Handset Alliance announced Android which in many ways is the direct competitor to iPhone.

What is it about touchscreen phones that we love? The answer is simple—feedback. Touchscreens offer a way to directly manipulate on-screen objects, which in the past had to be driven through a keyboard, mouse, joystick, or other input device. The touchscreen model of direct manipulation has a large impact on the way we think about our user interfaces as developers, and changes the expectations a user has for the application. Touchscreen devices require us to stop thinking in terms of forms, and start thinking about object-oriented user interfaces.

Android is used as the primary operating system for a rapidly expanding range of consumer electronics, including:

- Smartphones
- Netbooks
- Tablets
- Some desktop systems

While all of these devices have different purposes and specifications, all of them run Android. This is unlike many other operating environments which are almost always have a special purpose. The services and the APIs they provide to developers generally reflect their target hardware. Android on the other hand makes the assumption that a single application may be required to run on many different types of devices, with very different hardware capabilities and specifications, and makes it as easy as possible for developers to handle the differences between these devices simply and elegantly.

New challenges

As Android and the touchscreen devices it powers become increasingly common, they will bring a new set of challenges to user interface design and development:

- You generally don't have a mouse
- You may have more than one pointing device
- You often don't have a keyboard

- Any keyboard that does exist may be a software keyboard
- A software keyboard may consume some of your application's screenspace

The software keyboard reduces the amount of screen space available to your application, and in much the same vein, if there is a hardware keyboard present it may or may not always be exposed to the user. Therefore, not only are different Android devices different, but they may also appear to change features while your application is running.

The rule of finger

Most Android devices have touchscreens (although this is not a requirement). The first restriction placed on any touchscreen user interface is the size of the human forefinger, which of course varies widely from one person to another. If a widget is too small on the screen, it won't be clear what the user is trying to touch. You'll notice that most Android widgets take up plenty of space, and have more than the normal amount of padding around them. On a touchscreen device, you can't rely on pixel-perfect precision. You need to make sure that when the user touches a widget, they make contact, and they don't accidentally touch another widget.

The magic touch

Another impact touchscreens have on user interface design is that an application and all the widgets that it uses must be entirely self-explanatory (even more than usual). Far too often, we substitute good user interface planning and design with a roll-over or tooltip to indicate a widget's function. On a touchscreen device, there is no mouse or pointing device. The first interaction it has with the user is when they touch it, and they will expect something to happen.

A touchy subject

Most Android devices have a touchscreen, but it's not a requirement. The quality of a touchscreen also varies wildly from device to device. The category of touchscreens and their capabilities will also vary from one device to the next, depending on the intended use of the device and often its intended market segment.

A smaller view on the world

Most Android devices are small, and as a result have smaller screens and generally fewer pixels than a normal PC or laptop. This lack of size limits the size of the widgets. Widgets must be big enough to touch safely, but we also need to pack as much information onto the screen as possible. So don't give your users information that they don't want, and also avoid asking them for information you don't need.

Classic user interface principals

Here are some core guidelines which every user interface should follow. These guidelines are what will keep your users happy, and ensure your application is successful. Throughout the rest of the book, we'll be walking through these guidelines with practical examples of improvements that can be made to a user interface.

Consistency

This is the cornerstone of good user interface design. A button should look like a button. Make sure that the layout of each screen has a relationship with every other screen in your application. People often mistake this principle for "stick to the platform look and feel". Look and feel is important, consistency mostly applies to the layout and overall experience of the application, rather than the color scheme.

Recycling your interface

The easiest way to maintain a consistent user interface, is to recycle as much of it as possible. At first glance, this suggestion looks merely like a "good object-oriented" practice. However, a closer look will reveal ways to reuse graphical widgets in ways you hadn't thought of. By changing the visibility of various widgets, or you can reuse an edit screen to view list items of the intended type.

Simplicity

This is especially important in a phone-based application. Often, when a user encounters a new application, it's because they are looking for something. They may not have the time (or more often patience) to learn a new user interface. Make sure that your application asks for as little as possible, and guides the user to the exact information they want in as few steps as possible.

The Zen approach

Generally, when you are using a mobile device, your time is limited. You may also be using an application in less-than-ideal circumstances (perhaps, in a train). The lesser information a user needs to give an application, and the lesser they need to absorb from it, the better. Stripping away options and information also leads to a shorter learning-curve.

Android's hidden menu

A very useful feature of Android is the hidden menu structure. The menu is only visible when the user presses the "Menu" button, which would generally mean, they're looking for something that isn't currently on the screen. Typically, a user shouldn't need to open a menu. However, it's a good way of hiding advanced features until they are needed.

Feedback

Feedback is what makes a touchscreen device exciting. When you drag an object, it sticks to your finger across the screen until you let go of it. When the users puts their finger on your application, they expect some reaction. However, you don't want to get in their way—instead of showing an error message when they touch a button, disable the button until it's valid to use, or don't show it at all.

Location and navigation

When you're in a place you've never been to previously, it's easy to get disoriented, or lost. The same is true for a piece of software. Just because the application makes sense to you, the developer, it doesn't mean it seems logical to your user. Adding transition animations, breadcrumbs, and progress gauges help the user to identify where in the application they are, and what's happening.

The road to recovery

A common way to tell users that something is wrong on a desktop application, or on the web is to open an error dialog. On a mobile device, people want smoother use of an application. While in a normal application you may inform the user that they selected an invalid option, in a mobile application, you generally want to make sure they can't select that option in the first place. Also, don't make them scroll through huge lists of options. Instead, allow them to filter through the list using an auto-complete or something similar.

When something goes wrong, be nice, and be helpful—don't tell the user, "I couldn't find any flights for your search". Instead tell them, "There were no available flights for your search, but if you're prepared to leave a day earlier, here is a list of the available flights". Always make sure your user can take another step forward without having to go "Back" (although the option to go backwards should always exist).

The Android way

The Android platform is in many ways similar to developing applications for the web. There are many devices, made by many manufactures, with different capabilities and specifications. Yet as a developer, you will want your users to have the most consistent experience possible. Unlike a web browser, Android has built-in mechanisms for coping with these differences, and even leveraging them.

We'll be looking at Android from the point of view of a user rather than having a purely development-centric approach. We'll cover topics such as:

- What user interface elements Android provides
- How an Android application is assembled
- Different types of Android layouts
- Presenting various types of data to the user
- Customising of existing Android widgets
- Tricks and tools to keep user interfaces looking great
- Integration between applications

We're about to take a jump into building user interfaces for Android devices—all Android devices, from the highest speed CPU to the smallest screen.

What This Book Covers

Chapter 1, Developing a Simple Activity introduces the basics of building an Android application, starting with a simple user interface. It also covers the various options available to you when implementing your design as code.

Chapter 2, Views With Adapters shows us how to leverage Adapter-based widgets, Android's answer to the Model-View-Controller (MVC) structure. Learn about these widgets, and where they will best serve you.

Chapter 3, Specialized Android Views takes a close look at some of the more specialized widgets that the Android platform provides, and how they relate to the mundane widgets. This chapter covers widgets such as the gallery and rating-bar, and how they can be used and styled.

Chapter 4, Activities and Intents discusses more about how Android runs your application, and from that point-of-view, how best to write its user interfaces. This chapter takes a look at how to make sure that your application will behave the way users expect it to, with minimal effort on your part.

Chapter 5, Non-Linear Layouts takes a look at some of the advanced layout techniques which Android offers. It talks about the best way to present different screens to the user while taking into account the wide discrepancy in the screens on Android devices.

Chapter 6, Input and Validation provides tips regarding taking input from a user, and how to keep this experience as painless as possible. This chapter investigates the different input widgets Android provides and how to configure them best, depending on the situation. Also, when everything else fails, how best to inform your users that what they are doing is wrong.

Chapter 7, Animating Widgets and Layouts will inform the reader as to where, when, why, and how to animate your Android user interfaces. It also sheds light on what kind of animations are provided by default, how to compose them together, and how to build your own. This chapter looks at the importance of animations in a mobile user interface and demonstrates how complex animations are made easy by Android.

Chapter 8, Content-centric Design details how to go about designing the screen layout, when presenting the user with information on the screen. This chapter looks at the pros and cons of some of the different display techniques which Android offers.

Chapter 9, Styling Android Applications shows us how to keep the look of our entire application consistent, in order to make our application easier to use.

Chapter 10, Building an Application Theme looks at the design process, and how application-wide themes can be applied to help your application stand out.

3
Developing with Specialized Android Widgets

Along with the many generic widgets such as buttons, text fields, and checkboxes, Android also includes a variety of more specialized widgets. While a button is fairly generic, and has use in many situations, a gallery-widget for example, is far more targeted. In this chapter we will start looking at the more specialized Android widgets, where they appear, and how best they can be used.

Although these are very specialized `View` classes, they are very important. As mentioned earlier (and it really can't be stressed enough) one of the cornerstones of good user interface design is **consistency**. An example is the `DatePicker` widget. It's certainly not the prettiest date-selector in the world. It's not a calendar widget, so it's sometimes quite difficult for the user to select exactly which date they want (most people think in terms of "next week Tuesday", and not "Tuesday the 17th"). However, the `DatePicker` is standard! So the user knows exactly how to use it, they don't have to work with a broken calendar implementation. This chapter will work with Android's more specialized `View` and layout classes:

- `Tab` layouts
- `TextSwitcher`
- `Gallery`
- `DatePicker`
- `TimePicker`
- `RatingBar`

These classes have very specialized purposes, and some have slight quirks in the way they are implemented. This chapter will explore how and where to use these widgets, and where you need to be careful of their implementation details. We'll also discuss how best to incorporate these elements into an application, and into a layout.

Creating a restaurant review application

In the previous chapter, we built an ordering-in application. In this chapter, we're going to take a look at reviewing restaurants. The application will allow the user to view other people's opinions on the restaurant, a gallery of photos of the restaurant, and finally a section for making an online reservation. We will divide the application into three sections:

- **Review**: Review and ratings information for this restaurant
- **Photos**: A photo gallery of the restaurant
- **Reservation**: Request a reservation with the restaurant

When building an application where all three of these sections need to be quickly available to the user, the most sensible option available is to place each of the sections in a tab on the screen. This allows the user to switch between the three sections without having all of them on the screen at the same time. This also saves screen real estate giving us more space for each section.

The **Review** tab will include a cycling list of comments that people have made about the restaurant being viewed, and an average "star" rating for the restaurant.

Displaying photographs of the restaurant is the job of the **Photos** tab. We'll provide the user with a thumbnail "track" at the top of the screen, and a view of the selected image consuming the remaining screen space.

For the **Reservation** tab, we will want to capture the user's name and when they would like the reservation to be (date and time). Finally we also need to know for how many people the reservation will be made.

Time for action – creating the robotic review project structure

To start this example we'll need a new project with a new `Activity`. The new layout and `Activity` will be a little different from the structures in the previous two chapters. We will need to use the `FrameLayout` class in order to build a tabbed layout. So to begin, we'll create a new project structure and start off with a skeleton that will later become our tab layout structure. This can be filled with the three content areas.

1. Create a new Android project using the Android command-line tool:

```
android create project -n RoboticReview -p RoboticReview -k com.
packtpub.roboticreview -a ReviewActivity -t 3
```

2. Open the `res/layout/main.xml` file in an editor or IDE.

3. Clear out the default code (leaving in the XML header).

4. Create a root `FrameLayout` element:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
```

5. Inside the new `FrameLayout` element, add a `vertical` `LinearLayout`:

```
<LinearLayout android:id="@+id/review"
              android:orientation="vertical"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content">
</LinearLayout>
```

6. After the `LinearLayout`, add another empty `LinearLayout` element:

```
<LinearLayout android:id="@+id/photos"
              android:orientation="vertical"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content">
</LinearLayout>
```

7. Then, after the second `LinearLayout` element, add an empty `ScrollView`:

```
<ScrollView android:id="@+id/reservation"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent">
</ScrollView>
```

The `FrameLayout` will be used by the Android tab structures as a content area, each of the child elements will become the contents of a tab. In the preceding layout, we've added in two `LinearLayout` elements for the **Review** and **Photos** sections, and a `ScrollView` for the **Reservation** tab.

What just happened?

We've just started the "restaurant review" application, building a skeleton for the user interface. There are several key parts of this `main.xml` file which we should walk through before continuing the example.

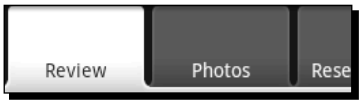
First, our root element is a `FrameLayout`. The `FrameLayout` anchors all of its children to its own top-left corner. In effect, the two occurrences of `LinearLayout` and the `ScrollView` will overlap each other. This structure can be used to form something like a Java AWT `CardLayout`, which will be used by the `TabHost` object to display these objects when their relative tab is active.

Second, each of the `LinearLayout` and the `ScrollView` have an ID. In order to identify them as tab roots, we need to be able to easily access them from our Java code. Tab structures may be designed in XML, but they need to be put together in Java.

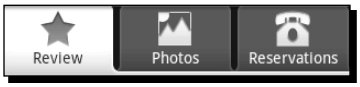
Building a TabActivity

In order to continue, we need our `Activity` class to set up the three tab content elements we declared in our `main.xml` file as tabs. By preference, all tabs in Android should have an icon.

The following is a screenshot of the tabs without their icons:



The following is a screenshot of the tabs with the icons:



Creating tab icons

Android applications have a specific look and feel defined by the default widgets provided by the system. In order to keep all applications consistent for users, there are a set of user interface guidelines that application developers should follow. While it's important to have your application stand out, users will often get frustrated with applications that are not familiar or look out of place (this is one of the reasons why automatically ported applications are often very unpopular).

Android tabs and icons

When selecting tab icons for your application, it's considered a good practice to include several different versions of the icon for different screen sizes and densities. The anti-aliased corners that look so good on a high-density screen, look terrible on low-density screens. You can also provide entirely different icons for very small screens, instead of loosing all of your icons details. Android tabs appear raised when they are selected, and lowered in the background when they are not selected. The Android tab icons should appear in the "opposite" etching effect to the tab that they are placed in, that is, lowered when they are selected and raised when they are not selected. The icons therefore have two primary states: selected and unselected. In order to switch between these two states, a tab-icon will generally consist of three resource files:

- The selected icon image
- The unselected icon image
- An XML file describing the icon in terms of its two states

Tab icons are generally simple shapes while the image size is squared (generally at a maximum of 32 x 32 pixels). Different variations of the image should be used for screens of different pixel densities (see *Chapter 1*, *Developing a Simple Activity* for "Resource Selection" details). Generally you will use a dark outset image for the selected state, since when a tab is selected, the tab background is light. For the unselected icon, the opposite is true and a light inset image should be used instead.

The bitmap images in an Android application should always be in the PNG format. Let's call the selected icon for the **Review** tab `res/drawable/ic_tab_selstar.png`, and name the unselected icon file `res/drawable/ic_tab_unselstar.png`. In order to switch states between these two images automatically, we define a special `StateListDrawable` as an XML file. Hence the **Review** icon is actually in a file named `res/drawable/review.xml`, and it looks like this:

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

    <item
        android:drawable="@drawable/ic_tab_selstar"
        android:state_selected="false"/>
    <item
        android:drawable="@drawable/ic_tab_unselstar"
        android:state_selected="true"/>
</selector>
```

Note the `android:constantSize="true"` of the `<selector>` element. By default, Android will assume that each state in the resulting `StateListDrawable` object will cause the image to be of a different size, which in turn may cause the user interface to re-run its layout calculations. This can be fairly expensive, so it's a good idea to declare that each of your states is exactly of the same size.

For this example, we'll be using three tab icons, each with two states. The icons are named `review`, `photos`, and `book`. Each one is composed of three files: A PNG for the selected icon, a PNG for the unselected icon, and an XML file defining the state-selector. From our application, we will only make direct use of the state-selector XML files, leaving the Android APIs to pickup the actual PNG files.

Implementing the ReviewActivity

As usual, we will want to have localized text in our `strings.xml` file. Open the `res/values/strings.xml` file and copy the following code into it:

```
<resources>
    <string name="app_name">Robotic Review</string>
    <string name="review">Review</string>
    <string name="gallery">Photos</string>
    <string name="reservation">Reservations</string>
</resources>
```

Time for action – writing the ReviewActivity class

As already said, we will need to set up our tabbed-layout structure in our Java code. Fortunately, Android provides a very useful `TabActivity` class that does much of the heavy lifting for us, providing us with a ready-made `TabHost` object with which we can construct the `Activity` tab structure.

1. Open the `ReviewActivity.java` file generated earlier in an editor or IDE.
2. Instead of extending `Activity`, change the class to inherit `TabActivity`:
   ```
   public class ReviewActivity extends TabActivity
   ```
3. In the `onCreate` method, remove the `setContentView(R.layout.main)` line (generated by the `android create project` utility) completely.
4. Now start by fetching the `TabHost` object from your parent class:
   ```
   TabHost tabs = getTabHost();
   ```

5. Next, we inflate our layout XML into the content view of the `TabHost`:

```
getLayoutInflater().inflate(
        R.layout.main,
        tabs.getTabContentView(),
        true);
```

6. We'll need access to our other application resources:

```
Resources resources = getResources();
```

7. Now we define a `TabSpec` for the **Review** tab:

```
TabHost.TabSpec details =
        tabs.newTabSpec("review").
        setContent(R.id.review).
        setIndicator(getString(R.string.review),
        resources.getDrawable(R.drawable.review));
```

8. Define two more `TabSpec` variables for the **Photos** and **Reservation** tabs using the preceding pattern.

9. Add each of the `TabSpec` objects to our `TabHost`:

```
tabs.addTab(details);
tabs.addTab(gallery);
tabs.addTab(reservation);
```

This concludes the creation of the tab structure for the `ReviewActivity` class.

What just happened?

We built a very basic tabbed-layout for our new `ReviewActivity`. When working with tabs, we didn't simply use the `Activity.setContentView` method, instead we inflated the layout XML file ourselves. Then we made use of the `TabHost` object provided by the `TabActivity` class to create three `TabSpec` objects. A `TabSpec` is a builder object that enables you to build up the content of your tab, similar to the way you build up text with a `StringBuilder`.

The content of a `TabSpec` is the content-view that will be attached to the tab on the screen (assigned using the `setContent` method). In this example, we opted for the simplest option and defined the tab content in our `main.xml` file. It's also possible to lazy-create the tab content using the `TabHost.TabContentFactory` interface, or even to put an external `Activity` (such as the dialer or browser) in the tab by using `setContent(Intent)`. However, for the purposes of this example we used the simplest option.

You'll notice that the `TabSpec` (much like the `StringBuilder` class) supports chaining of method calls, making it easy and flexible to either set up a tab in a "single shot" approach (as done previously), or build up the `TabSpec` in stages (that is, while loading from an external service).

The `indicator` we assigned to the `TabSpec` is what will appear on the tab. In the previous case, a string of text and our icon. As of API level 4 (Android version 1.6) it's possible to use a `View` object as an `indicator`, allowing complete customization of the tab's look and feel. To keep the example simple (and compatible with earlier versions) we've supplied a `String` resource as the `indicator`.

Time for action – creating the Review layout

We've got a skeleton tab structure, but there's nothing in it yet. The first tab is titled **Review**, and this is where we are going to start. We've just finished enough Java code to load up the tabs and put them on the screen. Now we go back to the `main.xml` layout file and populate this tab with some widgets that supply the user with review information.

1. Open `res/layout/main.xml` in an editor or IDE.

2. Inside the `<LayoutElement>` that we named `review`, add a new `TextView` that will contain the name of the restaurant:

```
<TextView android:id="@+id/name"
          android:textStyle="bold"
          android:textSize="25sp"
          android:textColor="#ffffffff"
          android:gravity="center|center_vertical"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"/>
```

3. Below the new `TextView`, add a new `RatingBar`, where we will display how other people have rated the restaurant:

```
<RatingBar android:id="@+id/stars"
           android:numStars="5"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"/>
```

4. Keeping this first tab simple, we add a `TextSwitcher` where we can display other people's comments about the restaurant:

```
<TextSwitcher android:id="@+id/reviews"
              android:inAnimation="@android:anim/fade_in"
              android:outAnimation="@android:anim/fade_out"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"/>
```

The **Review** tab only has three widgets in this example, but more could easily be added to allow the user to input their own reviews.

What just happened

We just put together the layout for our first tab. The `RatingBar` that we created has a width of `wrap_content`, which is really important. If you use `fill_parent`, then the number of stars visible in the `RatingBar` will simply be as many as can fit on the screen. If you want control over how many stars appear on your `RatingBar`, stick to `wrap_content`, but also make sure that (at least on portrait layouts) the `RatingBar` has its own horizontal line. If you install the `Activity` in the emulator now, you won't see anything in either the `TextView` or the `TextSwitcher`.

The `TextSwitcher` has no default animations, so we specify the "in" animation as the default `fade_in` as provided by the `android` package, while the "out" animation will be `fade_out`. This syntax is used to access resources that can be found in the `android.R` class.

Working with switcher classes

The `TextSwitcher` we've put in place is used to animate between different `TextView` objects. It's really useful for displaying things like changing stock-prices, news headlines, or in our case, reviews. It inherits from `ViewSwitcher` which can be used to animate between any two generic `View` objects. `ViewSwitcher` extends `ViewAnimator` which can be used as a sort of animated `CardLayout`.

We want to display a series of comments from past customers, fading between each of them with a short animation. `TextSwitcher` needs two `TextView` objects (which it will ask us to create dynamically), for our example. We want these to be in a resource file.

For the next part of the example, we'll need some comments. Instead of using a web service or something similar to fetch real comments, this example will load some comments from its application resources. Open the `res/values/strings.xml` file and add `<string-array name="comments">` with a few likely comments in it:

```
<string-array name="comments">
    <item>Just Fantastic</item>
    <item>Amazing Food</item>
    <item>What rubbish, the food was too hairy</item>
    <item>Messy kitchen; call the health inspector.</item>
</string-array>
```

Time for action – turning on the TextSwitcher

We want the `TextSwitcher` to display the next listed comment every five seconds. For this we'll need to employ new resources, and a `Handler` object. A `Handler` is a way for Android applications and services to post messages between threads, and can also be used to schedule messages at a point in the future. It's a preferred structure to use over a `java.util.Timer` since a `Handler` object will not allocate a new `Thread`. In our case, a `Timer` is overkill, as there is only one task we want to schedule.

1. Create a new XML file in your `res/layout` directory named `review_comment.xml`.
2. Copy the following code into the new `review_comment.xml` file:

```
<TextView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:gravity="left|top"
    android:textStyle="italic"
    android:textSize="16sp"
    android:padding="5dip"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
```

3. Open the `ReviewActivity.java` file in your editor or IDE.
4. We'll need to be able to load the `review_comment` resources for the `TextSwitcher`, so `ReviewActivity` needs to implement the `ViewSwitcher.ViewFactory` interface.
5. In order to be update the `TextSwitcher`, we need to interact with a `Handler`, and the easiest way to do that here is to also implement `Runnable`.
6. At the top of the `ReviewActivity` class, declare a `Handler` object:

```
private final Handler switchCommentHandler = new Handler();
```

7. We'll also want to hold a reference to the `TextSwitcher` for our `run()` method when we switch comments:

```
private TextSwitcher switcher;
```

8. In order to display the comments, we'll need an array of them, and an index to keep track of which comment the `TextSwitcher` is displaying:

```
private String[] comments;
private int commentIndex = 0;
```

9. Now, in the `onCreate` method, after you add the `TabSpec` objects to the `TabHost`, read the `comments` string-array from the `Resources`:

```
comments = resources.getStringArray(R.array.comments);
```

10. Next, find the `TextSwitcher` and assign it to the `switcher` field:

```
switcher = (TextSwitcher)findViewById(R.id.reviews);
```

11. Tell the `TextSwitcher` that the `ReviewActivity` object will be its `ViewFactory`:

```
switcher.setFactory(this);
```

12. In order to comply with the `ViewFactory` specification, we need to write a `makeView` method. In our case it's really simple—inflate the `review_comment` resource:

```
public View makeView() {
    return getLayoutInflater().inflate(
            R.layout.review_comment, null);
}
```

13. Override the `onStart` method so that we can post the first timed event on the `Handler` object declared earlier:

```
protected void onStart() {
    super.onStart();
    switchCommentHandler.postDelayed(this, 5 * 1000l);
}
```

14. Similarly, override the `onStop` method to cancel any future callback:

```
protected void onStop() {
    super.onStop();
    switchCommentHandler.removeCallbacks(this);
}
```

15. Finally, the `run()` method alternates the comments in the `TextSwitcher`, and in the `finally` block, posts itself back onto the `Handler` queue in five seconds:

```
public void run() {
    try {
        switcher.setText(comments[commentIndex++]);
        if(commentIndex >= comments.length) {
            commentIndex = 0;
        }
    } finally {
        switchCommentHandler.postDelayed(this, 5 * 1000l);
    }
}
```

Using `Handler` objects instead of creating `Thread` objects means all of the timed tasks can share the main user interface thread instead of each allocating a separate thread. This reduces the amount of memory and CPU load your application places on the device, and has a direct impact on the application performance and battery life.

What just happened?

We just built a simple timer structure to update the `TextSwitcher` with a rotating array of comments. The `Handler` class is a convenient way to post messages and actions between two application threads. In Android, as with Swing, the user interface is not thread-safe, so inter-thread communication becomes very important. A `Handler` object attempts to bind itself to the thread it's created in (in the preceding case, the `main` thread).

It's a prerequisite that a thread which creates a `Handler` object must have an associated `Looper` object. You can set this up in your own thread by either inheriting the `HandlerThread` class, or using the `Looper.prepare()` method. Messages sent to a `Handler` object will be executed by the `Looper` associated with the same thread. By sending our `ReviewActivity` (which implements `Runnable`) to the `Handler` object that we had created in the `main` thread, we know that the `ReviewActivity.run()` method will be executed on the `main` thread, regardless of which thread posted it there.

In the case of long-running tasks (such as fetching a web page or a long-running calculation), Android provides a class that bares a striking resemblance to the `SwingWorker` class, named `AsyncTask`. `AsyncTask` (like `Handler`) can be found in the `android.os` package, and you make use of it by inheritance. `AsyncTask` is used to allow interaction between a background task and the user interface (in order to update a progress bar or similar requirements).

Creating a simple photo gallery

The use of the word `Gallery` is a little misleading, it's really a horizontal row of items with a "single item" selection model. For this example we'll be using the `Gallery` class for what it does best, displaying thumbnails. However, as you'll see, it's capable of displaying scrolling lists of almost anything. Since a `Gallery` is a spinner, you work with it in much the same way as a `Spinner` object or a `ListView`, that is, with an `Adapter`.

Time for action – building the Photos tab

Before we can add images to a `Gallery`, we need the `Gallery` object on the screen. To start this exercise, we'll add a `Gallery` object and an `ImageView` to `FrameLayout` of our tabs. This will appear under the **Photos** tab that we created at the beginning of the chapter. We'll stick to a fairly traditional photo gallery model of the sliding thumbnails at the top of the screen, with the full view of the selected image below it.

1. Open `res/layout/main.xml` in your editor or IDE.

2. Inside the second `LinearLayout`, with `android:id="@+id/photos"`, add a new `Gallery` element to hold the thumbnails:

```
<Gallery android:id="@+id/gallery"
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"/>
```

3. `Gallery` objects, by default, squash their contents together, which really doesn't look great in our case. You can add a little padding between the items by using the `spacing` attribute of `Gallery` class:

```
android:spacing="5dip"
```

4. We also have tabs directly above the `Gallery`, and we'll have an `ImageView` directly below it. Again, there won't be any padding, so add some using a margin:

```
android:layout_marginTop="5dip"
android:layout_marginBottom="5dip"
```

5. Now create an `ImageView` which we can use to display the full-sized image:

```
<ImageView android:id="@+id/photo"
           android:layout_width="fill_parent"
           android:layout_height="fill_parent"/>
```

6. In order to ensure that the full display is scaled correctly, we need to specify the `scaleType` on the `ImageView`:

```
android:scaleType="centerInside"
```

The `Gallery` element provides us with the thumbnail track at the top of the screen. The image selected in the `Gallery` will be displayed at full-size in the `ImageView` widget.

What just happened?

We just populated the second tab with the standard widgets required for a basic photo gallery. This structure is very generic, but is also well known and understood by users. The `Gallery` class will handle the thumbnails, scrolling, and selection. However, you will need to populate the main `ImageView` with the selected image, and provide the `Gallery` object with the thumbnail widgets to display on the screen.

The spacing attribute on the `Gallery` element will add some whitespace, which serves as a simple separator between thumbnails. You could also add a border into each of the thumbnail images, border each `ImageView` widget you return for a thumbnail, or use a custom widget to create a border.

Creating a thumbnail widget

In order to display the thumbnails in the `Gallery` object, we will need to create an `ImageView` object for each thumbnail. We could easily do this in Java code, but as usual, it is preferable to build even the most basic widgets using an XML resource. In this case, create a new XML resource in the `res/layout` directory. Name the new file `gallery_thn.xml` and copy the following code into it:

```
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
           android:scaleType="fitXY"/>
```

That's right, it has just two lines of XML, but to reiterate, this allows us to customize this widget for many different configurations without editing the Java code. While editing the code might not seem like a problem (the resource needs to be recompiled anyway), you also don't want to end up with a long series of `if` statements to decide on exactly how you should create the `ImageView` objects.

Implementing a GalleryAdapter

For the example, we'll stick to using application resources to keep things simple. We'll have two arrays of resource IDs, thumbnails, and the full-size images. An `Adapter` implementation is expected to provide an identifier for each of the items. In this next example, we're going to provide an identifier as the resource identifier of the full-size image, which gives us easy access to the full-size image in classes outside of the `Adapter` implementation. While this is an unusual contract, it provides a convenient way for us to pass the image resource around within an already defined structure.

In order to display your gallery, you'll need some images to display (mine are sized 480 x 319 pixels). For each of these images, you'll need a thumbnail image to display in the `Gallery` object. Generally, these should simply be a scaled-down version of the actual image (mine are scaled to 128 x 84 pixels).

Time for action – the GalleryAdapter

Creating the `GalleryAdapter` is much like the `ListAdapter` classes we created in *Chapter 2*, *Presenting Data for Views*. The `GalleryAdapter` however, will use `ImageView` objects instead of `TextView` objects. It also binds two lists of resources together instead of using an object model.

1. Create a new Java class in your project root package named `GalleryAdapter`. It should extend the `BaseAdapter` class.

2. Declare an integer array to hold the thumbnail resource IDs:

```
private final int[] thumbnails = new int[]{
    R.drawable.curry_view_thn,
    R.drawable.jai_thn,
    // your other thumbnails
};
```

3. Declare an integer array to hold the full-size image resource IDs:

```
private final int[] images = new int[]{
    R.drawable.curry_view,
    R.drawable.jai,
    // your other full-size images
};
```

4. The `getCount()` method is simply the length of the `thumbnails` array:

```
public int getCount() {
    return thumbnails.length;
}
```

5. The `getItem(int)` method returns the full-size image resource ID:

```
public Object getItem(int index) {
    return Integer.valueOf(images[index]);
}
```

6. As mentioned earlier, the `getItemId(int)` method returns the full-size image resource ID (almost exactly the way that `getItem(int)` does):

```
public long getItemId(int index) {
    return images[index];
}
```

7. Finally, the `getView(int, View, ViewGroup)` method uses a `LayoutInflater` to read and populate the `ImageView` which we created in the `gallery_thn.xml` layout resource:

```
public View getView(int index, View reuse, ViewGroup parent) {
    ImageView view = (reuse instanceof ImageView)
            ? (ImageView)reuse
            : (ImageView)LayoutInflater.
                        from(parent.getContext()).
                        inflate(R.layout.gallery_thn, null);
    view.setImageResource(thumbnails[index]);
    return view;
}
```

The `Gallery` class is a subclass of `AdapterView` and so functions in the same way as a `ListView` object. The `GalleryAdapter` will provide the `Gallery` object with `View` objects to display the thumbnails in.

What just happened

Much like the `Adapter` classes built in the last chapter, the `GalleryAdapter` will attempt to reuse any `View` object specified in its `getView` method. A primary difference however, is that this `GalleryAdapter` is entirely self-contained, and will always display the same list of images.

This example of a `GalleryAdapter` is extremely simple. You could also build a `GalleryAdapter` that held bitmap objects instead of resource ID references. You'd then make use of the `ImageView.setImageBitmap` method instead of `ImageView.setImageResource`.

You could also eliminate the thumbnail images by having the `ImageView` scale the full-size images into thumbnails. This would just require a modification to the `gallery_thn.xml` resource file in order to specify the required size of each thumbnail.

```
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
           android:maxWidth="128dip"
           android:adjustViewBounds="true"
           android:scaleType="centerInside"/>
```

The `adjustViewBounds` attribute tells the `ImageView` to adjust its own size in a way such that it maintains the aspect ratio of the image it contains. We also change the `scaleType` attribute to `centerInside`, which will also retain the aspect ratio of the image when it scales. Finally, we set a maximum width for the `ImageView`. Using the standard `layout_width` or `layout_height` attributes is ignored by the `Gallery` class, so we instead specify the desired thumbnail size to the `ImageView` (the `layout_width` and `layout_height` attributes are handled by the `Gallery`, while the `maxWidth` and `maxHeight` are handled by the `ImageView`).

This would be a standard speed/size trade-off. Having the thumbnail images takes up more application space, but having the `ImageView` perform the scaling makes the application slower. The scaling algorithm in `ImageView` will also not be as high-quality as the scaling performed in an image-manipulation application such as Adobe Photoshop. In most cases this won't be a problem, but if you have high detail images, you often get "scaling artifacts" with simpler scaling algorithms.

Time for action – making the gallery work

Now that we've got the `GalleryAdapter` working, we need to connect the `Gallery`, the `GalleryAdapter`, and the `ImageView` together, so that when a thumbnail is selected, the full-view of that image is displayed in the `ImageView` object.

1. Open the `ReviewActivity` source code in your editor or IDE.
2. Add `AdapterView.OnItemSelectedListener` to the interfaces that the `ReviewActivity` implements.
3. Below the declaration of the `TextSwitcher`, declare a reference to the `ImageView` which will hold the full-size image:

```
private TextSwitcher switcher;
private ImageView photo;
```

4. At the end of the `onCreate` method, find the `ImageView` named `photo` and assign it to the reference you just declared:

```
photo = ((ImageView)findViewById(R.id.photo));
```

5. Now fetch the `Gallery` object as declared in the `main.xml` layout resource:

```
Gallery photos = ((Gallery)findViewById(R.id.gallery));
```

6. Create a new `GalleryAdapter` and set it on the `Gallery` object:

```
photos.setAdapter(new GalleryAdapter());
```

7. Set the `OnItemSelectedListener` of the `Gallery` object to `this`:

```
photos.setOnItemSelectedListener(this);
```

8. At the end of the `ReviewActivity` class, add the `onItemSelected` method:

```
public void onItemSelected(
        AdapterView<?> av, View view, int idx, long id) {

    photo.setImageResource((int)id);
}
```

9. `OnItemSelectedListener` requires an `onNothingSelected` method as well, but we don't need it to do anything for this example.

The `GalleryAdapter` provides the `ReviewActivity` with the resource to load for the full view of the photo through the `id` parameter. The `id` parameter could also be used as an index or identifier for a URL if the image was located on a remote server.

What just happened?

We've now connected the `Gallery` object to the `ImageView` where we will display the full-size image instead of the thumbnail. We've used the item ID as a way to send the resource ID of the full-size image directly to the event listener. This is a fairly strange concept since you'd normally use an object model. However, an object model in this example wouldn't just introduce a new class, it would also require another method call (in order to fetch the image object from the `Adapter` when the event is triggered).

When you specify an `Adapter` on an `AbsSpinner` class like `Gallery`, it will by default attempt to select the first item returned from its new `Adapter`. This in turn notifies the `OnItemSelectedListener` object if one has been registered. However, because of the single-threading model used by the Android user interface objects, this event doesn't get fired immediately, but rather some time after we return from the `onCreate` method. When we call `setAdapter(new GalleryAdapter())` on the `Gallery` object, it schedules a selection change event, which we then receive. The event causes the `ReviewActivity` class to display the first photo in the `GalleryAdapter` object.

If you now reinstall the application in your emulator, you'll be able to go to the **Photos** tab and browse through a `Gallery` of all the images that you had populated the `GalleryAdapter` with.

Pop quiz

1. What would happen in the previous example if you substituted `OnItemSelectedListener` with `OnItemClickListener` (as done in the `ListView` examples)?
   a. The full size won't appear anymore.
   b. The `Gallery` will not rotate the thumbnails when they are touched.
   c. The full-size photo won't appear until a thumbnail is clicked.
2. What is the primary difference between the `ScaleType` values `fitXY` and `centerInside`?
   a. The `fitXY` type will anchor the picture to the top-left, while `centerInside` will center the picture in the `ImageView`.
   b. `fitXY` will cause the picture to distort to the size of the `ImageView`, while `centerInside` will maintain the picture's aspect ratio.
   c. `centerInside` will cause the larger axis to be cropped in order to fit the picture into the `ImageView`, while `fitXY` will scale the picture so that the larger axis is of the same size as the `ImageView`.
3. What dictates the size of a `Gallery` object containing `ImageView` objects when using the `wrap_content` attribute?
   a. The width and height of the `ImageView` objects, as dictated by the size of their content image, or their `maxWidth` and `maxHeight` parameters.
   b. The `itemWidth` and `itemHeight` parameters on the `Gallery` object.
   c. The `LayoutParams` set on the `ImageView` objects (either with the `setLayoutParams` method, or `layout_width`/`layout_height` attributes).

Have a go hero – animations and external sources

Now that you have the basic example working, try improving the user experience a bit. When you touch the images, they should really animate instead of undergoing an instant change. They should also come from an external source instead of application resources.

1. Change the `ImageView` object of full-size images to an `ImageSwitcher`, use the standard Android fade-in/fade-out animations.
2. Remove the thumbnail images from the project, and use the `ImageView` declared in the `gallery_thn.xml` file to scale the images.
3. Change from a list of application resource IDs to a list of `Uri` objects so that the images are downloaded from an external website.

Building the reservation tab

While the **Review** and **Photos** tabs of this example have been concerned with displaying information, the **Reservation** tab will be concerned with capturing the details of a reservation. We really only need three pieces of information:

- The name under which the reservation needs to be made
- The date and time of the reservation
- How many people the reservation is for

In this part of the example we'll create several widgets which have formatted labels. For example, **How Many People: 2**, which will update the number of people as the user changes the value. In order to do this simply, we specify that the widget's text (as specified in the layout file) will contain the format to use for display. As part of the initialization procedure, we read the text from the `View` object and use it to create a format structure. Once we have a format, we populate the `View` with its initial value.

Time for action – implementing the reservation layout

In our `main.xml` layout resource, we need to add the `View` objects which will form the **Reservation** tab. Currently it consists only of an empty `ScrollView`, which enables vertically-long layouts to be scrolled by the user if the entire user interface doesn't fit on the screen.

1. Open the `main.xml` file in your editor or IDE.
2. Inside the `<ScrollView>` we had created for the `Reservation` tab earlier. Declare a new vertical `LinearLayout` element:

```
<LinearLayout android:orientation="vertical"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content">
```

3. Inside the new `LinearLayout` element, create a `TextView` to ask the user under what name the reservation should be made:

```
<TextView android:text="Under What Name:"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"/>
```

4. After the `TextView` label, create an `EditText` to allow the user to input the name under which reservation is to be made:

```
<EditText android:id="@+id/name"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"/>
```

5. Create another `TextView` label to ask the user how many people will be going. This includes a format element where we will place the number:

```
<TextView android:id="@+id/people_label"
          android:text="How Many People: %d"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"/>
```

6. Add a `SeekBar` with which the user can tell us about how many people are going:

```
<SeekBar android:id="@+id/people"
         android:max="20"
         android:progress="1"
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"/>
```

7. Use another `TextView` to ask the user what date the reservation will be on:

```
<TextView android:text="For What Date:"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"/>
```

8. Add a `Button` to display the date for which the reservation is made. When the user taps this `Button`, we will ask him to select a new date:

```
<Button android:id="@+id/date"
        android:text="dd - MMMM - yyyy"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
```

9. Create another `TextView` label to ask the time of reservation:

```
<TextView android:text="For What Time:"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"/>
```

10. Add another `Button` to display the time, and allow the user to change it:

```
<Button android:id="@+id/time"
        android:text="HH:mm"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
```

11. Finally add a `Button` to make the reservation, and add some margin to separate it from the rest of the inputs in the form:

```
<Button android:id="@+id/reserve"
        android:text="Make Reservation"
        android:layout_marginTop="15dip"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
```

Several of the preceding widgets include the format of their labels instead of the label literal, the actual label will be generated and set in the Java code. This is because these labels are subject to change when the user changes date, time, or the number of people expected for the reservation.

What just happened?

In the **Reservation** tab, we ask the user how many people the reservation is for, and in order to capture their answer, we make use of a `SeekBar` object. The `SeekBar` works in much the same way as a `JSlider` in Swing, and provides the user with a way of selecting the number of people for the reservation, as long as that number is within a range that we define. `SeekBar` in Android is actually built on top of the `ProgressBar` class, and so inherits all of its XML attributes, which will seem a little strange at times. Unfortunately, unlike a `JSlider` or `JProgressBar`, the `SeekBar` class has no minimum value, and since you can't make a reservation for 0 people, we work around this by always adding 1 to the selected value of the `SeekBar` before display. This means that the default value is `1` (setting the displayed value to 2 people).

Most people would make a restaurant reservation for two people, hence the default value of `1`.

In the **How Many People:** label, we put in a `%d`, which is a `printf` marker for where we will put the number of people the reservation is being made for. When the `SeekBar` is manipulated by the user, we'll update the label with the number the user selects using `String.format`. In the "date" and "time" `Button` labels, we want to display the currently selected date and time for the reservation. We set the label in the XML file to the format that we want to display this data in, and we'll parse it later with a standard `java.text.SimpleDateFormat`.

What about internationalization in our previous example? Shouldn't we have put the labels in the `strings.xml` file so that the layout doesn't need to change? The answer is: Yes, if you want to internationalize your user interface. Later, make sure you have all of your display text in an application resource file. However, I strongly recommend fetching the format strings directly from the layout, since it allows you to decouple the format data one additional level.

In the preceding layout, you created `Button` widgets to display the date and time. Why not use a `DatePicker` and `TimePicker` object directly? The answer is: They unfortunately don't fit well into normal layouts. They take up a large amount of vertical space, and don't scale horizontally. If we placed a `DatePicker` and `TimePicker` inline in this user interface, it would look like the following screenshot on the left, while the actual user interface is the screenshot on the right.

As you can see, the `Button` objects give a much cleaner user interface. Thankfully, Android provides us with a `DatePickerDialog` and `TimePickerDialog` for just this sort of situation. When the user taps on one of the `Button` widgets, we'll pop up the appropriate dialog and then update the selected `Button` label when he approves.

While the use of a `Button` and `Dialog` adds at least two more touches to the user interface, it dramatically improves the look and feel of the application. User interfaces that are not properly aligned will irritate users, even if they can't tell why it's irritating. Screens that users find annoying or irritating are screens that they will avoid, or worse—simply uninstall.

Time for action – initializing the reservation tab

In the **Reservation** tab we made use of formatted labels. These labels shouldn't be displayed to the user as-is, but need to be populated with data before we let the user see them. For this, we need to go to our Java code again and build some functionality to remember the format, and populate the label.

1. Open the `ReviewActivity` Java source in your editor or IDE.
2. Below of all the fields you've declared so far, we need to add some more for the **Reservations** tab. Declare a `String` to remember the formatting of the **How Many People:** label:

   ```
   private String peopleLabelFormat;
   ```

3. Then declare a reference to the **How Many People:** label:

   ```
   private TextView peopleLabel;
   ```

4. Declare a `SimpleDateFormat` object for the format of the `date Button`:

```
private SimpleDateFormat dateFormat;
```

5. Declare a reference to the `date Button`:

```
private Button date;
```

6. Add another `SimpleDateFormat` for the format of the `time Button`:

```
private SimpleDateFormat timeFormat;
```

7. Next, declare a `Button` reference for the `time Button` object:

```
private Button time;
```

8. At the end of the `onCreate` method, we need to initialize the **Reservations** tab. Start by assigning out the `peopleLabel` and fetching the `peopleLabelFormat` using the `TextView.getText()` method:

```
peopleLabel = (TextView)findViewById(R.id.people_label);
peopleLabelFormat = peopleLabel.getText().toString();
```

9. Then fetch the `date Button` reference and its label format:

```
date = (Button)findViewById(R.id.date);
dateFormat = new SimpleDateFormat(date.getText().toString());
```

10. Do the same for the `time Button` and its label format:

```
time = (Button)findViewById(R.id.time);
timeFormat = new SimpleDateFormat(time.getText().toString());
```

11. Now we need to populate the `Button` objects with a default date and time, and for this we need a `Calendar` object:

```
Calendar calendar = Calendar.getInstance();
```

12. If it's later than 4:00p.m., it's likely that the reservation should be made for the next day, so we add one day to the `Calendar` if this is the case:

```
if(calendar.get(Calendar.HOUR_OF_DAY) >= 16) {
    calendar.add(Calendar.DATE, 1);
}
```

13. Now we set the default time of day for a reservation on the `Calendar` object:

```
calendar.set(Calendar.HOUR_OF_DAY, 18);
calendar.clear(Calendar.MINUTE);
calendar.clear(Calendar.SECOND);
calendar.clear(Calendar.MILLISECOND);
```

14. Set the label for the `date` and `time` button from the `Calendar` object:

```
Date reservationDate = calendar.getTime();
date.setText(dateFormat.format(reservationDate));
time.setText(timeFormat.format(reservationDate));
```

15. Now we need the `SeekBar` so that we can fetch its default value (as declared in the layout application resource):

```
SeekBar people = (SeekBar)findViewById(R.id.people);
```

16. Then we can use the label format, and the `SeekBar` value to populate the **How Many People** label:

```
peopleLabel.setText(String.format(
            peopleLabelFormat,
            people.getProgress() + 1));
```

Now we have the various formats in which the labels need to be displayed on the user interface. This allows us to regenerate the labels when the user changes the reservation parameters.

What just happened?

The **Reservations** tab will now be populated with the default data for a reservation, and all the formatting in the labels has disappeared. You will probably have noticed the many calls to `toString()` in the previous code. Android `View` classes generally accept any `CharSequence` for labels. This allows for much more advanced memory management than the `String` class, as the `CharSequence` may be a `StringBuilder`, or may facade a `SoftReference` to the actual text data.

However, most traditional Java APIs expect a `String`, not a `CharSequence`, so we use the `toString()` method to make sure we have a `String` object. If the underlying `CharSequence` is a `String` object, the `toString()` method is a simple `return this;` (which will act as a type cast).

Again, to work around the fact that the `SeekBar` doesn't have a minimum value, we add `1` to its current value in the last line, when we populate the `peopleLabel`. While the `date` and `time` formats are stored as a `SimpleDateFormat`, we store the `peopleLabelFormat` as a `String` and will run it through `String.format` when we need to update the label.

Time for action – listening to the SeekBar

The user interface is now populated with the default data. However, it's not interactive at all. If you drag the `SeekBar` the **How Many People:** label will remain at its default value of **2**. We need an event listener to update the label when the `SeekBar` is used.

1. Open the `ReviewActivity` Java source in your editor or IDE.
2. Add `SeekBar.OnSeekBarChangeListener` to the interfaces that `ReviewActivity` implements.
3. In `onCreate`, after fetching the `SeekBar` with `findViewById`, set its `OnSeekBarChangeListener` to `this`:

```
SeekBar people = (SeekBar)findViewById(R.id.people);
people.setOnSeekBarChangeListener(this);
```

4. Implement the `onProgressChanged` method to update `peopleLabel`:

```
public void onProgressChanged(
            SeekBar bar, int progress, boolean fromUser) {

    peopleLabel.setText(String.format(
            peopleLabelFormat, progress + 1));
}
```

5. Implement an empty `onStartTrackingTouch` method:

```
public void onStartTrackingTouch(SeekBar bar) {}
```

6. Implement an empty `onStopTrackingTouch` method:

```
public void onStopTrackingTouch(SeekBar bar) {}
```

The `String.format` method is a common method of placing parameters in a localized string in Android. While this is rather different to the normal `java.text.MessageFormat` class, it's the preferred method in Android (although `MessageFormat` is still supported).

What just happened?

When you reinstall the application in the emulator, you'll now be able to use `SeekBar` to select the number of people that the reservation is to be made for. While we didn't implement the `onStartTrackingTouch` or `onStopTrackingTouch` methods, they can be extremely useful if you hide the actual status value by default. For example, you could use a `Dialog` containing icons of people to inform the user how many people the reservation is for. When they touch the `SeekBar`—display the `Dialog`, and then when they release the `SeekBar`—hide the `Dialog` again.

Time for action – selecting date and time

We've made the `SeekBar` work as expected, but what about the `date` and `time` `Button` widgets? When the users touch them, they expect to be able to select a different date or time for their reservation. For this we'll need a good old `OnClickListener`, the `DatePickerDialog` and `TimePickerDialog` classes.

1. Open the `ReviewActivity` Java source in your editor or IDE again.
2. Add `View.OnClickListener`, `DatePickerDialog.OnDateSetListener`, and `TimePickerDialog.OnTimeSetListener` to the interfaces that `ReviewActivity` implements. Your class declaration should now look something like this:

```
public class ReviewActivity extends TabActivity
        implements ViewSwitcher.ViewFactory,
        Runnable,
        AdapterView.OnItemSelectedListener,
        SeekBar.OnSeekBarChangeListener,
        View.OnClickListener,
        DatePickerDialog.OnDateSetListener,
        TimePickerDialog.OnTimeSetListener {
```

3. Implement a utility method to parse a `CharSequence` into a `Calendar` object with a specified `SimpleDateFormat`:

```
private Calendar parseCalendar(
        CharSequence text, SimpleDateFormat format) {
```

4. Open a `try` block to allow handling of parse errors if the `CharSequence` is not formatted according to the `SimpleDateFormat`.
5. Parse the `CharSequence` into a `Date` object:

```
Date parsedDate = format.parse(text.toString());
```

6. Then create a new `Calendar` object:

```
Calendar calendar = Calendar.getInstance();
```

7. Set the time on the `Calendar` object to the time in the `Date` object:

```
calendar.setTime(parsedDate);
```

8. Return the parsed `Calendar` object:

```
return calendar;
```

9. You'll need to `catch(ParseException)` in this method. I recommend wrapping it in a `RuntimeException` and re-throwing it:

```
catch(ParseException pe) {
    throw new RuntimeException(pe);
}
```

10. In the `onCreate` method, after setting the labels of the `date` and `time` `Button` widgets, set their `OnClickListener` to `this`:

```
date.setText(dateFormat.format(reservationDate));
time.setText(timeFormat.format(reservationDate));
date.setOnClickListener(this);
time.setOnClickListener(this);
```

11. Implement the `onClick` method to listen for when the user taps the `date` or `time` `Button`:

```
public void onClick(View view) {
```

12. Use the `View` parameter to determine if the clicked `View` is the `date` `Button`:

```
if(view == date) {
```

13. If so, use the `parseCalendar` method to parse the current value of the `date` `Button` widget's label:

```
Calendar calendar = parseCalendar(date.getText(), dateFormat);
```

14. Create a `DatePickerDialog` and populate it with the date in the `Calendar`, then `show()` the `DatePickerDialog`:

```
new DatePickerDialog(
        this, // pass ReviewActivity as the current Context
        this, // pass ReviewActivity as an OnDateSetListener
        calendar.get(Calendar.YEAR),
        calendar.get(Calendar.MONTH),
        calendar.get(Calendar.DAY_OF_MONTH)).show();
```

15. Now check if the user has clicked on `View` `Button` instead of `date`:

```
else if(view == time) {
```

16. If so, parse a `Calendar` using the `time` `Button` widget's label value:

```
Calendar calendar = parseCalendar(time.getText(), timeFormat);
```

17. Now create a `TimePickerDialog` with the selected time, then `show()` the new `TimePickerDialog` to the user:

```
new TimePickerDialog(
```

```
        this, // pass ReviewActivity as the current Context
        this, // pass ReviewActivity as an OnTimeSetListener
        calendar.get(Calendar.HOUR_OF_DAY),
        calendar.get(Calendar.MINUTE),
        false) // we want an AM / PM view; true = a 24hour view
        .show();
```

18. Now implement the `onDateSet` method to listen for when the user accepts the `DatePickerDialog` with a new date selected:

```
public void onDateSet(
        DatePicker picker, int year, int month, int day)
```

19. Create a new `Calendar` instance to populate the date into:

```
Calendar calendar = Calendar.getInstance();
```

20. Set the year, month, and day on the Calendar:

```
calendar.set(Calendar.YEAR, year);
calendar.set(Calendar.MONTH, month);
calendar.set(Calendar.DAY_OF_MONTH, day);
```

21. Set the label of the `date` `Button` to the formatted `Calendar`:

```
date.setText(dateFormat.format(calendar.getTime()));
```

22. Implement the `onTimeSet` method to listen for when the user accepts the `TimePickerDialog` after selecting a new time:

```
public void onTimeSet(TimePicker picker, int hour, int minute)
```

23. Create a new `Calendar` instance:

```
Calendar calendar = Calendar.getInstance();
```

24. Set the `Calendar` object's `hour` and `minute` fields according to the parameters given by the `TimePickerDialog`:

```
calendar.set(Calendar.HOUR_OF_DAY, hour);
calendar.set(Calendar.MINUTE, minute);
```

25. Set the label of the `time` `Button` by formatting the `Calendar` object:

```
time.setText(timeFormat.format(calendar.getTime()));
```

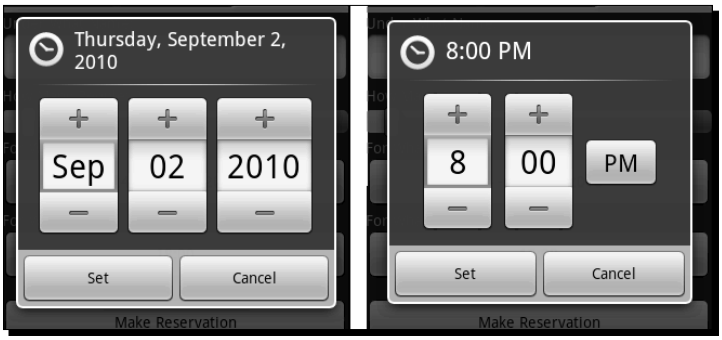
Having stored the format for the `date` and `time` objects, we can now display the values selected by the user in the `Button` widgets. When the user has selected a new date or time we update the `Button` labels to reflect the new selections.

What just happened

If you install and run the application in the emulator, you can now tap on either the `date` or `time` `Button` widgets, and you will be greeted by a modal `Dialog` allowing you to select a new value. Beware of overusing modal `Dialog` widgets, because they block access to the rest of your application. You should avoid using them for displaying status messages as they effectively render the rest of the application useless during that time. If you do display a modal `Dialog`, ensure that there is some way for the user to dismiss the `Dialog` without any other interaction (that is, a **Cancel** button or something similar).

The first advantage to using a `DatePickerDialog` and `TimePickerDialog` comes from the fact that both include **Set** and **Cancel** buttons. This allows the user to manipulate the `DatePicker` or `TimePicker`, and then cancel the changes. If you used an inline `DatePicker` or `TimePicker` widget, you could provide a **Reset** button, but this would take up additional screen space, and generally would seem out-of-place (until it's actually needed).

Another advantage of the `DatePickerDialog` over the `DatePicker` widget is that the `DatePickerDialog` displays a long-format of the selected date in it's title area. This long-format date generally includes the day of the week that the user has currently selected. The "day of the week" is a field that is noticeably missing from the `DatePicker` widget, which makes it surprisingly difficult to use. Most people think in terms of "next Thursday", instead of "the 2nd of August, 2010." Having the day of the week visible makes the `DatePickerDialog` a much better choice for date selection than an inline `DatePicker`.



Creating complex layouts with Include, Merge, and ViewStubs

In this chapter we've built a single layout resource with three different tabs in it. As a result of this, the `main.xml` file has become quite large and hence, more difficult to manage. Android provides several ways in which you can break up large layout files (such as this one) into smaller chunks.

Using Include tags

The `include` tag is the simplest one to work with. It's a straight import of one layout XML file into another. For our previous example, we could separate each tab out into its own layout resource file, and then `include` each one in the `main.xml`. The `include` tag has only one mandatory attribute: `layout`. This attribute points to the layout resource to be included. This tag is not a static or compile-time tag, and so the included layout file will be selected through the standard resource selection process. This allows you to have a single `main.xml` file, but then add a special `reviews.xml` file (perhaps for Spanish).

The `layout` attribute on the include tag is **not** prefixed with the `android` XML namespace. If you attempt to use the `layout` attribute as `android:layout`, you won't get any compile-time errors, but your application will strangely fail to run.

The `include` element can also be used to assign or override several attributes of the root included element. These include the element `android:id`, and any of the `android:layout` attributes. This allows you to reuse the same layout file in several parts of your application, but with different layout attributes and a different ID. You can even `include` the same layout file several times on the same screen, but with a different ID for each instance. If we were to change our `main.xml` file to include each of the tabs from other layout resources, the file would look something more like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/
android"
             android:layout_width="fill_parent"
             android:layout_height="fill_parent">
    <include
        android:id="@+id/review"
        layout="@layout/review"/>
    <include
        android:id="@+id/photos"
        layout="@layout/photos"/>
    <include
        android:id="@+id/reservation"
        layout="@layout/reservations"/>
</FrameLayout>
```

Merging layouts

The `include` element is very fine and well when you want to include a single `View` or `ViewGroup` into a larger layout structure. However, what if you want to include multiple elements into a larger layout structure, without implying the need for a root element in the included structure? In our example each tab needs a single root `View` in order that each tab carries a single and unique ID reference.

However, having an additional `ViewGroup` just for the sake of an `include` can adversely affect the performance of large layout trees. In this case, the `merge` tag comes to the rescue. Instead of declaring the root element of a layout as a `ViewGroup`, you can declare it as `<merge>`. In this case, each of `View` objects in the included layout XML will become direct children of the `ViewGroup` that includes them. For example, if you had a layout resource file named `main.xml`, with a `LinearLayout` that included a `user_editor.xml` layout resource, then the code would look something like this:

```
<LinearLayout android:orientation="vertical">
    <include layout="@layout/user_editor"/>
    <Button android:id="@+id/save"
            android:text="Save User"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"/>
</LinearLayout>
```

The simple implementation of the `user_editor.xml` looks something like this:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView android:text="User Name:"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"/>
    <EditText android:id="@+id/user_name"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"/>
    <!-- the rest of the editor -->
</LinearLayout>
```

However, when this is included into the `main.xml` file, we embed the `user_editor.xml` `LinearLayout` into the `main.xml` `LinearLayout`, resulting in two `LinearLayout` objects with identical layout attributes. Obviously it would be much better to simply put the `TextView` and `EditView` from `user_editor.xml` directly into the `main.xml` `LinearLayout` element. This is exactly what the `<merge>` tag is used for. If we now re-write the `user_editor.xml` file using the `<merge>` tag instead of a `LinearLayout`, it looks like this:

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">
    <TextView android:text="User Name:"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"/>
```

```
    <EditText android:id="@+id/user_name"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"/>
    <!-- the rest of the editor -->
</merge>
```

Note that we no longer have the `LinearLayout` element, instead the `TextView` and `EditView` will be added directly to the `LinearLayout` in the `main.xml` file. Beware of layouts that have too many nested `ViewGroup` objects, as they are almost certain to give trouble (more than about ten levels of nesting is likely to cause your application to crash!). Also be careful with layouts that have too many `View` objects. Again, more than 30 is very likely to cause problems or make your application crash.

Using the ViewStub class

When you load a layout resource that includes another layout, the resource loader will immediately load the included layout into the memory, in order to attach it to the layout you've requested. When `main.xml` is read in by the `LayoutInflator`, so are the `reviews.xml`, `photos.xml`, and `reservations.xml` files. In situations with very large layout structures, this can consume a huge amount of your application memory, and even cause your application to crash. The Android API includes a specialized `View` named `ViewStub` which allows lazy-loading of layout resources.

A `ViewStub` is by default a zero-by-zero sized empty `View`, and when it's specialized, `inflate()` method is invoked. It loads the layout resource and replaces itself with the loaded `View` objects. This process allows the `ViewStub` to be garbage-collected as soon as its `inflate()` method has been called.

If we were to make use of a `ViewStub` in the example, you would need to lazy-initialize the content of a tab when it is selected by the user. This also means that none of the `View` objects in a tab would exist until that tab has been selected. While using a `ViewStub` is a bit more work than a straight `include`, it can allow you to work with much larger and more complex layout structures than would otherwise be possible.

Any layout attributes set on a `ViewStub` will be passed on to its inflated `View` object. You can also assign a separate ID to the inflated layout. If we wanted to include each of our tabs in a `ViewStub`, the `main.xml` file would look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<FrameLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <ViewStub android:id="@+id/review"
```

```
                android:inflatedId="@+id/inflated_review"
                android:layout="@layout/review"/>
        <ViewStub android:id="@+id/photos"
                android:inflatedId="@+id/inflated_photos"
                android:layout="@layout/photos"/>
        <ViewStub android:id="@+id/reservations"
                android:inflatedId="@+id/inflated_reservations"
                android:layout="@layout/reservations"/>
    </FrameLayout>
```

Note that unlike the `include` tag, the `ViewStub` requires the `android` XML namespace for its `layout` attribute. After you `inflate()` one of the `ViewStub` objects, it will no longer be available by its original `android:id` reference. Instead, you will be able to access the inflated layout object using the `android:inflatedId` reference.

Have a go hero – separate the tabs

Extract each of the tabs into its own layout resource file, and use the `include` tag to load each of them. This shouldn't require any changes to the Java source code.

For more of a challenge, try using `ViewStub` objects instead of the `include` tag. This will require you to break up the `onCreate` method and listen for when tabs are clicked. For this you'll need to use `TabHost.OnTabChangeListener` to know when to load a specific tab's content.

Summary

Tabs are a great way of breaking an `Activity` into different areas of work. With limited screen real estate, they are a great way to make an `Activity` more accessible to the user. They also have a performance impact since only one tab is rendered on the screen at a time.

The `RatingBar` and `SeekBar` are two different methods of capturing, or displaying numeric data to the user. While they are closely related, and both function in the same way, each class is used to address different types of data. Keep in mind the limitations of both of these, before deciding whether and where to use them.

The `Gallery` class is brilliant for allowing the user to view a large number of different objects. While in this example we used it to simply display thumbnails, it could be used as a replacement for tabs in a web browser by displaying a list of page thumbnails above the actual browser view. All you need to do to customize its function is to change the `View` objects returned from the `Adapter` implementation.

When it comes to date and time capturing, try to stick to using the `DatePickerDialog` and `TimePickerDialog` instead of their inline counterparts (unless you have good reason). The use of these `Dialog` widgets helps you conserve screen space and improve the user experience. When they open a `DatePickerDialog` or `TimePickerDialog`, they have better access to the editor than you can generally provide as part of your user interface (especially on a device with a small screen).

In the next chapter, we'll take a closer look at `Intent` objects, the activity stack, and the lifecycle of an Android application. We'll investigate how `Intent` objects and the activity stack can be used as a way to keep applications more usable. Also, we shall learn about improving the reuse of `Activity` classes.

Where to buy this book

You can buy Android User Interface Development: Beginner's Guide from the Packt Publishing website: `https://www.packtpub.com/android-user-interface-development-beginners-guide/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com