Charlie Collins
Michael Galpin
Matthias Käppler

Android IN PRACTICE

Includes 91 Techniques

MANNING

SAMPLE CHAPTER

Android in Practice

by Charlie Collins,
Michael D. Galpin,
and Matthias Käppler

Chapter 4

brief contents

Part 2

Real world recipes

In the second part of *Android in Practice*, you'll move beyond the basics and build many complete example applications that will cover, in depth, many of the most common application features and requirements. In chapter 4, you'll start with the user interface. This will cover resources and views, and additional concepts such as using styles and themes, and supporting different screen sizes. Chapter 5 will show you how to effectively multitask on Android using background services. Chapter 6 will continue the theme by presenting an overview of threads and concurrency including working with threads and handlers, asynchronous tasks, and more. Chapter 7 will then change gears to focus on storing data locally. Here, you'll use the file system, the internal and external storage, shared preferences, and a database. Chapter 8 will shift to sharing data between applications using content providers. Here, you'll both consume content from other applications, and learn how to create your own provider and expose data to others. Chapter 9 will take your data beyond the local device and delve into networking. Here, you'll learn how to cope with the instability that is inherent in mobile data connections, as well as how to work with HTTP and web services using JSON and XML. Chapter 10 will then navigate into location-based services and working with location providers. Here, you'll learn how to determine what providers are available and how to switch between them, and how to work with map based data and activities. Chapter 11 will bring in multimedia, where you'll work with audio and video, and learn a little about files, resources, and animation too. Chapter 12 will extend the animation and visual elements to teach you about 2D and 3D drawing, including working with the canvas, and using OpenGL.

4 Getting the pixels perfect

In this chapter

- Rendering views
- Creating layouts
- Working with themes and styles
- Creating interfaces for mobile apps

I don't know answers, I just do eyes. You Nexus, huh? I design your eyes.

—Blade Runner

This chapter is about all things visual. We'll see how views are laid out in a hierarchy and drawn to screen in several passes. We'll also explore more about the layout managers Android provides, and how layout parameters are applied. We'll then learn how to use themes and styles to customize an application, how to draw custom buttons and other window elements, and how to make user interfaces scale to different devices. Finally, most importantly, we'll see how to deal with common problems arising in all of these areas along the way. Be aware that this is one of the longest chapters in this book, but don't fret! It's also one of the most fundamental and widely applicable, so you'll find plenty of material here that'll make your Android developer life easier.

4.1 The MyMovies application

To carry us through the examples in this chapter, we'll be starting a new sample application, *MyMovies*. The DealDroid application we introduced in chapter 2 served us well to demonstrate most of Android's core elements, but in fairness wasn't the prettiest Droid to look at. Smartphone users are humans, not Androids, and humans are visual beings—we love a bit of bling in our applications! That's why this time around, we'll focus on presentation and deal less with functionality.

GRAB THE PROJECT: MYMOVIES You can get the source code for this project, and/or the packaged APK to run it, at the *Android in Practice* code website. Because some code listings here are shortened to focus on specific concepts, we recommend that you download the complete source code and follow along within Eclipse (or your favorite IDE or text editor).

Source: http://mng.bz/7JxQ, APK File: http://mng.bz/26DZ

The task is to write a simple application that keeps track of your personal movie collection. To achieve that, we'll present the user with a list of movie titles, each of which can be flagged as *have* or *don't have* by tapping the list entry. As mentioned earlier, we'll keep it simple featurewise. In later chapters, we'll make it truly useful by extending the feature set introduced here. Using the example application, we'll learn how to create highly customized user interfaces, which not only work and scale well, but also look good. To whet your appetite, figure 4.1 shows a screen shot of the application you'll complete by the end of this chapter.

As you can see, the list of movies that are known to the application takes the majority of the screen. We'll accomplish this by using a `ListView` (which we met in chapter 2) that has been customized to add a translucent background and a gradient list selector with rounded corners that changes color when it's clicked. We've also added a background image and a title image that automatically scale with the screen width and orientation. These changes are by no means specific or limited to this particular application. Anything you learn in this chapter can be applied to your own applications. But first things first: let's make sure we understand what's happening under the hood when Android renders a user interface. Therefore, before discussing the MyMovies implementation, we'll discuss view rendering, layouts, and layout managers in detail.

Figure 4.1 The MyMovies application title screen. Note how we've customized the user interface to use features such as a translucent list selector.

4.2 View hierarchies and rendering

View rendering is an integral aspect of any application that involves a UI. We all love nifty-looking applications, but your application will spend a lot of time drawing its various interface elements. Therefore, it's important to understand what happens under the hood so you can avoid performance pitfalls. It's bad if your applications are beautiful, but slow. Though we've already introduced and used views, we're going to expand on their features. Specifically, we'll explain how they're organized, how they're drawn, and what sort of things you should keep an eye on in order to keep the UI snappy.

4.2.1 View hierarchies

We know that views in Android are typically defined in a declarative fashion using XML. XML structures information into trees; all nodes extend and branch from a single root node. It's no coincidence that Android employs this kind of representation, apart from XML's general popularity. Internally, the user interface of any Android application is represented as a tree of `View` objects. This is known as the *view hierarchy* or *view tree*. At the root of every view tree—and every application UI—sits a single `DecorView`. This is an internal framework class that you can't use directly; it represents the phone window you're currently looking at. The `DecorView` itself consists of a single `LinearLayout`, which branches into two `FrameLayouts`: one to hold the title section of the currently visible `Activity`, and one to holds its content (`FrameLayouts` block out an area on the screen to display a single item). *Content* here means anything that's defined in the current activity's layout XML. To illustrate, let's examine the XML layout for the MyMovies main screen (res/layout/main.xml):

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

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

</LinearLayout>
```

To understand how the hierarchy of view elements works for this screen, we'll use the *hierarchyviewer* tool that comes with the SDK. You can launch this tool either from the command line, or if you're using the latest version of the ADT, via the Hierarchy View perspective in Eclipse. Either method will connect to a running emulator instance or connected device and then present multiple options about your layouts. Figure 4.2 shows the view hierarchy for the main layout seen earlier.

The single dark box sitting in the center of figure 4.2 is the `LinearLayout` with which we began the XML file. As you can see from the hierarchy, `LinearLayout` has a `FrameLayout` parent for the content node (identified by the `android.R.id.content` resource ID). This is Android's way of representing the content area of the screen—the area

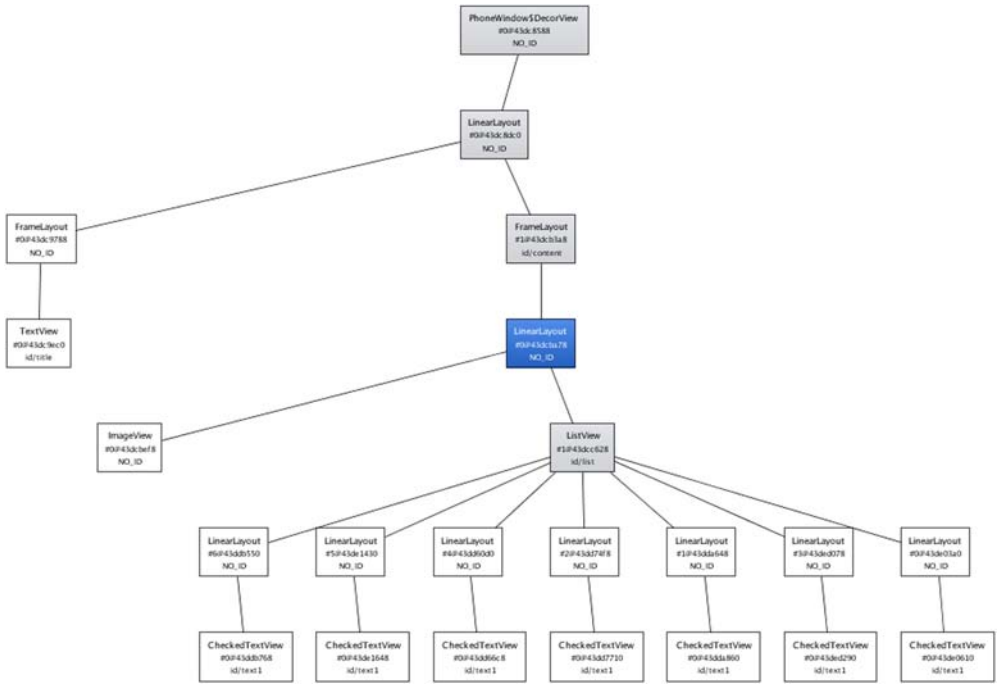

Figure 4.2 The view hierarchy for the MyMovies main layout created using the hierarchyviewer tool. The left branch represents the window's title bar, the right branch the current activity's contents.

which will make up most of your application's user interface. The majority of the time, you'll only be concerned with this branch—anything extending from the content node. The sibling `FrameLayout` for the title node (to the left) is also shown; this layout makes up the window's title bar. Underneath MyMovie's root `LinearLayout`, we see the `ListView` we defined in listing 4.1, which in turn has a child `LinearLayout` for each item in the list.

Whenever an `Activity` is started, its layout view tree is inserted into the application view tree by a call to the `Activity`'s `setContentView(int layoutId)`. This effectively replaces everything beneath the current content node with the view tree identified by `layoutId`, which, as we've seen, is a layout as defined in a layout XML file. The process of loading a layout and merging it into the current view tree is referred to as *layout inflation*. This is done by the `LayoutInflater` class, which resembles a tree growing in nature. Once in a while, a new branch grows, and from that branch grows another branch, and so on. Layouts aren't directly inflated from XML because Android converts XML into an efficient binary format before compilation (you'll learn about Android's build logic in chapter 14).

When a `View` has been inflated, it becomes part of the rendering chain, which means it can be drawn to the screen, unless it's obscured by another view. Android uses a two-pass algorithm to do that, which we're going to look at briefly now.

4.2.2 View rendering

Once a view tree is in memory, Android must draw it. Each view is responsible for drawing itself, but how the view is laid out and positioned on the screen can only be determined by looking at it as part of the whole tree. This is because the position of every view affects the position of the next. In order to figure out where to draw a view and how big it should be, Android must do the drawing in two separate passes: a *measure pass* and a *layout pass*.

MEASURE PASS

During the measure pass, each parent view must find out how big their child views want to be by calling their `measure` method. This includes pushing a measure specification object down the tree that contains the size restrictions imposed by a parent view on a child. Every child must then find out how big it wants to be, while still obeying these restrictions.

LAYOUT PASS

Once all views have been measured, the layout pass is entered. This time, each parent must position every child on the screen using the respective measurements obtained from the measure pass by calling their `layout` method. This process is illustrated in figure 4.3.

The layout and measuring of views happens transparently to the developer, unless you're implementing your own `View`, in which case you must override `onMeasure` and `onLayout` and hence actively take part in the rendering passes. Knowing about the complexity of view rendering makes one thing obvious: drawing views is expensive, especially if many views are involved and the view tree grows large. Unfortunately, your application will spend a fair amount of time in Android's drawing procedures. Views are invalidated and redrawn all the time, either because they're obscured by other views or they change state. There isn't much you can do about this, but what you *can* do is be aware of the overhead when writing your code and try to reduce unnecessary

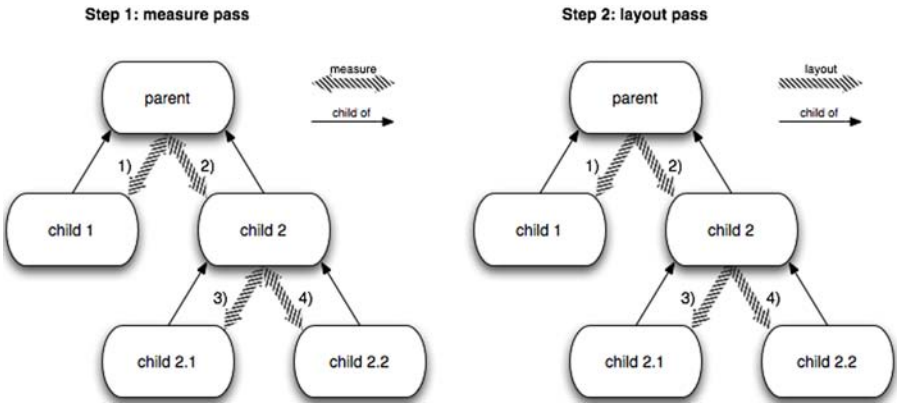

Figure 4.3 Views are rendered using a two-pass traversal of the view tree. Pass one collects dimension specifications (left); pass two does the positioning on the screen (right).

rendering. Table 4.1 lists some best practices for trying to optimize performance when working with views.

Table 4.1 Best practices when working with `Views`

Advice	Why
The cheapest `View` is the one that's never drawn	If a `View` is hidden by default, and only appears in reaction to a user interface event such as a tap/click, you may want to consider using a `ViewStub` instead (a placeholder view). You can also dynamically add/remove a view from the rendering chain by setting its visibility to `View.GONE`.
Avoid `View` cluttering	Along the lines of the previous advice, think twice about using a view and simplify your UI when you can. Doing so will keep screen layouts clean, and improve performance.
Try to reuse `Views` where possible	Often you can avoid extra inflation and drawing by caching and reusing views. This is important when rendering lists, where many items are displayed at once and state changes frequently (like when scrolling the list). The `convertView` and `ViewHolder` pattern can help here, and is covered in technique 1 of this chapter.
Avoid excessive nesting of layouts	Some developers use nested `LinearLayouts` to arrange elements relative to each other. Don't do that. The same result can usually be achieved by using a single `RelativeLayout` or `TableLayout`.
Avoid duplication	If you find yourself copying view definitions in order to use them in more than one layout, consider using the `<include>` tag instead. Similarly, nesting layouts of the same kind is in most cases useless duplication, and can be avoided using the `<merge>` tag.

`View` performance should always be on your mind when working with views and layouts. Some things may be obvious to you if you already have experience with Android, but we wouldn't have mentioned them if we didn't see applications violating these rules on a regular basis. A good idea is to always double-check your layouts for structural weaknesses using the `layoutopt` tool that ships with the SDK. It's by no means the only thing you should rely on, but it's fairly clever about finding smells in your layouts.

Mobile applications are all about user interaction: your application will likely spend most of its time in drawing its various interface elements and reacting to user input, so it's important that you have a solid understanding of what drives your UI. Now that we've seen how views and layouts are organized in memory, and what algorithms Android uses to measure and position views before it draws them on the screen, we'll next turn to more detail about layouts themselves.

4.3 Arranging views in layouts

Whenever you implement the user interface for an `Activity`, you're dealing with that `Activity`'s *layout*. As we've already noted, layouts arrange views on screen. A layout is like a blueprint for a screen: it shows which elements the screen consists of, how they're arranged, what they look like, and so on. Hence, when implementing a screen for your application, thinking about layout is one of the first things you should do. Knowing your layout managers is crucial if you work with designers. You'll probably get mockups or wireframes for each screen, and you should know how to map a design to Android's layout managers.

LAYOUT VERSUS LAYOUT MANAGER

When speaking of *layout,* we mean the set of all views for a single `Activity` as arranged by its layout XML file located in the `res/layout` folder. This is not to be confused with a certain layout class, called a *layout manager.* An `Activity`'s layout may involve more than one layout manager at a time, depending on its complexity. As we've already discussed, a layout manager is another `View` (a `ViewGroup` more precisely) that serves as a container and arranges views in a specific manner.

In the next section, we're going to give you a detailed rundown of general layout anatomy, plus a complete overview of the layout managers Android supports.

4.3.1 Layout anatomy

You've already seen several layouts at this point, such as the Hello Android layout from chapter 1 and the `DealList` layout from chapter 2. We've discussed the basics of these layouts, but we haven't specifically addressed which elements can be placed in layout files and how they're structured overall. We also haven't touched on the parameters and attributes layouts support. We'll look at these aspects now.

COMPOSITION OF LAYOUT FILES

Every layout file starts with an XML preamble, where you can define the file's encoding, typically UTF-8. Like any other XML document, a layout consists of a single root node with zero or more children, depending on whether the root node is a `ViewGroup`, which is the case for all layout managers, or a simple `View`, in which case it must be the only view defined in the layout. Node names correspond to class names, so you can put anything in a layout that's a concrete class inheriting from `android.view.View`. By default, class names are looked up in the `android.view` (`SurfaceView`, `ViewStub`, and so on) and `android.widget` (`TextView`, `ListView`, `Button`, and so on) packages. For other views that aren't part of the framework, such as those you define yourself, you have to use the fully qualified class name instead (such as `com.myapp.MyShinyView`). This becomes particularly important if you want to embed a Google Maps `MapView` (we'll learn about location and `MapView` in chapter 10). This class contains Google proprietary code and isn't distributed along with the core framework. Therefore, you have to use `MapView` using its fully qualified name:

```
<com.google.android.maps.MapView
    android:id="@+id/mapview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:clickable="true"
    android:apiKey="Your Maps API Key"
/>
```

LAYOUT ATTRIBUTES AND PARAMETERS

Every `View` in a layout can take two different kinds of attributes: those that are specific to the view class and its parent classes, and those that are specific to the layout manager it's being placed into. Which attributes a view can take may be obtained from the

view's documentation (or Eclipse's completion). A `TextView`, for instance, defines the `android:text` attribute, which allows you to define its default text value. It also understands the `android:padding` attribute, because that attribute is inherited from Android's `View` base class.

AVAILABLE VIEW ATTRIBUTES You can look up all view attributes exposed by Android in one place in the documentation of the `android.R.attr` class.

Layout parameters are different: you can tell them apart from normal attributes by their `layout_` prefix. They define how a `View` should be rendered while participating in the layout. Unlike normal attributes, which apply to the `View` directly, layout parameters are hints to the view's parent view in the layout, usually a layout manager. Don't confuse a view's parent *view* in a layout with a view's parent *class*; the former is a separate view in the layout in which the view is embedded, whereas the latter refers to the view's type hierarchy. All layout managers, and also some other views such as the `Gallery` widget, define their own layout parameters using an inner class called `LayoutParams`. All `LayoutParams` support the `android:layout_width` and `android:layout_height` attributes, and all layout manager parameters further support the `android:layout_margin` attributes.

MARGIN AND PADDING Margin and padding can also be defined separately for each edge. In that case, define any of these attributes for a view:

- `android:layout_marginLeft`
- `android:layout_marginTop`
- `android:layout_marginRight`
- `android:layout_marginBottom`

The same approach works for the `android:padding` attribute.

Any other parameters are specific to the various `LayoutParams` implementations found across the framework. The width and height parameters are special in two ways: they must *always* be present on any view or Android will throw an exception. Moreover, they can take not only numeric values (pixels), but also two reserved values:

- `fill_parent`—Indicates that the `View` would like to take up as much room as possible inside its parent view. It'll try to grow as big as its parent (minus padding and margins), regardless of how much room its own children occupy. If, for instance, the parent is a square of 100px and neither margins nor padding were defined, the child will be a square of 100px, too. Note that `fill_parent` has been deprecated and is now called `match_parent`. You'll likely want to support older versions of Android, so stick to `fill_parent` until older platform versions disappear.
- `wrap_content`—Indicates that the `View` would like to take only as much room inside its parent, as it needs to fully render its own content. If for instance, the parent is again a square of 100px, and the view's own children only occupy a 50px square, then the view itself will only be a square of 50px.

Now that we've seen several layout files in action, and have touched on how they're composed and the attributes and parameters they support, our next step is to dig further into layouts while we also examine the available layout managers in more detail.

4.3.2 Layout managers

Android currently defines four different layout managers that you can use to arrange views on the screen. You're free to implement your own if you need something more elaborate, but we won't cover that here. They can be divided into structured and unstructured, or by complexity, as summarized by table 4.2.

Table 4.2 Available built-in Android layout managers

Complexity	Unstructured	Structured
Lower	`FrameLayout`	`LinearLayout`
Higher	`RelativeLayout`	`TableLayout`

There's also a fifth layout manager, `AbsoluteLayout`, but it has been deprecated and shouldn't be used, because it doesn't scale to different screen configurations (which is important, as we'll see in section 4.7). With the exception of `AbsoluteLayout`, we're now going to visit each of these types briefly. Let's start with the simplest, `FrameLayout`, and work our way up to `RelativeLayout`, the most complex.

FRAMELAYOUT

This is the simplest of all layout managers. `FrameLayout` doesn't do any real layout work, but serves as a container (a *frame*). `FrameLayout` displays a single child element at a time. It supports multiple children, but they're placed in a stack. Child elements are slapped to the top-left corner and drawn on top of each other in their order of declaration. Does that sound useless to you? To be frank, `FrameLayout` is rarely useful for anything beyond a mere container or box-style layout. One case where it *is* useful is for fitting floating views next to a screen layout (for instance, the ignition library, which is a useful set of Android utilities and enhanced components, uses this technique to render "sticky notes" that can be attached to any widget). The following listing shows how to define a `FrameLayout` holding two `TextView` views.

Listing 4.1 An example FrameLayout containing two TextViews

```
<?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">
  <TextView
      android:layout_width="150px"
      android:layout_height="150px"
      android:background="@android:color/darker_gray"
      />
  <TextView
      android:layout_width="75px"
```

```
        android:layout_height="75px"
        android:background="@android:color/white"
        />
</FrameLayout>
```

Figure 4.4 Two views arranged using `FrameLayout`. Note how one view lays on top of the other and both are pinned to the top-left corner.

You may wonder how these two text views are being rendered as part of this layout. Have a look at figure 4.4, where you can see how they're laid out on top of each other, with the topmost view being the last rendered.

It goes without saying that `FrameLayout` isn't only the simplest, but also the fastest layout manager, so always think twice before jumping to more complex ones! Let's move on to a more useful layout manager, one that we've already seen used a few times, and one that you'll probably spend some quality time with: `LinearLayout`.

LINEARLAYOUT

`LinearLayout` is the most commonly used (sometimes overused) layout manager. It's simple, easy to use, and serves many purposes. As we've noted, in a `LinearLayout`, all views are arranged in lines, either horizontally or vertically, depending on the value of the `android:orientation` attribute. If you don't explicitly specify the orientation, it'll default to horizontal. `LinearLayout` has two additional layout parameters to be used by its children, as seen in table 4.3.

Table 4.3 Layout parameters specific to LinearLayout

Attribute	Effect
`android:layout_weight`	Tells the layout manager how much room this `View` should occupy relative to its siblings. The size of the `View` will be determined based on the relation of all weights to each other. If, for example, all views define the same weight, then the available space will be distributed equally among them. Which axis (width or height) should be affected can be controlled by setting the respective size to a value of `0px`. Note that weights don't have to add up to 1, although it's common to distribute layout weight over all children as fractions of 1 (percentage semantics). The relation between all weights is what matters.
`android:layout_gravity`	Tells the layout manager in which direction the `View` likes to be floated inside its container. This attribute is only meaningful when the `View`'s size on the same axis is either fixed to a constant value or set to `wrap_content`.

In the listing 4.2, we define a layout similar to what we did with `FrameLayout` in listing 4.1, but this time using the `LinearLayout` layout manager. You can also see how we use the `weight` attribute to distribute the available space equally among the two text views in the next listing.

Listing 4.2 An example LinearLayout with weighted children

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <TextView
      android:layout_width="0px"
      android:layout_height="100px"
      android:layout_weight="0.5"
      android:background="@android:color/darker_gray"
      />
  <TextView
      android:layout_width="0px"
      android:layout_height="100px"
      android:layout_weight="0.5"
      android:background="@android:color/white"
      />
</LinearLayout>
```

Figure 4.5 shows how this layout is rendered. Note how the two views take up exactly the same space across the horizontal screen axis. Again, the relation of the weights is all that matters: setting both to 1 would have the same effect, because 0.5 / 0.5 = 1 / 1 = 1.

`LinearLayout` is simple but effective. It's well suited to solving typical layout problems such as arranging buttons next to each other. You can also use it to create grids and tables, but there's a more convenient way to do this: `TableLayout`.

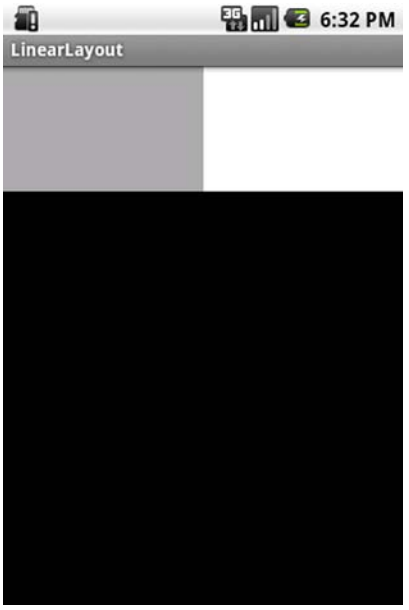

Figure 4.5 Two views arranged using LinearLayout. You can see how both views are sized to take the same amount of space and are aligned horizontally.

TABLELAYOUT

`TableLayout` is a `LinearLayout` (it inherits from it) with additional semantics that make it useful for rendering tables or grids. It introduces a special `View` class called `TableRow`, which serves as a container for table cells. Each cell must consist only of a single `View`. This `View` can again be a layout manager or any other `ViewGroup`. The following listing shows a simple `TableLayout` with a single row.

Listing 4.3 An example TableLayout with a single row

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <TableRow>
    <TextView
        android:layout_width="150px"
        android:layout_height="100px"
        android:background="@android:color/darker_gray"
        />
    <TextView
        android:layout_width="150px"
        android:layout_height="100px"
        android:background="@android:color/white"
        />
  </TableRow>
</TableLayout>
```

Figure 4.6 illustrates what our table definition looks like onscreen. The differences to the `LinearLayout` example are marginal, for the aforementioned reasons.

From figure 4.6, you can see how each `TableRow` child `View` becomes a column in the table. You'd need significantly more code to arrive at the same layout using `LinearLayout`, so remember to use this layout manager whenever you need cells for tables or grids.

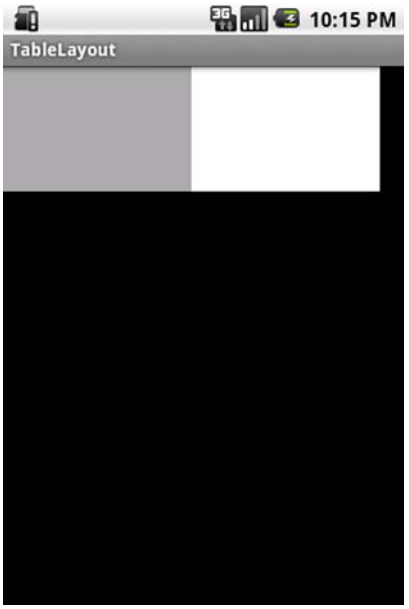

Figure 4.6 Two views arranged using `TableLayout`. It differs from `LinearLayout` only in the way you set up the layout, because `TableLayout` is merely a specialized `LinearLayout`.

At this point, we've seen the first three layout managers Android provides out of the box, and we still don't know how to create truly complex layouts. The layout managers so far are performing simple tasks; at best, they line up views next to each other. If you need more control over how views should be arranged on the screen, then you need to turn to what's arguably the most useful Android layout manager: `RelativeLayout`.

RELATIVELAYOUT

`RelativeLayout` is the most sophisticated of the four layout managers. It allows almost arbitrary placement of views by arranging them relative to each other. `RelativeLayout` exposes parameters that allow its children to reference each other by their IDs (using the `@id` notation). To boil this down, let's look at another sample to see how this works.

Listing 4.4 An example RelativeLayout showing the use of relative attributes

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <TextView android:id="@+id/text_view_1"
      android:layout_width="150px"
      android:layout_height="100px"
      android:background="@android:color/darker_gray"
      />
  <TextView android:id="@+id/text_view_2"
      android:layout_width="150px"
      android:layout_height="100px"
      android:layout_toRightOf="@id/text_view_1"
      android:layout_centerVertical="true"
      android:background="@android:color/white"
      />
</RelativeLayout>
```

1 Attribute for relative positioning

In listing 4.4, `text_view_2` declares that it should be drawn "to the right of" `text_view_1` using the `layout_toRightOf` attribute 1. This demonstrates how `RelativeLayout` views reference each other to define their positions. This is an effective and scalable way of positioning views, but it has a subtle side effect: because you can only reference views using `@id/view_id` that have already been defined, you may find yourself in a situation where you need to shuffle around `View` definitions to reference them. This can quickly become awkward with complex layouts. To solve this problem, you can use a special ID notation to tell the framework to create any IDs that don't yet exist.

HANDLING IDS IN LAYOUTS

Consider again the example from listing 4.4. If `text_view_1` were to reference `text_view_2` instead, you'd have to swap their definitions, or you'd get an error about `text_view_2` not being defined. To avoid this problem, you can use the special `@+id` notation when declaring an ID. When you add the `+`, Android will create a new ID for any `View` that doesn't exist yet.

> **NOTE** Though it may sound awkward, because IDs are used to identify resources, IDs themselves are also resources. This means that as with any other resources such as strings, you can create an ids.xml file in `res/values` and use it to predefine blank IDs (IDs that have a name, but don't reference any other resources yet). The `@+id` notation is then no longer needed to use these IDs, because they already exist (but using it doesn't hurt, either). To define an ID in an `ids.xml` resource file, use the `item` tag:
>
> ```
> <item type="id" name="my_id" />
> ```

What `@+id` does is create a new ID in Android's internal ID table, but only if it doesn't yet exist. If it does already exist, it references the existing ID (it doesn't cause an error). You may use `+` on the same ID as often as you like, but the ID will only be created once (we say it's an *idempotent* operation). This means you can use this notation

to reference a `View` that's defined further down in a layout XML file. Android will create any such ID when you use it for referencing the `View`, and when the `View` is finally defined, it'll reuse it.

To complete our discussion about layout managers, figure 4.7 shows the two `TextView` views from the previous examples arranged using `RelativeLayout`.

With the four built-in layout managers Android provides, you should be able to create almost any layout you need, even fairly complex ones. Once you start building more involved layouts, it's also a good idea to go back to the `layoutopt` tool and let it guide you with any issues it might uncover (we noted this previously, but it's easy to use, and often overlooked, so it bears repeating).

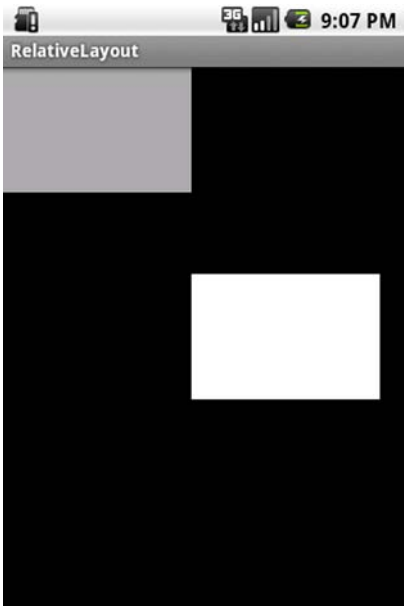

Figure 4.7 Two views arranged using RelativeLayout. Note how we can arrive at almost arbitrary arrangements by specifying all positioning attributes using only relative values.

That covers our views and layouts 101. We feel that we've equipped you with enough background knowledge that you should be able to understand what makes an `Activity` in Android, including the layout containing its visual elements, and even how it's drawn to the screen. It's time to get our hands on some techniques now. We want to show you advanced techniques that will likely become good companions in your day-to-day Android UI development. Let's wrap up our discussion of layouts with our first technique: merging and including layouts.

TECHNIQUE 1 The merge and include directives

A handy optimization for your own sanity, and for layouts, is to not repeat yourself. As your layouts get more and more complex, you'll increasingly find yourself duplicating parts of your layout code because you want to reuse it elsewhere. A good example of this is a `Button` bar with Ok and Cancel buttons. Here's an example of such a layout.

Listing 4.5 A layout with a button bar for Ok and Cancel actions

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <TextView android:text="An activity with a button bar"
      android:layout_width="wrap_content"
```

```
      android:layout_height="wrap_content"
      />
  <!-- a button bar -->
  <LinearLayout android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <Button android:text="@string/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
    <Button android:text="@string/cancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
  </LinearLayout>
</LinearLayout>
```

Things are well and good if you only need these buttons in a single `Activity`, but what happens if you need them in multiple places? As we've seen, every `Activity` defines its own layout file. You can copy the layout section you need into multiple files, but as a good programmer, you know that this is a bad idea. Code duplication leads to programs that are brittle and difficult to maintain.

PROBLEM

You want to share certain parts of a layout with other layout files to minimize code duplication resulting from copying the same code over and over to other layout files.

SOLUTION

When you notice repetitive sections across different layout files, like this one, it's time to check into the special layout `<merge>` and `<include>` elements. These elements allow you to extract commonly used `View` snippets into their own layout files (think of view or layout *components*) that can then be reused in other layouts.

To see how this works, we can extract the button bar related section from listing 4.5 into its own file called button_bar.xml (and put it in `res/layout`) as shown in the following listing.

Listing 4.6 A reusable button bar component defined in its own layout file

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

  <Button android:text="@string/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      />
  <Button android:text="@string/cancel"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      />
</LinearLayout>
```

In order to pull one layout section or component into another file you can then use the `<include>` element:

```
<include layout="@layout/button_bar" />
```

That's it! The `include` element doesn't take any specific parameters other than the layout. Nevertheless, if you want, you can pass it custom layout parameters or a new ID to override the attributes defined for the root view of the layout you're including.

> **INCLUDE AND LAYOUT ATTRIBUTES GOTCHA** When overriding `layout_width` or `layout_height` using include, remember to always override both. If, for example, you only override `layout_width`, but not `layout_height`, Android will silently fail and ignore any `layout_*` overridden settings. This isn't well documented, and somewhat controversial, but easy to work around once you know what's going on.

One question remains: what happens if you want to include views that don't have a common parent `View` or you want to include a layout in different kinds of parent views? You'd think that you could get rid of the parent `LinearLayout` and redefine `button_bar.xml` from listing 4.6 as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<Button android:text="@android:string/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />
<Button android:text="@android:string/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />
```

Unfortunately, that's impossible, because that's not a valid XML document. Remember that XML documents are trees, and a tree always has a root. This one doesn't, so it'll fail. Android has a solution: the `<merge>` element is a placeholder for whatever parent `View` the views in the (partial) layout will be included into. This means we can rewrite the previous snippet as follows to make it work:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <Button android:text="@android:string/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      />
  <Button android:text="@android:string/cancel"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      />
</merge>
```

Think of `<merge>` as a wildcard: it's a glue point, and whatever the layout it wraps will be inserted into (a `LinearLayout`, `RelativeLayout`, and so on) will replace the

<merge> node at runtime. You can use this technique anytime you'd otherwise have to include a View under another View of the same kind (which is duplication, and hence discouraged).

DISCUSSION

We urge you to internalize this technique and apply it to your layout code whenever you can. It'll keep complex layout files clean and easy to read, and significantly reduce code duplication and hence maintenance work. A fundamental engineering principle is to never encode the same information more than once in your application, commonly known as the *DRY principle* (don't repeat yourself). That being said, <merge> and <include> help keep your layouts DRY.

One disadvantage of chopping up your layouts like this is that Android's graphical layout tool in Eclipse will sometimes get confused and not render the preview correctly. On the other hand, it should be a question of time until Android's tool support improves enough to preview even layouts that are merged together in complex ways.

With this first little technique, we conclude our discussion about layouts in Android. Remember how we promised to code a full application—one that manages movie titles and that looks good? We keep our promises. Because MyMovies, like the DealDroid, is made up from a list view, we're going to come back one more time to list views and adapter. There's more to them then you may have expected.

4.4 Expanding on ListView and Adapter

We've already seen how a ListView can be used with an Adapter in chapter 2. That example was simple: we showed a list of deal items, but the data behind that list was static. Once it was put behind the list, it didn't change. For MyMovies, we'd like to be able to tick off Movies that are in our collection. For this to happen, we need to expand on list views and adapters. We now need a stateful list that includes elements that can be in two states: checked or not checked. Moreover, we're going to spice things up by showing you how to add header and footer elements to lists. Along the way, we'll also explore a few optimizations, such as the ViewHolder pattern, which will make your list render significantly faster and hence scroll more smoothly.

First things first. In order to maintain the checked state we'll need for MyMovies, we have to store that the user owns a movie when it's selected from the list, and remove it when it's unselected. For this example, we won't bother to persist that information to a database or file because we haven't yet introduced the mechanisms to do so. This means that for now, all movies we'll add to the "collection" by tapping them will be lost when we restart the application. This is intentional at this point, so we can stay focused on understanding how adapters can maintain state. Don't worry—we'll learn about saving information to files and databases, and more, in chapter 7.

To see how the views for our ListView are bound to the data source we'll use a static movies file, via our Adapter. For now, we need to return to ListActivity and review the code for MyMovies.

TECHNIQUE 2 Managing a stateful list

Recall from listing 4.1 that the main MyMovies screen is composed of a `ListActivity` that takes up the entire screen with a single `ListView`. Also, each movie in the list contains a check box that can be toggled. This toggle is a simple example of maintaining state between our model and our views. The question is: where do we store this state? And how do we reflect updates to this state in the `ListView`?

PROBLEM

You have a list that's backed by data coming from an adapter, and you want either changes coming from the view (such as from a list item click) to be written back into the data source, or changes to the data reflected back to the view.

SOLUTION

When having to maintain dynamic data that can change in reaction to view events (or any other event), you'll have to create your own adapter implementation that performs the following tasks:

- If the data wrapped by the adapter changes, it must inform the view about these changes so it can redraw itself.
- If the user interacts with the view in a way that's supposed to update the data, we must inform the adapter about this and make the according changes to the data source.

To see how this works, we'll start with the seemingly sparse code for the main MyMovies screen, as shown in the following listing. Then, we'll move on to the custom adapter.

Listing 4.7 The MyMovies.java ListActivity class file

```
public class MyMovies extends ListActivity {

   private MovieAdapter adapter;

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

      setContentView(R.layout.main);

      ListView listView = getListView();

      this.adapter = new MovieAdapter(this);
      listView.setAdapter(this.adapter);
      listView.setItemsCanFocus(false);
   }

   @Override
   protected void onListItemClick(ListView l, View v,
                                  int position, long id) {
      this.adapter.toggleMovie(position);
      this.adapter.notifyDataSetChanged();
   }
}
```

1 Extend ListActivity

2 Include Adapter

3 Set Adapter on ListView

Our MyMovies main screen works similarly to the other activities we've seen up to this point, including the fact that it extends `ListActivity` (1) (which we first saw in chapter 2). Also similar to chapter 2, this `Activity` includes an `Adapter` (2). Before going further, remember that we use `ListActivity` because it takes care of many of the details of managing a `ListView`. This includes things such as easy access to `ListView` via `getListView`, and easy click handling with `onListItemClick`. Also, we again set our `Adapter` into our `ListView` to provide the data source for the items the list will handle (3).

Next, we need to visit the data source and custom `MovieAdapter` our `ListView` will be using. Before setting up the adapter, let's look at the source of our movie data. We could fetch data from the Internet (some web services do that kind of thing, as we saw with the DealDroid, and will see in even more detail in chapter 9), but let's keep it simple for this example. We'll take IMDB's top-100 movies and store them as an array resource in our application. To do that, we create a new file called movies.xml in `res/values` with the following structure:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="movies">
    <item>The Shawshank Redemption (1994)</item>
    <item>The Godfather (1972)</item>
    <item>The Godfather, Part II (1974)</item>
    <item>The Good, the Bad, and the Ugly (1966) (It.)</item>
    ...
  </string-array>
</resources>
```

Recalling what we learned about resources in chapter 2, we can now reference this array in our application as `R.array.movies`. Note that we could've hard-coded the list as a plain old Java array in one of our application classes. But using Android's resource mechanism gives us the advantage of both having full control over when we want to load the data into memory, and at the same time keeping our application code clean. After all, application code is supposed to contain logic, not data.

The next step is to get this data to show in the `ListView`. Because we're dealing only with an array, Android's `ArrayAdapter` class is a perfect choice to implement our `MovieAdapter` (which we assigned to the `ListView` in listing 4.7). `MovieAdapter` is where we'll track the movies a user adds and provide an interface to its state so the check box view can update accordingly. The following listing shows how this is implemented.

Listing 4.8 The MovieAdapter keeps track of selected movies

```
public class MovieAdapter extends ArrayAdapter<String> {

   private HashMap<Integer, Boolean> movieCollection =
            new HashMap<Integer, Boolean>();

   public MovieAdapter(Context context) {
```

(1) Extend ArrayAdapter

(2) Include HashMap

```
        super(context, R.layout.movie_item,
            android.R.id.text1, context
                  .getResources().getStringArray(R.array.movies));
    }

    public void toggleMovie(int position) {
        if (!isInCollection(position)) {
            movieCollection.put(position, true);
        } else {
            movieCollection.put(position, false);
        }
    }

    public boolean isInCollection(int position) {
        return movieCollection.get(position) == Boolean.TRUE;
    }

    @Override
    public View getView(int position, View convertView,
        ViewGroup parent) {

        View listItem = super.getView(position, convertView, parent);

        CheckedTextView checkMark = null;
        ViewHolder holder = (ViewHolder) listItem.getTag();
        if (holder != null) {
            checkMark = holder.checkMark;
        } else {
            checkMark = (CheckedTextView)
                listItem.findViewById(android.R.id.text1);
            holder = new ViewHolder(checkMark);
            listItem.setTag(holder);
        }

        checkMark.setChecked(isInCollection(position));

        return listItem;
    }

    private class ViewHolder {
        protected final CheckedTextView checkMark;

        public ViewHolder(CheckedTextView checkMark) {
            this.checkMark = checkMark;
        }
    }
}
```

3 Use super constructor

4 Toggle movie state

5 Override getView

6 Try to get ViewHolder

7 Establish view

8 Set up ViewHolder

9 Implement inner ViewHolder class

The first thing to note about `MovieAdapter` is that it extends `ArrayAdapter` (1). By doing this, we need only implement the adapter methods we're interested in, and we don't have to reinvent the wheel. From there we also see that `MovieAdapter` includes a local `HashMap` for storing movie state data (2). The user's movie collection is modeled as a mapping from positions in the movie list to Boolean values (`true` meaning the user owns the movie). Again, this state is transient, so once the user exits the application it'll be lost, but you can see how we could reference a database, the filesystem, or any other storage mechanism here if we wanted to.

Next, we see that the constructor makes a call to the `ArrayAdapter` super constructor, like any good Java subclass, and there provides the context, layout to use for each item, the ID of a `TextView` to populate for each item, and the initial data collection (3). The first method we see is `toggleMovie`, which is used to update the model's state (4). Next is the all-important `getView` method that we first saw in chapter 2. This method returns the view needed for each item, and is called whenever a list item must be (re)drawn (5).

Redrawing of the list items happens frequently, for example when scrolling through the list. Because this behavior can have a lot of overhead, we use the ViewHolder pattern to optimize the way our `ListView` gets and populates list items. The idea is to eliminate extra calls to `findViewById` because that's a relatively expensive method. To do so, we cache `findViewById`'s result in a `ViewHolder` object. `ViewHolder` is an internal class that we've created to hold the `CheckedTextView` we need (9).

But how do we associate the cached view with the current list item? For that, we use a handy method called `getTag`. This method allows us to associate arbitrary data with a view. We can leverage that method to cache the view holder itself, which in turn caches the view references. Call the `getTag` method on the current `listItem` (we let the super class decide whether a new one is created or whether it's recycled from the `convertView`) to check if the `ViewHolder` is present (6). If it is, then we get the `CheckedTextItem` we need directly from it (7). If it isn't there, we know that we're not dealing with a recycled view, and we must call `findViewById` to get a reference to the `CheckedTextItem` (7), create the `ViewHolder`, and use `setTag` to stick it onto the `listItem` (8). The ViewHolder pattern can help make your `ListView` faster and more efficient, and should always be considered when you expect to have more than a few items in the list.

> **CODE WITH VIEW REUSE IN MIND** Because we don't persist any of this information, you may think that it seems contrived to go through the hassle of creating a custom adapter that only saves data in memory, and which is therefore going to be thrown away whenever the application is closed. You may also think that we could toggle the check box view whenever a user taps it and see the same effect, right? Wrong! That's because adapters are responsible for creating the view that represents the element at a certain position of the underlying data set, and all standard adapter classes such as `ArrayAdapter` (and also all well-implemented custom adapters) cache and reuse item views via `convertView` for performance reasons. Consequently, we'd see checked items reappearing across the list even though we never checked them. Again, even if that weren't the case, remember that state *should be* updated in the model, not the view.

Note that our custom `Adapter` is not *the* solution; it's *a* solution (a simple one). Another perfectly valid approach would be to create a `Movie` model class, and remember in these objects whether a movie is owned. We could then get rid of the `HashMap`

and instead get the `Movie` object at a given position and ask the object whether it's owned by the user. In fact, we'll do that in chapter 9, when we extend MyMovies to talk to an online movie database. In any case, we have to update the model managed by our new adapter whenever the user clicks a movie item, which is achieved by implementing the `ListActivity.onListItemClick` handler we saw in listing 4.7.

DISCUSSION

Using adapters this way is a powerful approach for managing dynamic data that's completely decoupled from any views, and therefore from how it's being displayed. Even though we didn't write that much code, sometimes this is too much complexity already. If you only need to ask the user to select from multiple choices in a list (a list dialog would be a good example), then there's a much easier way than implementing your own adapter: the `ListView` `choiceMode` attribute. With the choice mode set to `multipleChoice`, you can receive any list items the user selected by calling `ListView.getCheckedItemIds()`. This roughly corresponds to the map of Boolean values we maintain. You can also set the choice mode programmatically using `ListView.setChoiceMode`. This is one instance where state is only maintained in the views, without a model behind it, but it's a limited approach.

There are many different kinds of adapters, some of which you'll meet in later chapters. For now it's sufficient that you've learned how to work with them, and how you can use adapters to arrive at more flexible designs by separating data from its representation on the screen. At the same time, you've probably noticed that list views, though useful, can get quite complex. This is why we'll look at some general tips coming up, but first, we'll quickly look at how header and footer views can be added to a list.

TECHNIQUE 3 Header and footer views

List views are a great way to cope with large data sets, but the ability to scroll alone doesn't always help—your thumbs may be sore by the time you reach the bottom of a list. It'd be nice, for instance, to have a back-to-top button at the bottom of our list, so that the user doesn't have to scroll all the way back through a hundred titles to reach the top of the list again. But our list contains movie titles, which are text views, not buttons. We need a list element that looks and behaves differently from normal entries, but our adapter is only able to create one type of list element: a movie item. Looks like we're stuck.

PROBLEM

You need dedicated list elements at the top or bottom of a list, which scroll with the list content like a normal entry, but may have entirely different layout and functionality.

SOLUTION

This is what Android's list header and footer views are for. You can set them using `ListView.addHeaderView` and `ListView.addFooterView` respectively. To see how this works, we'll build a back-to-top button for MyMovies. We first define a layout (list_footer.xml) for our footer view that contains a single button, perhaps like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="@android:attr/listPreferredItemHeight"
    android:gravity="center_vertical"
    android:text="Back to top"
    android:onClick="backToTop"
/>
```

Note how we use the `onClick` attribute in our footer view layout. This tells Android to look for a public method called `backToTop` defined in our activity, which takes a single `View` object as a parameter. This is a nifty way to wire a click handler to a component from XML (without having to write the boilerplate code to explicitly assign the handler). This is going to be our callback where we do the scrolling. We then have to inflate this layout to receive a `Button` object, and set it as our list's footer view as seen in our `Activity`'s `onCreate` method:

```
public void onCreate(Bundle savedInstanceState) {
   ...
   Button backToTop =
      (Button) getLayoutInflater().inflate(R.layout.list_footer, null);
   backToTop.setCompoundDrawablesWithIntrinsicBounds(getResources()
      .getDrawable(android.R.drawable.ic_menu_upload), null, null,
        null);
   listView.addFooterView(backToTop, null, true);
   ...
}
```

No magic involved here. We add this code to the middle of the `onCreate` method we saw in listing 4.7, directly after we declare `ListView listView = getListView()`. This way, it's included when our `ListActivity` is created, and it's set *before* we declare and set the adapter. To get an idea of what this will look like, figure 4.8 shows the list with the button at the bottom in all its glory.

We've also added an icon (we reused a framework icon that has an up-arrow on it) to the button. In case you've wondered about the allowed number of header and footer views: you can add as many header or footer views as you like. They'll stack up and appear before (for header views) or after (for footer views) the list items.

Figure 4.8 We added a button that returns us to the top of the list using a list footer view. The button icon can be set using the setCompoundDrawableWithIntrinsicBounds method.

DISCUSSION

Header and footer views are useful for displaying content that's not part of the list

data, but because they're treated differently from ordinary list elements, you need to watch for some subtleties. For one, you *must* add all header and footer views before adding any data to your list (via `ListView.setAdapter`); otherwise you'll get an exception. This makes header and footer views fairly inflexible, because you can't add or remove them at will.

Moreover, even though header and footer views appear as normal list elements, remember that they're not representations of adapter data, and hence have no corresponding element in the adapter backing the list. This also means that if you want to count the list elements visible to the user, you can't rely on `Adapter.getCount` anymore. That method is oblivious to any header or footer views. Always keep in mind this asymmetry between your list view and its adapter when working with headers and footers.

We've arrived at our first MyMovies development milestone: we have an application that displays IMDB's top 100 movies, and the user can select which titles they have in their collection! List views are powerful, and now you know their ins and outs—does it feel good? In all fairness, `ListView` is a complex beast and there are many caveats regarding its use. It's likely that you'll run into one or more issues as your list item layouts grow more complex. Hence, in table 4.4 we've collected some common `ListView` related caveats and their solutions. You may be grateful for them one day!

Table 4.4 General ListView tips

ListView caveat	Solution
Don't use wrap content	Never use `wrap_content` for a list view's `height` attribute. A list is a scroll container, and by definition is infinitely large, so you should always let a `ListView` `fill_parent` (or let it otherwise expand itself, for example, using `layout_weight`).
Be careful with clickable list items	Generally, when you click a list item, the list item itself receives the click event—the view or container that's the root element of the item layout. If you place a button inside an item layout, the button steals the focus from the list item, which means while the button remains clickable, the list item itself can neither be focused nor clicked. You can mitigate this effect to at least let the entire list item be focusable again by setting `ListView.setItemsCanFocus(false)`, which will bring back the list highlight when selecting that item. But any click handling must still be performed on a per-element basis inside your item layout.
Pay attention to `getView` performance	List views can be performance killers. The `getView` method of an adapter is used to render a list item and is called frequently. You should avoid doing expensive operations like view inflations, or at least cache them. Reuse views, and consider the ViewHolder pattern we introduced earlier.

It's now time to move on to a topic that developers typically fear, but at the same time is fundamental to a successful mobile application: look and feel. We won't discuss graphic design here—that's probably not what you get your paycheck for, considering

you're reading a book on programming—but you still need to know how to set your application up to make use of design elements. Implementing custom designs on Android requires a lot of work on the programming side. Hence, the next few sections will equip you with the knowledge to implement highly customized Android user interfaces. Pimp my Android!

4.5 Applying themes and styles

Let's not beat around the bush: a typical stock Android application looks unimpressive. With Android 3.0 (aka Honeycomb) and the tablet game, things are getting significantly better, but a vanilla pre-3.0 Android installation is visually underwhelming. Fortunately, Google has given developers the necessary tools to spice up their application UIs by means of the myriad of view attributes you can define or override for your views. As you can probably imagine, this can become a tedious and repetitive task, which is why Google added a *theme engine* to Android, and we've prepared two techniques to explore it.

4.5.1 Styling applications

First, we should get a common misconception about Android themes and styles out of the way. Yes, you can deploy custom application themes for users to download and use, but first and foremost, Android themes are a tool for developers, not end users. Many applications, such as the Firefox web browser, allow users to create their own themes and share them with others. That's because Firefox's theme engine is meant to be usable by end users, so it's not exclusively accessed by the Firefox developers themselves. That's typically not true for Android themes, because theme development is tightly coupled to the development and deployment of the application itself, and there's no mechanism to change an application's theme unless such functionality has been built into the application by the developer.

What are Android themes then, and what are styles? Let's answer that last part first.

TECHNIQUE 4 **Applying and writing styles**

It should be mentioned that you could potentially get a super-nifty looking application without writing a single style definition. You *could*. But would you want to? To understand the problem, consider this view definition:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:text="Hello, text view!"
    android:textSize="14sp"
    android:textStyle="bold"
    android:textColor="#CCC"
    android:background="@android:color/transparent"
    android:padding="5dip"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
   />
```

There's a lot of noise here. We've encoded a lot of information in this view definition that affects the view's appearance. Does that belong in a layout file? Layouts are about structure, not appearance. Plus, what if all our text views are supposed to use the same font size? Do we want to redefine it for every single text view? This would mean that if we were to change it, we'd have to touch the code of *all* of the text views in our application.

PROBLEM

Customizing view attributes in your layouts, especially those that affect appearance, leads to code clutter, code duplication, and generally makes them impossible to reuse.

SOLUTION

Whenever you find yourself applying several related attributes directly to a view, consider using a *style* instead. A style is a surprisingly simple concept.

> **DEFINITION** A *style* in Android is a set of view attributes, bundled as a separate resource. Styles are by convention defined in res/values/styles.xml.

If, for instance, we create a custom style for our text views, we can define any customized attributes *once,* in a styles.xml file, like so:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="MyCustomTextView" parent="@android:style/Widget.TextView">
    <item name="android:textSize">14sp</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textColor">#CCC</item>
    <item name="android:background">@android:color/transparent</item>
    <item name="android:padding">5dip</item>
  </style>
</resources>
```

As you can see, we took all the styling attributes from our view definition and put them inside a `<style>` element. A style is defined in terms of *style items*, each of which refers to an attribute of the view the style is being applied to. Styles can also inherit from each other: in this case, we've inherited from the default `Widget.TextView` style, which is semantically equivalent to copying all attributes from the parent style to our own style. All attributes that are redefined in our style will overwrite any attributes of the same name that were defined in the parent style. Finally, styles can be applied to any view like this:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:text="Hello, text view!"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/MyCustomTextView"
/>
```

Note how we use the `style` attribute without the `android:` prefix. This is intentional: the `style` attribute is global and not defined under the `android` XML namespace. It's

important to understand here that styles are a set of view attributes bundled together. There are no type semantics: if you define attributes in a style and then apply the style to a view that doesn't have these attributes, the style will silently fail but you won't see an error. It's up to the developer to design and apply styles in a meaningful way.

DISCUSSION

Styles are meant to alleviate two common design problems. For one, placing code for defining appearance and code for defining structure into the same file isn't a good separation of concerns. Though you're building the screen layout, the styling is most likely being done by someone who does *not* write application code. Constantly messing about with the same source code files is almost like asking for merge conflicts during commits (you *do* use a source code control system, don't you?). Even if you take on both roles, a good separation of concerns helps keep your code clean, readable, and easy to maintain.

Speaking of maintenance, this brings us to design problem number two. Defining attributes for appearance directly in your view XML—or worse, in application code—makes them impossible to reuse. This means that you'll end up copying the same style attributes to other views of the same kind, resulting in code duplication and proliferation. We've said before that this is bad, because it violates the DRY principle. Like the `<merge>` and `<include>` elements, styles in Android help you keep your view code DRY. You can put shared attributes from your views into a style resource, which can then be applied to many views while being maintained from a single point of your application.

A question that remains is: what can we style, and which styles already exist so we can inherit from them? The answer is that anything defined in the `android.R.styleable` and `android.R.attr` classes can become part of a style—may be used as a value for a style item's `name` attribute. Existing styles are defined in `android.R.style`, so anything defined in that class can be used as a value for a style's `parent` attribute, or even be applied directly to a view. As usual, underscores in the `R` class attribute names translate to dot-notation in view code, so `android.R.style.Widget_TextView_SpinnerItem` becomes `android:style/Widget.TextView.SpinnerItem`. Now you know what styles in Android are, but let's move on to themes.

TECHNIQUE 5 Applying and writing themes

We've already seen how to extract common view attributes into styles, but we're still repeating ourselves—violating the DRY principle. If we define a style for text views, we still have to apply the style manually to every `TextView`. This is clearly not DRY. Maybe not wet, perhaps moist, but surely not DRY. It also opens new questions: what if we forget to apply the style to one of our views?

PROBLEM

Bundling view attributes to styles is useful, but it's only half of the solution. We still need to apply styles to all views that are targeted by the style. This should be done automatically.

SOLUTION

The complete solution to this is, you guessed it, themes. Fortunately, explaining themes is simple. Themes are styles. Yes, it's as simple as that. The only difference between a theme and a style such as the one shown in the previous technique is that themes apply to activities or the entire application (which means, all activities in an application), and not to single views. The difference therefore is one of scope, not semantics or even structure.

> **DEFINITION** A *theme* in Android is a style that applies either to a single activity or all activities (in which case it becomes the application's *global theme*).

Because themes are styles, they behave exactly the same, and are defined exactly the same: using the `<style>` tag. Because they're applied to activities, they take different style attributes than a widget style. You can identify style attributes meant for theme definitions by their `Theme_` prefix in `android.R.styleable`.

Let's proceed and apply a theme to our MyMovies app. We introduced the style concept using `TextView`, which is a good example because it takes many different attributes (we'll show you another, even better way of cutting down on `TextView` attribute bloat coming up). But our example application doesn't use many `TextView` elements, so it would seem contrived to do that. Instead, let's see if we can make our movie list look fancier. We want to add a background image to the application—something related to films would be good. It should blend with the list, letting the window background shine through. Moreover, we want to make a couple of smaller changes such as rendering a fast-scroll handle. The following listing shows how to do that using themes and styles.

Listing 4.9 The style and theme definition file for the MyMovies application

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="MyMoviesTheme"                                    Theme definition (1)
      parent="@android:style/Theme.Black">
    <item name="android:listViewStyle">@style/MyMoviesListView</item>
    <item name="android:windowBackground">@drawable/film_bg</item>
  </style>

  <style name=" MyMoviesListView"
     parent="@android:style/Widget.ListView">
   <item name="android:background">#A000</item>
   <item name="android:fastScrollEnabled">true</item>
   <item name="android:footerDividersEnabled">false</item>
  </style>
</resources>
```

(2) Style definition for list views

The theme definition shown in listing 4.9 (which uses the style element, too) applies custom styling to all `ListView` instances in the application and sets a custom window background (1). The custom `ListView` style that's being applied defines the attributes that all list views in this application will now share (2).

We've defined a theme for our application, but we haven't yet applied it to anything. Recall that themes may be applied to single activities or the entire application (all activities), so we need to tell Android what we want to style. Themes are applied in the manifest file using the `android:theme` attribute. If we were to apply it to a single activity, we'd set that attribute on an activity element; otherwise we set it on the single application element, as follows:

```
<application android:theme="@style/MyMoviesTheme" ...>
    ...
</application>
```

Figure 4.9 shows how MyMovies looks now that we've slapped some styling onto it.

You can see how the window background is visible through the semitransparent list view background. You can also see the fast-scroll handle at the right side of the list. You can grab it to scroll quickly, and it'll fade away and get out of your way if the scroll has ended.

Figure 4.9 The MyMovies title screen after some styling has been applied. Note how we included a transparent background image and added a fast scrolling handle.

DISCUSSION

As you can define view attributes both in XML and in program code, you can also apply themes to an activity programmatically using `Activity.setTheme`. Nevertheless, this way of doing it is discouraged. In general, if something can be done in the XML, it's good practice to do it in the XML, and not in your application code. Calling `setTheme` will also only work *before* you inflate the activity's layout; otherwise the theme's styles won't be applied. This also means that you can't change a theme on-the-fly (such as through the click of a button), because you must restart the activity for it to have an effect.

That covers the basics of defining and using styles and themes. Still, a few things worth knowing about remain. Remember how we mentioned that `ListView` is a complex beast and that we'll come back to it? Here we are. Styling list views has a nasty pitfall that almost all developers new to Android step into, so let's get it out of the way once and for all.

TECHNIQUE 6 Styling ListView backgrounds

`ListView` is a complex widget, and sometimes this complexity gets in your way when trying to change its appearance. In fact, we weren't completely honest with you when we showed you the code for the list view style in listing 4.9. It's lacking a setting that

will allow the style to be rendered correctly. If you try to apply a custom background, or as in our example let the background of the window or another underlying view shine through, then you may observe visual artifacts such as flickering colors when performing scrolls or clicks on the list. On a related note, if you try to set its background to a transparent color, expecting to see the widget rendered underneath the list, you'll find that Android still renders the default black background.

PROBLEM

You apply a custom background (color or image) to a `ListView`, but you don't get the desired effect or get visual artifacts when rendering the list.

SOLUTION

These problems can be attributed to a rendering optimization `ListView` performs at runtime. Because it uses fading edges (transparency) to indicate that its contents are scrollable, and this blending of colors is expensive to compute, a list view uses a color hint (by default the current theme's window background color) to produce a prerendered gradient that mimics the effect. The majority of the time a list view is rendered using the default color schemes, and in those cases this optimization is effective. Yet, you can run into the aforementioned anomalies when using custom color values for backgrounds.

To fix this problem, you need to tell Android which color it should use as the hint color using the `android:cacheColorHint` attribute. This is done as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  ...
  <style name="MyMoviesListView" parent="@android:style/Widget.ListView">
    <item name="android:background">#A000</item>
    <item name="android:cacheColorHint">#0000</item>
    ...
  </style>
</resources>
```

Setting the `cacheColorHint` properly (or disabling it by setting it to transparent) will fix any obscure problems you may encounter when working with a list with a custom background color.

DISCUSSION

Regarding the value for the color hint, if you use a solid list background color, set it to the same color value as the background. If you want the window background to shine through, or use a custom graphic, you must disable the cache color hint entirely by setting its value to transparent (#0000 or `android:color/transparent`). And that's that. This is a rather obscure issue, but you should keep it in mind when working with `ListView` implementations that need custom backgrounds.

Understanding the `cacheColorHint` wraps up our discussion of styles and themes, almost. We've collected one more set of tidbits about styles in Android, which we want to show you before moving to the next topic.

4.5.2 Useful styling tidbits

Ready to get even deeper into styling applications? You now know how to create and apply themes and styles, but we have some extra useful tips and tricks to round out this discussion. The paragraphs that follow cover things that haven't been mentioned yet, in no particular order, but all of which we think make your life easier when working with styles. In particular, we'll demystify color values, tell you how to work with text appearances, and also introduce some rarely seen but useful style notations.

COLOR VALUES

When working with styles, you'll often find yourself specifying color values, either directly using hexadecimal syntax or by referencing a color resource—yes, colors can be resources, too! Any color value in XML is defined using hex notation and identified by the # prefix, so let's briefly cover it now. Color values in Android are defined using the *Alpha/Red/Green/Blue* color space (ARGB), where each color is mapped to a 32-bit wide number, with the first 8 bits defining the alpha channel (the color's opacity), and the remaining 24 bits representing the three color components, with 8 bits for red, green, and blue each. Because each component may use 8 bits of information, the value range for each component is 0 to 255, or 00 to FF in the hexadecimal system. A color value of #800000FF would therefore represent blue with 50% opacity.

> **COLOR VALUE SHORTCUTS** In cases where each color channel is represented by two identical hex digits, you're allowed to use an abbreviated hex string where every hex digit pair is collapsed into a single digit. For example, the colors #FFFFFFFF and #AABBCCDD can be abbreviated to #FFFF and #ABCD, respectively. Moreover, you can always omit the alpha channel, in which case full opacity is assumed, so #FFFF can be abbreviated even further to #FFF.

Typing out these color values can get tedious. For one, you're repeating yourself, which as we've learned is a bad thing. Worse maybe, these values aren't intuitive unless you're good at mentally mapping hex values to a color space. Hence, you'll typically define these values only once, as a color resource, which you can address using a human-readable identifier. To do that, create a file named colors.xml in your `res/values` folder, and add the following code to it:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="translucent_blue">#800000FF</color>
</resources>
```

You can now reference this color from your views and styles as `@color/translucent_blue`. Note that Android already defines a handful of colors for you in this way. A commonly used predefined color is `android:color/transparent`, which is equivalent to a color value of #00000000. When in application code, you can also use the color definitions from the `Color` class, but you can't reference these from XML.

One last thing about colors. Colors defined as shown here can be used as *drawables* in Android. We haven't covered drawables yet (though we touched on them in chapter 2), but for now, keep in mind that you can also assign color values to attributes that expect drawables, for example, for backgrounds or list selectors. Colors as drawables can be powerful, and we'll show you how as part of section 4.6.

TEXT APPEARANCE

We mentioned before that when working with text styles, there's a better way than defining your text styles from scratch: Android's text appearances. A *text appearance* is a style that contains elements that apply to any `TextView` in your app. Moreover, overriding these default styles will immediately affect all text styles in your application. For example, if you were to change the default text style to bold red across your entire application, you could do this:

```
<style name="MyTheme">
   <item name="android:textAppearance">@style/MyTextAppearance</item>
</style>

<style name="MyTextAppearance">
   <item name="android:textColor">#F00</item>
   <item name="android:textStyle">bold</item>
</style>
```

Plenty of text appearance attributes are available for themes, such as `textAppearance` for default text, `textAppearanceButton` for button captions and `textAppearanceInverse` for inverted text (such as highlighted text). But you'll probably ask yourself, what's the difference between this and defining a default style for `TextView`, as seen before? The differences are subtle but important. First, the styling defined here will actually be applied to all `TextView` views, including subclasses. This isn't the case for the `textViewStyle` attribute—it won't affect text views such as `Button` or `EditText`, which both inherit from `TextView`. Second, the `textAppearance` attribute can be applied to a theme and a single `TextView` (and hence, also to a `TextView` style definition). This allows you to bundle shared text styles together and apply them en masse to different kinds of text views—think another layer of styling DRYness for your code.

By any means, if you start styling text in your application you'll want to do this using text appearances. Let them inherit from Android's default text appearance styles, and overwrite only what you need to change.

SPECIAL STYLE VALUES

You've seen how to reuse styling attributes by bundling them together into styles and by letting styles inherit from each other. You've also seen the `@` notation that you use to reference existing resources. This works well if you want to address the complete resource, but what if you need only a single value? Consider a text style. For example, suppose you want to change the link color to the color Android uses for plain text, but you don't know that color's value. Furthermore, the primary text color isn't exposed as a color resource because it's variable; it changes with the theme.

The solution is to use the ? notation, which like @, only works in XML. You can use it to address style items of the *currently applied theme* by their name. To stick with the example, if you want to set your application's link color to the default text color, you could do this:

```
<style name="MyTheme">
  <item name="android:textColorLink">?android:attr/textColorPrimary</item>
</style>
```

The last thing we'd like to mention is the `@null` value. You can use it whenever you want to remove a value that's set by default (in a parent style). This is probably seldom required, but it makes sense if you want to get rid of a drawable that's set by default. For instance, Android will set the `windowBackground` attribute to a default value, but if the window background is always obscured by your own views, you can remove it by setting it to `@null`. This will result in a slight performance boost, because Android currently can't optimize views that are completely obscured away from the rendering chain, although this limitation may change in future versions.

Styles are a complex topic, and there's often confusion about the distinction between themes and styles. Hopefully we've solved most of these mysteries. We started by showing how styles are defined and how they're applied to views. We then showed you how you can assign styles globally to your application using themes, and even sorted out some confusion with background styling in list views. As always, we suggest you play around with view styling yourself. That's the best way to get your head around a problem. Now, let's move forward and learn about another important concept of Android's UI framework, drawables.

4.6 Working with drawables

To be frank, we've dodged the concept of drawables up to this point and tried shamelessly to sweep it under the rug. It's difficult to discuss all the user interface topics without touching on drawables. But we can't fool you, can we? You've already seen several occurrences of drawables: images (bitmaps) and colors. So what exactly is a drawable?

> **DEFINITION** A *drawable* in Android, defined by the `Drawable` class, is a graphical element displayed on the screen, but unlike a widget is typically noninteractive (apart from picture drawables, which actually allow you to record images by drawing onto a surface).

Apart from images and colors, there are drawables for custom shapes and lines, drawables that change based on their state, and drawables that are animated.

Drawables are worth covering separately, because they're powerful and ubiquitous in Android. You need them practically everywhere: as backgrounds, widget faces, custom surfaces, and generally anything involving 2D graphics. We won't cover every kind of drawable here, only the most widely used ones. Additionally we'll come back to drawables in chapter 12, where we'll discuss 2D/3D graphics rendering. For now, we'll focus on drawables that are important for styling your applications.

4.6.1 Drawable anatomy

Drawables always live in the `res/drawables` folder and its various configuration-specific variants and come in two different formats: binary image files and XML. If you want to use a custom image file in your app, it's as easy as dropping it into that folder. The ADT will discover the drawable and generate an ID for it that's accessible in Java through `R.drawable.the_file_name`, (like with any other Android resource). The same is true for XML drawables, but you'll have to write these first, and we'll show you in a minute how to do that.

If you've placed a drawable in the drawables folder, you can access it from an `Activity` by a call to `getResources().getDrawable(id)`. But it's more likely that you'll use drawables in your style or view definitions, for use as backgrounds or other graphical parts of a widget, and sometimes it's difficult to identify them as such. Let's recall our list style definition from listing 4.9:

```
<style name="MyMoviesListView" parent="@android:style/Widget.ListView">
  <item name="android:background">#A000</item>
  <item name="android:fastScrollEnabled">true</item>
  <item name="android:footerDividersEnabled">false</item>
</style>
```

We're already using a drawable here. Can you see it? To be fair, it's not jumping out. It's the color we used as the list's background. For some attributes, Android allows color values where it usually expects a normal drawable, such as a bitmap image. In that case, it'll turn the color into a `ColorDrawable` internally. Note that for some reason this doesn't work everywhere. For instance, the `android:windowBackground` attribute doesn't accept color values.

We can also define special drawables entirely in XML. When doing so, the root element is typically the drawable's class name with the *Drawable* part stripped off. For instance, to define a `ColorDrawable` you'd use the `<color>` tag, as we showed you in the previous section (we'll see two exceptions to this rule in a moment, admittedly making this a little confusing). Note that not all kinds of drawables can be used everywhere; list selectors for instance don't accept a plain color value because a selector has more than one state and one color drawable isn't enough to reflect that. Curiously, a plain image would work here, though.

Creating custom drawables is unfortunately not one of the most well-documented parts of the platform SDK. We'll cover three types of custom drawables, each of which is useful for styling your apps: shape drawables, selector drawables, and nine-patch drawables.

TECHNIQUE 7 Working with shape drawables

Sometimes, static images such as PNGs or JPEGs aren't flexible; they don't scale well if the area they're supposed to cover can change in size. Two examples immediately come to mind. First example is gradients. The nature of gradients is to start with one color on one end and have it blend into another color. If you define this as a static

image, then stretching or squeezing the image will result in rendering problems. Another good example is dashed outlines and borders. If you define a dashed border as a background image that has the border painted on it, then the dash length will stretch along with the view you apply it to. But what if you want the dash length and dash gaps to remain static while still being able to arbitrarily resize the view? Clearly, in both cases you'd be better off if those images were generated at runtime.

PROBLEM

You want to render graphics that are difficult to scale, such as gradients or patterns, or graphics that are easier to manipulate at runtime.

SOLUTION

If you find yourself in either situation, you may want to use a *shape drawable*. A shape drawable is a declaratively defined and dynamically rendered graphical element that you can use in any place where drawables are allowed, for instance for view backgrounds.

Shape drawables are represented internally by the `GradientDrawable` and `ShapeDrawable` classes, depending on which kind you use. In XML, they're always defined using the `<shape>` element. We've already seen how we can change the selector drawable for our list view to a predefined drawable, but this time, let's create our own, without any help from imaging applications. A gradient list selector sounds like a cool idea for MyMovies, so let's do that. In `res/drawable`, create a new file (let's call it list_selector.xml) that contains a new selector as a shape drawable:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">
  <gradient android:startColor="#AFFF"
            android:endColor="#FFFF"
            android:angle="0"/>
  <stroke android:color="#CCC" android:width="1px" />
  <corners android:radius="5px" />
</shape>
```

We set the drawable's shape to `rectangle` because that's what we need. We could've also omitted it because that's the default shape. The subelements of the shape element define its features. For our purpose, we set a custom background gradient (`<gradient>`), a border line (`<stroke>`), and a border radius (`<corners>`). Now let's apply it to our list view style:

```
  <style name="MyMoviesListView" parent="@android:style/Widget.ListView">
    ...
    <item name="android:listSelector">@drawable/list_selector</item>
  </style>
```

As you can see, the `list_selector` shape is applied like any other drawable. We've replaced the former value for the list selector drawable with a reference to our new gradient shape. Figure 4.10 shows how a selected list element now looks when booting up MyMovies with that change applied.

Figure 4.10 The list view selector using a custom shape drawable. Note how we used a custom gradient that goes across the selector horizontally.

Like what you see? We've only just started. There are many more options to create shape drawables, as we'll see in a moment.

Gotchas with list selectors

List views use the `android:listSelector` attribute to determine which color or image to use as the list selector. Android allows two different ways of rendering this selector: behind a list element's layout, or in front of it, as specified by `android:drawListSelectorOnTop`. Each approach has its advantages and disadvantages, of which you should be aware. The default is to draw selectors behind a list element: this requires that all views in a list element's layout have a transparent background (which is the case for most Android views unless you change it). Otherwise, they'd obscure the selector. If you render images such as photos as part of a list element, you have no choice: they'll obscure it, because a photo is always solid. This means that when using images or solid backgrounds in your list elements, you probably want to draw the selector on top. Therefore the selector must be translucent; otherwise it would itself obscure all views of a list item. Keep this in mind when designing custom list selectors.

DISCUSSION

Shape drawables are a great way of arriving at neat-looking visual elements without having to mess about with image-manipulation programs. They're customizable via the source code, and you as the developer, have full control. Did I hear the design team scream? Yes, it's their job to create nice-looking graphics, but if you want to add an outline to a widget or you need more flexibility, use the color palette assigned by the designers. That's teamwork!

We mentioned that you can create more than boxes and borders. Shape drawables can take various sizes and shapes, from rectangles and ovals to lines and rings. Table 4.5 summarizes most of the elements you can use to define shape drawables (we've omitted some of the more obscure attributes for brevity).

Table 4.5 Valid elements for defining shape drawables

Element name	Description	Attributes
`<shape>`	The root element.	`android:shape`—the type of shape, `rectangle`, `oval`, `line`, `ring`

`<gradient>`	Defines the shape's gradient background.	`android:type`–the gradient type, `linear`, `radial`, `sweep`
		`android:startColor`—the start color of the gradient
		`android:centerColor`—an optional third center color for the gradient
		`android:endColor`—the end color of the gradient
		`android:angle`—the gradient angle, if type is `linear`
		`android:centerX/android:centerY`—the center color position, if one is set
		`android:gradientRadius`—the gradient's radius if it's either `radial` or `sweep`.

`<solid>`	Gives the shape a solid background.	`android:color`—the background color
`<stroke>`	Defines the border/ outline of the shape.	`android:color`—the border color `android:width`—the border width `android:dashGap`—the gap width if you want the line to be dashed `android:dashWidth`—the dash width if you want the line to be dashed
`<corners>`	Defines the corner radius of the shape.	`android:radius`—the radius for all four corners `android:topLeftRadius`, `android:topRightRadius`, `android:bottomLeftRadius`, `android:bottomRightRadius`—the radius of each individual corner
`<padding>`	Defines the padding for this shape.	`android:top`, `android:bottom`, `android:left`, `android:right`—the padding for each side of the shape
`<size>`	Defines the size of the shape.	`android:width`, `android:height`—width and height of this shape

You can find the full reference at http://mng.bz/v0Rg

At this point, we've created a nice-looking list selector graphic, but there's a problem with it. It always looks the same. When interacting with widgets you'd normally expect some visual feedback as soon as you click or select it. But how would that work? We only have a single `listSelector` attribute, which takes exactly one value, but we'd need at least two in order to use different graphics. The answer is that no, you don't. What you need instead is a *selector drawable*.

TECHNIQUE 8 Working with selector drawables

Sometimes, you need to display graphical elements that change along with a view's state. A good example is a button in Android. If you select one with the D-pad or trackball it receives focus and a light orange overlay is rendered. When you press it, the overlay changes its color to a darker orange, and a long press again uses a different effect. Because we can only assign one drawable at a time to be used for a background or highlight, we clearly need some sort of stateful drawable.

PROBLEM

A view exposes an attribute that takes a drawable, but you want that drawable to change with the view's state.

SOLUTION

Stateful drawables in Android are called selector drawables, and are declared using the `<selector>` element. This special kind of drawable can be thought of as a drawable switcher, if you will. Depending on which state a view is in (selected, pressed, focused, and so on), this drawable replaces one of its managed drawables with another. A true shape shifter!

Coming back to our application, we want to apply this to our list selector. Instead of always showing the same gradient, we want the gradient to change its start color from grey to a light blue whenever a list item is pressed. Because we now have two different list selectors—one for default state one for pressed state—we need to keep them in separate files (let's call them list_item_default.xml and list_item_pressed.xml). Here's a snippet for the new `list_item_pressed` drawable:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
  android:shape="rectangle">
  <gradient android:startColor="#AA66CCFF"
            android:endColor="#FFFF"
            android:angle="0"/>
  <stroke android:color="#CCC" android:width="1px" />
  <corners android:radius="5px" />
</shape>
```

Nothing terribly new here; we've replaced the gradient's start color with a different one. Now that we have two drawables, we need to bring them together in a selector drawable. For that, we modify list_selector.xml from the previous technique to something like this:

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

What we've done here is replace the root element of the list selector from a shape to a shape switcher—a selector. Selector drawables are defined in terms of `<item>` elements,

Figure 4.11 The new list selector in the default (left) and pressed (right) states. Note how it changes colors when in the pressed state.

each of which takes two arguments: a state, and a drawable to be displayed whenever the view to which this selector is being applied enters that state (you can also use color values as we'll see in a minute). Figure 4.11 shows the selector in both states.

You can switch all sorts of drawables using the `<selector>` tag, not just shape drawables. The most common examples are nine-patch images, which we'll cover next. In fact, Android uses this combination of selectors and nine-patches all over the place to render its system UI.

DISCUSSION

Our example was simple in that it only used two different states: pressed and not pressed, but there are plenty more states, each representing a Boolean condition. We've summarized them for you in table 4.6 (again, we only list the more commonly used ones).

Table 4.6 Common selector drawable states

State	Description
`state_focused`	The view has received the focus.
`state_window_focused`	The view's window has received the focus.
`state_enabled`	The view is enabled/activated.
`state_pressed`	The view has been pressed/clicked.

Table 4.6 Common selector drawable states *(continued)*

State	Description
`state_checkable`	The view can be checked/ticked (not supported by all views).
`state_checked`	The view has been checked/ticked (not supported by all views).
`state_selected`	The view has been selected (not supported by all views).

For a more exhaustive list of states, refer to http://mng.bz/Math and http://mng.bz/qzXz.

One thing to watch for is the order of your selector's state items. To find a drawable in the selector that matched a view's current state, Android walks through the list of items from top to bottom—in the order they're declared in the selector—and selects the first one that satisfies the view's current state. Why is this important? Imagine a focused and checked `CheckBox` view that's skinned with the following selector:

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

You may expect that Android would pick the second drawable, because it clearly declares that it's the one that should be used whenever the `CheckBox` is checked, right? Wrong! That's because the first item is less restrictive, and also matches the view's state: it only requires the focused state, which is true, and doesn't require any specific checked state. Because it's the first match Android finds, Android will use this for the `CheckBox` whenever it receives focus, regardless whether it's checked or not. The second item will therefore *never* match. What does this tell us? It tells us: always make sure that the least-restrictive state items are the *last* items in the state list. Otherwise they'll obscure more specific state items.

Here's another useful hint. We mentioned before that you may use color values in selectors using the `android:color` attribute. This is particularly useful when working with stateful text appearances, where you want the text color or size to change. For example, if `TextView` receives focus, you can assign a selector that switches colors with states to the `android:color` attribute of a `TextView`! It's in these details where it becomes apparent how flexible and awesome Android's view system is.

As with shape drawables, selector drawables are mapped to Java framework classes. When using plain drawables, a selector will become a `StateListDrawable`, and a `ColorStateList`, which curiously is *not* a `Drawable`, when using colors. Hence, when using selectors that switch color values, remember that you can't use them in places where drawables are expected, only for color values (but then again we learned earlier that you can turn colors into drawables easily, so this problem can be circumvented).

We're getting close—we have one kind of drawable left up our sleeves. We've mentioned nine-patch drawables before, and they're perhaps the most useful and commonly used drawables, so read carefully!

TECHNIQUE 9 Scaling views with nine-patch drawables

Nine-patch drawables are best explained by example. How about this; in our application, we'd like to add some kind of title image above the movie list view that says "IMDB Top 100". We can do this by changing the MyMovies main screen as seen in the following listing.

Listing 4.10 Extending the MyMovies layout to include a title image

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

    <ImageView android:src="@drawable/title"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:scaleType="fitXY" />

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

1 Add ImageView and set width to fill_parent

2 Scale bitmap to fit ImageView

In listing 4.10 we've added an `ImageView` that displays our title image (stored as /res/drawable/title.9.png) 1. The title image is a PNG image file, not an XML-based drawable. Note how we tell it to `fill_parent` across the horizontal axis. If the device is in portrait mode, this has no effect (at least on a standard screen size, which we assume for this example), because our image happens to be exactly 320 pixels wide. But if we turn the device to landscape mode, we want the `ImageView` to remain stretched across the whole screen width. Because the layout parameter only affects the `ImageView` (the widget), but not the image itself (the bitmap), we also set the `scaleType` to `fitXY`, which means that the bitmap should be resized to fill its `ImageView` container in width and height 2. The result is shown in figure 4.12 for both portrait and landscape mode.

Figure 4.12 The MyMovies header image shown in portrait (left) and landscape (right) modes. In portrait mode, the image won't stretch; in landscape mode it'll stretch to fill the view horizontally, resulting in distorted proportions.

Note how the text of the title image gets blurry and loses proportion in landscape mode. The white dots turned into eggs! That's because Android stretches the image to fill the screen in landscape mode, interpolating pixels until the

image is the same size as its container. The result looks horrible, so what can we do to avoid this problem?

PROBLEM

You want to display a static image on a view that's variable in width or height, but stretching the image results in a loss of quality.

SOLUTION

Take an educated guess... correct, nine-patch drawables are the solution. A nine-patch drawable is a PNG image that defines stretchable areas as part of its image data (the bitmap), and can be rendered across arbitrary-sized views without any noticeable loss in quality. Nine-patch PNGs must end in `*.9.png`. Without this convention, Android won't recognize the PNG as a nine-patch image. Turning a conventional image into a nine-patch image is simple:

- First, take the image you'd like to use and add a one-pixel-wide transparent border (in fact, any color will work except black).
- Second, define the stretchable areas of your image by marking the top and left edges of that border at the respective sections using a solid black stroke. The stretchable area is defined by the box these demarcations form if you extended them with imaginary lines until they intersect.
- Optionally, you may repeat this step for the right and bottom edges to pad the image, whereby any areas *not* marked with black will be used as padding.

Figure 4.13 illustrates the process of defining stretchable areas and adding padding.

The top image in figure 4.13 shows how the stretchable areas of the PNG are defined. Here, the center box indicates the area that it'll be interpolated in order to resize the image (you're allowed to have several of these boxes, not just one). In the lower image, the center box defines the image's content area. Anything else is padding; if the image's

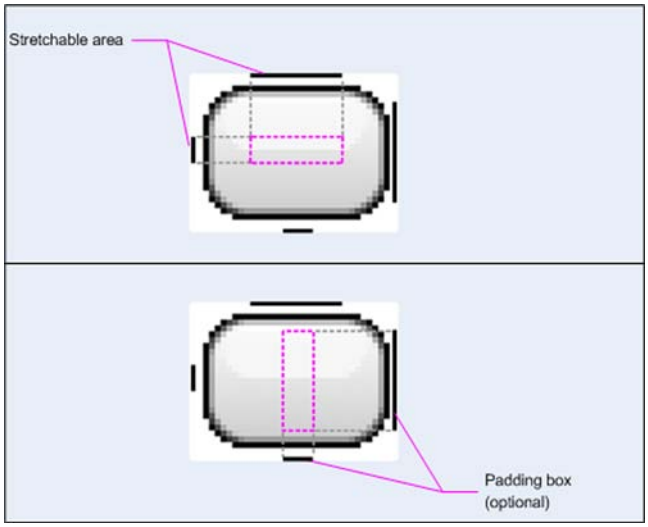

Figure 4.13
Defining stretchable areas (top) and padding boxes (bottom) in nine-patch drawables using the draw9patch tool. Whereas the stretchable area is defined using the top-left strokes, the padding area is defined using the bottom-right strokes (source: http://developer.android.com).

content outgrows this area, the image will be resized accordingly by duplicating the pixels within the content box.

PADDING BOX GOTCHA Be careful if you're not explicitly defining a padding using the padding box. It is indeed optional, but if you omit it, Android will assume the padding box to be identical to the stretchable area (it copies it), which can have awkward and unexpected side effects when the image is rendered.

For our title image, we only want the small areas between the rounded corners and the text to be stretched, because they're all of the same color, and hence can be interpolated without any loss of visual quality. Figure 4.14 shows our title image again, now redefined as a nine-patch and viewed using the `draw9patch` SDK tool. Note that the thick lines going straight across the borders are merely visual guides added by that tool and are *not* part of the image itself.

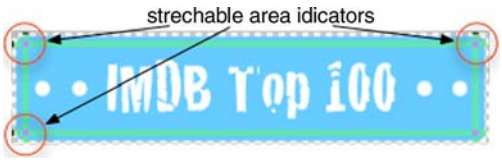

Figure 4.14 MyMovies title image defined as a nine-patch drawable; the corners demarcated by the black lines will be used for scaling (no padding has been defined).

We save this file to res/drawable/title.9.png and restart the application. Looking at the title image again in figure 4.15, in both portrait and landscape modes, shows that we've fixed the problem. It scales!

Figure 4.15 The title image now scales correctly in landscape mode. This is achieved by stretching only those parts of the image that are safe to stretch (they don't result in distorted imagery), as defined by the nine-patch format.

We're pretty good at fixing things, aren't we? Wait, what? You noticed the skewed background image, right? We didn't fix that, but *you* can do it, now that you know how nine-patch drawables work.

DISCUSSION

Nine-patch drawables are incredibly useful, and they're ubiquitous in Android itself. All standard widgets that come with the platform use them. They're particularly useful for widgets such as buttons and text fields, which frequently have to scale with their containing layout. If you want to restyle all standard widgets in your application, a good approach is to take the existing nine-patch drawables that are part of the Android open source project and do your modifications around them. A common example is changing the color of highlights to match the palette of an application's brand—the standard orange doesn't cut it sometimes.

Nine-patch drawables can be created using any kind of imaging software, but thankfully, for everyone without a license for fancy commercial image editing software, Google has added the aforementioned `draw9patch` tool to the SDK. The `draw9patch`

tool helps you by automatically adding the one-pixel border to an existing image, rendering visual guides such as marking the resulting scaling boxes, and by computing live previews of your image scaled in all directions with the current modifications.

This section has been all about the visuals. You've seen how to organize your application's view attributes in styles and themes, as well as what drawables are and how to use them to create completely customized, beautiful user interfaces—well, if your design skills are as good as your programming skills. We've talked about scaling images (on a small scale). But what about scaling the entire user interface? Android devices come in various screen sizes, and even the most beautiful user interface will fall apart if it doesn't render correctly on all devices. Therefore, the next section is all about making your application's UI scale with the various kinds of displays and configurations that are available now, and even those that aren't available yet!

4.7 Creating portable user interfaces

When talking about portability, we can mean different things: portability with respect to software (not all SDK functions are available on all handsets) or hardware capabilities (not every Android device has a light sensor or hardware keyboard). In this section, we'll talk about portability and scalability with respect to user interfaces and screen sizes.

We started developing for the Android platform in its early Alpha days, when there wasn't a single device that would run the platform... unless you count the Nokia Internet tablets that a handful of adventurous developers flashed with prerelease versions of Android. Then came Google's G1, aka HTC Dream, and life was good—you only had one device configuration to care about. Today, there are so many different devices that we've stopped counting, and Android still grows rapidly with more manufacturers jumping on the train.

Having to support many devices, as mentioned in chapter 1, is a common criticism leveled at Android. Fortunately, the Android platform introduced support for different device configurations in a graceful way, making it almost trivial to make an application work with screens sizes that weren't around when the application was developed.

Against this backdrop, the following three techniques show you how to enable your application to gracefully scale with different screen configurations, from a simple just-works approach meant for legacy applications to more elaborate, native support approaches.

TECHNIQUE 10 Automatically scaling to different screens

All the screenshots of our example application you've seen so far have been taken from the emulator, running with the default screen configuration, the one that was standard in the pre-1.6 days. With Android 1.6 came support for new configurations, and with that support, devices using these configurations such as the HTC Tattoo, which had a QVGA screen that was shorter in height than the previous ones, as seen in figure 4.16.

Figure 4.16 Different Android devices may come with different display configurations. Whereas the HTC Magic (left) comes with a 320x480px (160dpi) 3.4 inch screen, the HTC Tattoo has a smaller, lower-resolution 240x 320px (120dpi) 2.8 inch screen.

The question is: if you've developed an application on Android 1.5 or earlier, and that application has already been released on the market, how do you make sure the user interface is displayed correctly on all these different devices? New devices may not only have lower or higher resolutions (fewer or more physical pixels), their displays may also have different pixel densities (fewer or more pixels spread across the same space). The latter problem may lead to rendering issues on these devices—if a UI element had been defined as being 100 pixels wide, then on a display with a higher density, that element would appear shrunken, because it occupies less physical space. This problem is illustrated in figure 4.17 using two speculative screen pixel densities of 3 and 6 dots per inch.

Android 1.6+ implements a set of algorithms that can automatically mitigate these problems. In a moment, we'll learn how to proactively solve these problems, but let's

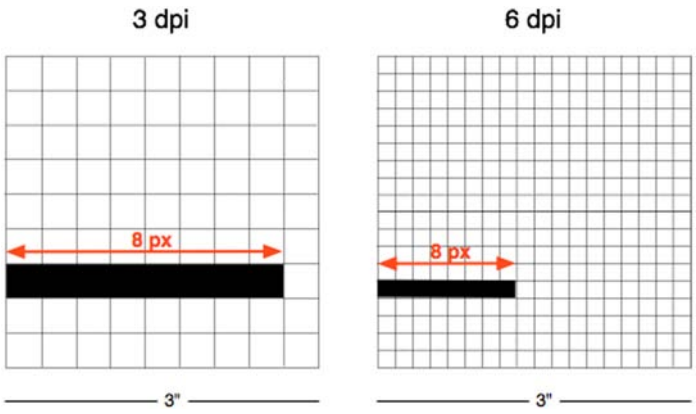

Figure 4.17 A line that's 8 pixels long and 1 pixel high on a 3 dpi screen will be only half the height and length on a 6 dpi screen because a pixel occupies less physical space. Note that this example simplifies dots to pixels, which isn't necessarily true.

assume for now that we're lazy and don't feel like fixing our application manually, and instead let the Android runtime take care of it somehow.

PROBLEM

Your application was developed with a specific class of displays in mind (specific size and pixel density), and you now want to target other screen configurations without having to change your view code.

SOLUTION

You can report the screen sizes and densities your application supports by means of the `<supports-screens>` element, which was introduced with Android 1.6. Because we haven't developed MyMovies with screen configurations that are different from the default in mind, we should tell Android about this so that it's aware of this fact. You set the configurations you support in the application manifest, as seen here:

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

    <uses-sdk android:minSdkVersion="4" />

    <supports-screens
        android:smallScreens="false"
        android:normalScreens="true"
        android:largeScreens="false"
        android:xlargeScreens="false"
        android:anyDensity="false"
    />

    ...
</manifest>
```

We tell Android that we only support devices that fall into the normal screens class. Note that this doesn't necessarily imply that our application isn't installable anymore on other devices. It doesn't even mean that it's doomed to break on them, but it has other implications as we'll see in a moment.

> **WARNING** The configuration from listing 4.21 is the one you'd automatically get when building for Android 1.5 or earlier (as indicated by the `<uses-sdk>` element). Hence, the default assumption for applications that run on Android 1.6 or newer, but which were built with an earlier SDK, is that they were developed with only the normal screen size and pixel density in mind because those were standard at that time. This is a sensible default for these applications. This is *not* the case if you're targeting an API level of 4 or higher (Android 1.6+). If you set the `minSdkVersion` to at least 4, all `<supports-screens>` attributes will default to `true` instead, meaning that if you want to remain in legacy-mode for one or more screen sizes, then you'll have to set the respective values to `false` explicitly.

What does it mean when we say normal screen? We didn't mention actual sizes in pixels or densities in dpi. That's because Android collapses all kinds of available displays into a 4x4 configuration matrix as noted in table 4.7. This matrix is organized around

a central baseline configuration, which was the sole available configuration before Android 1.6 came out, and which was used by all 1.5 devices such as the G1.

Table 4.7 Screen configuration matrix with example configurations

	Low density	Medium density	High density	Extra high density*
Small screen	Sony Xperia Mini (QVGA 240x320, 2.55")			
Normal screen		baseline configuration Google G1, HTC Magic (HVGA 320x480, 3.2")	Google Nexus One (WVGA 480x800, 3.7")	
Large screen			HTC Desire HD2 (qHD 540x960, 4.3")	
Extra large screen*		Motorola Xoom tablet (WXGA 1280x800, 10.1")		

More detailed coverage at http://mng.bz/InPC
*The xlargeScreens and hxdpi configurations were added in Android 2.3 and Android 2.2, respectively.

If your application was developed for the baseline configuration (320x480, 160dpi), Android ensures that it'll continue to work on a WVGA device by entering a fallback mode. This doesn't work with all configurations though. Understanding which fallbacks Android uses for which configurations is important.

DISCUSSION

Whenever you specify `false` for any of the previously mentioned attributes, Android will enter a fallback mode for the respective screen configurations. What happens in fallback mode is different for every attribute. If `smallScreens` is set to `false`, users with a device classified as having a small screen won't see your application on the Android market anymore (although they'd still be able to install it manually). That's because it's likely that the application's user interface will break when there's suddenly less room to render it. Keep this in mind, or you may lose a significant portion of potential users because they can't even find your application in Android Market!

It's an entirely different story with large-screen devices such as tablets, because they have enough display space to render your application in its entirety. More precisely, if `largeScreens` is set to `false`, Android will render your application in *letterbox mode*, which means it'll render it using the baseline size and density and fill any unused screen space with a black background. Not beautiful, but at least functional.

This leaves the `anyDensity` flag. Here, things get more elaborate. If set to `false`, Android will enter a compatibility mode which takes care of scaling all values specified in `px` (absolute pixels) against the baseline density of 160 dpi in order to translate them to the device's screen density. If the density is higher, these values are scaled upward; if smaller, downward. This is done to ensure that any coordinates or dimensions specified

in pixels will result in approximately the same physical positions and sizes regardless of the device's screen density (recall from figure 4.17 that measurements of screen elements defined in absolute pixels would normally have different outcomes on displays with different pixel densities).

Example: Android's auto-scaling mode

Imagine you want to display a 100px-wide image. Using the baseline configuration of 320x480 and 160 dpi, one physical pixel is 0.00625 inches wide (1/160), so this image would be 0.625 inches in width on a device using that configuration. If you now run this image on a device with a high-res 480x800 240 dpi screen, the same image would suddenly only be about 0.417 inches wide because with a higher pixel density, a single pixel takes less physical space on that screen. To counter this effect, Android multiplies the original value specified in pixels by 1.5 (240/160) which is also 0.625, and voilà, the image specified as 100 pixels wide uses the same space on both screens!

Moreover, because a density of 160 dpi is assumed, but the high-res display has significantly more pixels at a higher density, Android must report a similarly scaled-down screen size to the application, or the screen would appear to be larger, with more pixels spread across more room. Therefore, Android also downscales the screen size of the device by 0.75 (160/240) and reports a screen size of 320x533 pixels to the application, which would fall into the normal screens class.

Who said that lying can't work out well sometimes?

In addition to these measures, Android will also automatically scale all drawables it loads from the standard drawables folder, because these are assumed to have been created with the baseline configuration in mind. For instance, a 100-pixel-wide PNG image will now always take up the same room on a screen, scaling up or down depending on the size and density of the current screen (using the same logic we just discussed). This is called *prescaling* and is done when the resource in question is loaded. Scaling bitmaps comes at a cost, and we'll show you how to avoid these costly computations shortly.

To summarize, if you report in your manifest file that you don't support any but the baseline configuration, table 4.8 shows what happens.

Table 4.8 A synopsis of supports-screens settings and effects

Attribute set to false	Effect
`smallScreens`	On devices with small screens, your application will be filtered from Android Market. It can still be installed manually, and the same scaling logic discussed earlier will apply.
`normalScreens`	On devices with normal screens, this will enable Android's auto-scaling mode. Unless you specifically develop for small- or large-screen devices, this is pointless.

Table 4.8 A synopsis of supports-screens settings and effects *(continued)*

Attribute set to false	Effect
`largeScreens`	On devices with large screens, your application will be displayed using the baseline configuration and scaled to that accordingly. If it still doesn't occupy the entire screen, the unused space will be painted in black (letterbox mode).
`xlargeScreens`	Same as `largeScreens`.
`anyDensity`	On devices with pixel densities deviating from the baseline density, Android will auto-scale all images (unless specifically prepared, see next technique) and absolute pixel values to match the different configuration.

Using the appropriate supports-screens, settings can be effective, which makes it easy to forward-enable legacy applications, but it has its down sides. To do things right you need to turn to Android's alternative resources framework.

TECHNIQUE 11 Loading configuration dependent resources

The mechanisms explained in the previous technique are a great way to easily enable legacy applications to support almost all screen configurations, without having to explicitly program for it. Still, this is merely a convenience and should by no means be considered good practice for applications that you develop today.

The drawbacks are obvious: no visibility on Android Market for small-screen devices, no guaranteed full-screen mode on large screen devices, and an often noticeable loss of quality for prescaled images (coupled with a slight loss in load times). What can you do to better support various display configurations?

PROBLEM

Instead of relying on Android's image prescaling, you want to supply resources such as layouts or images created for specific screen sizes or densities, so as to eliminate any loss of visual quality introduced by Android's scaling procedures.

SOLUTION

The solution is to leverage Android's alternative resources framework. We've already touched on how this works in chapter 2, where we mentioned that you can use several different string resource files for different languages. You do this by using different resource folders with separate resource values for each permutation you need to support (for example, `/res/values-en` for English strings, and `/res/values-de` for German strings). We can leverage the same system to provide configuration-specific resources such as drawables or layouts to Android. For these resources, Android assumes that they've been designed for that specific configuration and won't attempt to prescale them.

Say for instance we were to add a custom icon to MyMovies, one that says "MyMovies" on it. The problem with this is that, on a mid- to low-resolution screen, this text will be difficult or even impossible to read. Hence, we only want to show the full text when we run on an HDPI (high dots per inch) device, and abbreviate the text to "MM" on LDPI devices (low dots per inch—we don't change anything for normal configurations). For

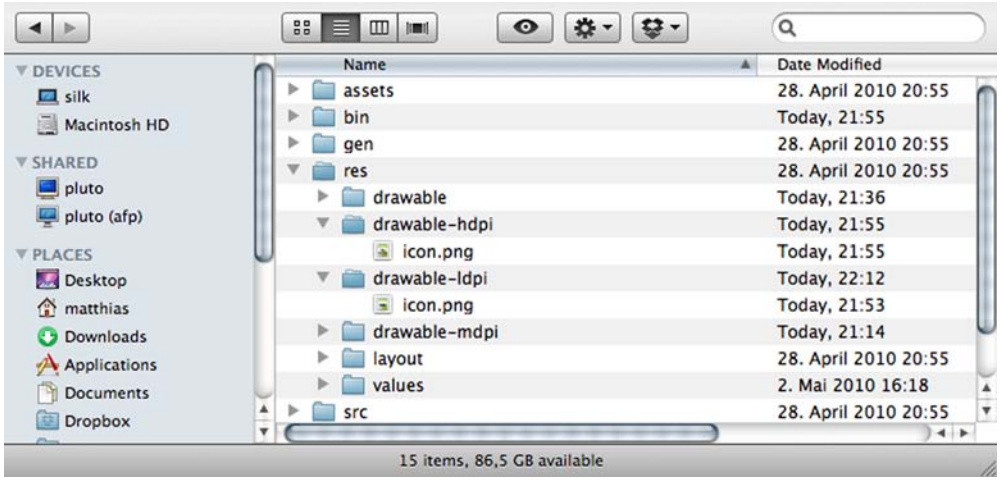

Figure 4.18 Supplying different image files for different screen configurations is done by placing them in the appropriate resource folders for a given configuration. Which folder a resource is loaded from at runtime is then determined by matching the current device configuration against the folder names.

this we need to create two variations of the standard icon and place them into the `drawable-hdpi` and `drawable-ldpi` folders respectively, as shown in figure 4.18.

We're now explicitly targeting low- and high-density devices by providing two new icon files specifically created for these screen densities. No more setup is required. Android will automatically find and load these files for you, even when you're running in fallback mode! Figure 4.19 shows how the new icons compare on both a large high-density screen, and a small low-density screen.

Figure 4.19 The two different icon files as rendered on the configurations they were made for: the icon with full text for HDPI screens (big picture), and the abbreviated version for LDPI screens (small picture).

There are plenty of ways you can leverage this technique; you could even load different strings (any kind of resource) for different screen sizes, but as you can imagine it's useful for images and layouts.

DISCUSSION

Screen densities aren't the only configuration options you can target. Much more can be encoded into resource folder names. You can load different resources based on language, SIM card country, touchscreen type, keyboard type, screen orientation, API level, and more. You can even combine them. Table 4.9 summarizes those configuration qualifiers that are relevant for screen support.

Table 4.9 Resource qualifiers relevant for screen support

Targeted attribute	Qualifiers	Examples

Screen size class	`small`—for small screens (about 2-3 inches)
	`normal`—for normal screens (baseline size, about 3-4 inches)
	`large`—for large screens (about 4-7 inches)
	`xlarge`—for very large screens (more than 7 inches)
		`/res/drawables-small`
		`/res/drawables-small-ldpi`
		`/res/layouts-normal-land`

Extended screen height	`long`—longer screens (such as WQVGA, WVGA, FWVGA)
	`notlong`—normal aspect ratio (such as QVGA, HVGA, and VGA)
		`/res/drawables-long`
		`/res/drawables-large-long`
		`/res/layouts-notlong-port`

Pixel density (dpi)	`ldpi`—low density (about 120dpi)
	`mdpi`—medium density (about 160dpi)
	`hdpi`—high density (about 240dpi)
	`xhdpi`—extra high density (about 320dpi)
	`nodpi`—disable scaling for these resources
		`/res/drawables-ldpi`
		`/res/drawables-large-mdpi`
		`/res/layouts-port-hdpi`

For a full list plus qualifier ordering rules, see http://mng.bz/d0M9.

Keep in mind that any qualified resource folder is completely optional. If you don't have any prescaled images in the `/res/drawable-hdpi` folder (or if it doesn't even exist), Android will still look for an image in this folder first, but if it can't find the image in there, it'll fall back to the default drawable folder. That means it's always safe to put all your stuff in the nonqualified resource folders; that way Android will always find a resource.

HOW ANDROID SELECTS RESOURCE FOLDERS If more than one folder qualifies for lookup, Android will load the resource from the folder that most closely matches the current configuration. The algorithm for this is quite refined, and is documented at http://mng.bz/7NiH.

Even though this technique can increase the size of your application when multiple versions of a given resource are bundled with it, it's a sensible choice for images, such as icons or window backgrounds that are likely to suffer a loss in quality when scaled. Consider again an icon being resized from 100 pixels to 150 pixels on an HDPI device. That's a 50% increase in pixels, and chances are that the image will look washed out when scaled. Nine-patch images on the other hand scale well by their nature, and are less problematic even when Android is in auto-scale mode.

Now that you've seen how to let Android handle everything and how to provide configuration specific resources, there's one more thing you should learn. It's last, but certainly not least: programming your application with different screen configurations up front.

TECHNIQUE 12 Programming pixel-independently

This is the last technique we're going to show you in this chapter, and it's short but important. The one big question that remains is, if we enable support for all screen densities in the `<supports-screens>` element, Android's auto-scaling logic will be disabled and we again have the problem that any values specified in absolute pixels will have different outcomes on different devices.

PROBLEM

Explicitly declaring support for screens that don't have the baseline pixel density will disable Android's auto-scaling mode, which means that any values specified in pixels won't scale to these devices.

SOLUTION

The solution is surprisingly simple: don't use absolute pixel values. Ever. The `px` unit is unsafe. As we've seen, any value specified in `px` is tailored toward the device you're developing on. So how should we specify positions and dimensions then? Android provides a set of density-independent units that, on a device using the baseline configuration, behave exactly as if specified in absolute pixels. On other screens, these same densities will be auto-scaled as seen before.

Remember how we defined our list selector's corner radius to be five pixels? Have a look at figure 4.20. On the left side you see the list selector as it's supposed to look; on the right side the corner radius appears to be *less* than the specified five pixels. Both screenshots were taken from an emulator instance running with a high-density screen configuration, but for the left image we used the *density-independent pixels* unit (`dip` or `dp`) to specify the corner radius, whereas on the right side we used the plain old `px` unit.

When specifying values in `dip`, this value will be considered to be the value in pixels you would've used to arrange the screen element on a device that uses the baseline configuration. This means that if you're running such a device, then you won't notice any difference between `px` and `dip`, but on devices with other pixel densities, Android makes sure you get the same result!

Figure 4.20 Corner radii mismatch on a high-density screen. On the left side, the radius was specified using density-independent pixels (dip), where the right side uses absolute pixels (px), resulting in a physically smaller corner radius.

DISCUSSION

Whenever possible, you should use density-independent units rather than absolute units when specifying positions or dimensions. Android defines two units you can use to auto-scale values:

- `dip` (alias `dp`)—Density-independent pixels, useful for specifying positions and dimensions in a scalable way
- `sip` (alias `sp`)—Scale-independent pixels, useful for specifying font sizes in a scalable way

You can and should use these units anywhere in your layouts or styles. If you need to specify a pixel value in your program code, but want to achieve the same effect, then you'll have to do the scaling yourself (unless you're running in fallback mode), because the SDK functions typically expect values to be in absolute pixels. The conversion is easy though. Here's one implementation of a helper function that does the scaling from dip to pixels for you:

```
public static int dipToPx(Activity context, int dip) {
      DisplayMetrics displayMetrics =
          context.getResources().getDisplayMetrics();

      return (int) (dip * displayMetrics.density + 0.5f);
}
```

With this helper you can write applications that scale across all kinds of displays, and not have to rely on Android's fallback mode anymore.

In this last section, we've taken you through three techniques that showed you how to prepare your legacy applications to run on devices with different screen configurations by leveraging Android's fallback mode. More importantly, we saw how to make

newly developed applications scale to different screen sizes and densities gracefully by means of customized resources and density-/scale-independent pixel units. It's time to wrap this chapter up.

4.8 Summary

In this chapter, we focused on the user interface. We've seen how to configure views in layouts, and how view hierarchies are drawn to the screen. We've also seen how all of the supplied Android layout managers work, and how they work with attributes to create structure. This where the UI starts.

From there we looked more closely at working with `ListView` to uncover a few handy features, such as header and footer views, and to see how to maintain state between views and the data model exposed by an `Adapter`. This helped us focus on working more closely with this common and powerful widget. Also, while working with `ListView` we saw how to reuse styles rather than repeat look and feel values on every view, and how to go even further and create and apply themes.

Along with themes, we learned how to take the UI to the next level by working with and defining drawables. We also learned how to provide device-independent resources for different device configurations. This allowed us to create pixel-perfect layouts and images so that our application looks the way we expect on many different screen sizes.

Overall, we've now seen a good deal about the basics of building Android applications, and how to perfect the form (the UI). Now it's time to move toward function in the next chapter, where we'll depart from the UI and hone in on a new topic: background services.

MOBILE DEVELOPMENT

Android IN PRACTICE

Collins • Galpin • Käppler

It's not hard to find the information you need to build your first Android app. Then what? If you want to build real apps, you will need some how-to advice, and that's what this book is about.

Android in Practice is a rich source of Android tips, tricks, and best practices, covering over 90 clever and useful techniques that will make you a more effective Android developer. Techniques are presented in an easy-to-read problem/solution/discussion format. The book dives into important topics like multitasking and services, testing and instrumentation, building and deploying applications, and using alternative languages.

What's Inside

- Techniques covering Android 1.x to 3.x
- Android for tablets
- Working with threads and concurrency
- Testing and building
- Using location awareness and GPS
- Styles and themes
- And much more!

This book requires a working knowledge of Java, but no prior experience with Android is assumed.

Charlie Collins is a mobile and web developer at MOVL, a contributor to several open source projects, and a coauthor of *GWT in Practice* and *Unlocking Android*. **Michael Galpin** is a developer at Bump Technologies and worked on two of the most downloaded apps on the Android Market, Bump, and eBay Mobile. **Matthias Käppler** is an Android and API engineer at Qype.

For access to the book's forum and a free ebook for owners of this book, go to www.manning.com/AndroidinPractice

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]

"In-depth coverage of the No. 1 smartphone platform."
—Gabor Paller, Ericsson

"Practical and immediately useful."
—Kevin McDonagh
Novoda

"Gets you thinking with an Android mindset."
—Norman Klein
POSMobility

"The hows and the whys. Highly Recommended!"
—Al Scherer, Follett Higher Education Group

"Go from regular old Java developer to cool Android app author!"
—Cheryl Jerozal, Atlassian

ISBN 13: 978-1-935182-92-4
ISBN 10: 1-935182-92-7
54999
9 781935 182924