Defining Integer Resources in XML

In addition to strings and Boolean values, you can also store integers as resources. Integer resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

Integer values are appropriately tagged with the <integer> tag and represent a namevalue pair. The name attribute is how you refer to the specific integer programmatically, so name these resources wisely.

Here's an example of the integer resource file /res/values/nums.xml:

Using Integer Resources Programmatically

To use the integer resource, you must load it using the Resource class. The following code accesses your application's integer resource named numTimesToRepeat:

int repTimes = getResources().getInteger(R.integer.numTimesToRepeat);



Тір

Much like string arrays, you can create integer arrays as resources using the <integer-array> tag with child <item> tags, defining one for each item in the array. You can then load the integer array using the getIntArray() method of the Resource class.

Working with Colors

Android applications can store RGB color values, which can then be applied to other screen elements. You can use these values to set the color of text or other elements, such as the screen background. Color resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

RGB color values always start with the hash symbol (#). The alpha value can be given for transparency control. The following color formats are supported:

- #RGB (example, #F00 is 12-bit color, red)
- #ARGB (example, #8F00 is 12-bit color, red with alpha 50%)
- #RRGGBB (example, #FF00FF is 24-bit color, magenta)
- #AARRGGBB (example, #80FF00FF is 24-bit color, magenta with alpha 50%)

Color values are appropriately tagged with the <color> tag and represent a name-value pair. Here's an example of a simple color resource file /res/values/colors.xml:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<resources>
<color name="background_color">#006400</color>
<color name="text_color">#FFE4C4</color>
</resources>
```

The example at the beginning of the chapter accessed a color resource. Color resources are simply integers. The following code retrieves a color resource called prettyTextColor:

```
int myResourceColor =
   getResources().getColor(R.color.prettyTextColor);
```

Working with Dimensions

Many user interface layout controls such as text controls and buttons are drawn to specific dimensions. These dimensions can be stored as resources. Dimension values always end with a unit of measurement tag.

Dimension values are appropriately tagged with the <dimen> tag and represent a namevalue pair. Dimension resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

The dimension units supported are shown in Table 6.4.

Unit of Measurement	Description	Resource Tag Required	Example
Pixels	Actual screen pixels	px	20px
Inches	Physical measurement	in	lin
Millimeters	Physical measurement	mm	1mm
Points	Common font measurement unit	pt	14pt
Screen density independent pixels	Pixels relative to 160dpi screen (preferable dimension for screen compatibility)	dp	ldp
Scale independent pixels	Best for scalable font display	sp	14sp

Table 6.4 Dimension Unit Measurements Supported

Here's an example of a simple dimension resource file /res/values/dimens.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="FourteenPt">14pt</dimen>
    <dimen name="OneInch">1in</dimen>
    <dimen name="TenMillimeters">10mm</dimen>
    <dimen name="TenPixels">10px</dimen>
</resources>
```

Dimension resources are simply floating point values. The following code retrieves a dimension resource called textPointSize:

```
float myDimension =
   getResources().getDimension(R.dimen.textPointSize);
```



Warning

Be cautious when choosing dimension units for your applications. If you are planning to target multiple devices, with different screen sizes and resolutions, then you need to rely heavily on the more scalable dimension units, such as dp and sp, as opposed to pixels, points, inches, and millimeters.

Working with Simple Drawables

You can specify simple colored rectangles by using the drawable resource type, which can then be applied to other screen elements. These drawable types are defined in specific paint colors, much like the Color resources are defined.

Simple paintable drawable resources are defined in XML under the /res/values project directory and compiled into the application package at build time. Paintable drawable resources use the <drawable> tag and represent a name-value pair. Here's an example of a simple drawable resource file /res/values/drawables.xml:

Although it might seem a tad confusing, you can also create XML files that describe other Drawable subclasses, such as ShapeDrawable. Drawable XML definition files are stored in the /res/drawable directory within your project along with image files. This is not the same as storing <drawable> resources, which are paintable drawables. PaintableDrawable resources are stored in the /res/values directory, as explained in the previous section.

Here's a simple ShapeDrawable described in the file /res/drawable/red_oval.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
    xmlns:android=
        "http://schemas.android.com/apk/res/android"
        android:shape="oval">
        <solid android:color="#f00"/>
        </shape>
```

We talk more about graphics and drawing shapes in Chapter 9, "Drawing and Working with Animation."

Drawable resources defined with <drawable> are simply rectangles of a given color, which is represented by the Drawable subclass ColorDrawable. The following code retrieves a ColorDrawable resource called redDrawable:

```
import android.graphics.drawable.ColorDrawable;
...
```

```
ColorDrawable myDraw = (ColorDrawable)getResources().
getDrawable(R.drawable.redDrawable);
```



Тір

There are many additional drawable resource types that can be specified as XML resources. These special drawables correspond to specific drawable classes such as ClipDrawable and LevelListDrawable. For information on these specialized drawable types, see the Android SDK documentation.

Working with Images

Applications often include visual elements such as icons and graphics. Android supports several image formats that can be directly included as resources for your application. These image formats are shown in Table 6.5.

e		
Supported Image Format	Description	Required Extension
Portable Network Graphics (PNG)	Preferred Format (Lossless)	.png
Nine-Patch Stretchable Images	Preferred Format (Lossless)	.9.png

Table 6.5 Image Formats Supported in Android

Supported Image Format	Description	Required Extension
Joint Photographic Experts Group (JPEG)	Acceptable Format (Lossy)	.jpg, .jpeg
Graphics Interchange Format (GIF)	Discouraged Format	.gif

Table 6.5 Continued

These image formats are all well supported by popular graphics editors such as Adobe Photoshop, GIMP, and Microsoft Paint. The Nine-Patch Stretchable Graphics can be created from PNG files using the draw9patch tool included with the Android SDK under the /tools directory.



Warning

All resources filenames must be lowercase and simple (letters, numbers, and underscores only). This rule applies to all files, including graphics.

Adding image resources to your project is easy. Simply drag the image asset into the /res/drawable directory, and it is automatically included in the application package at build time.

Working with Nine-Patch Stretchable Graphics

Phone screens come in various dimensions. It can be handy to use stretchable graphics to allow a single graphic that can scale appropriately for different screen sizes and orientations or different lengths of text. This can save you or your designer a lot of time in creating graphics for many different screen sizes.

Android supports Nine-Patch Stretchable Graphics for this purpose. Nine-Patch graphics are simply PNG graphics that have patches, or areas of the image, defined to scale appropriately, instead of scaling the entire image as one unit. Often the center segment is transparent.

Nine-Patch Stretchable Graphics can be created from PNG files using the draw9patch tool included with the Tools directory of the Android SDK. We talk more about compatibility and using Nine-Patch graphics in Chapter 25.

Using Image Resources Programmatically

Images resources are simply another kind of Drawable called a BitmapDrawable. Most of the time, you need only the resource ID of the image to set as an attribute on a user interface control.

For example, if I drop the graphics file flag.png into the /res/drawable directory and add an ImageView control to my main layout, we can set the image to be displayed programmatically in the layout this way:

```
import android.widget.ImageView;
```

• • •

```
ImageView flagImageView =
   (ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
```

If you want to access the BitmapDrawable object directly, you simply request that resource directly, as follows:

```
import android.graphics.drawable.BitmapDrawable;
...
BitmapDrawable bitmapFlag = (BitmapDrawable)
    getResources().getDrawable(R.drawable.flag);
int iBitmapHeightInPixels =
    bitmapFlag.getIntrinsicHeight();
int iBitmapWidthInPixels = bitmapFlag.getIntrinsicWidth();
```

Finally, if you work with Nine-Patch graphics, the call to getDrawable() returns a NinePatchDrawable instead of a BitmapDrawable object.

```
import android.graphics.drawable.NinePatchDrawable;
...
NinePatchDrawable stretchy = (NinePatchDrawable)
    getResources().getDrawable(R.drawable.pyramid);
int iStretchyHeightInPixels =
    stretchy.getIntrinsicHeight();
int iStretchyWidthInPixels = stretchy.getIntrinsicWidth();
```



Тір

There is also a special resource type called <selector>, which can be used to define different colors or drawables to be used depending on a control's state. For example, you could define a color state list for a Button control: gray when the button is disabled, green when it is enabled, and yellow when it is being pressed. Similarly, you could provide different drawables based on the state of an ImageButton control. For more information, see the Android SDK documentation regarding the color and drawable state list resources.

Working with Animation

Android supports frame-by-frame animation and tweening. Frame-by-frame animation involves the display of a sequence of images in rapid succession. Tweened animation involves applying standard graphical transformations such as rotations and fades upon a single image.

The Android SDK provides some helper utilities for loading and using animation resources. These utilities are found in the android.view.animation.AnimationUtils class.

We discuss animation in detail in Chapter 9. For now, let's just look at how you define animation data in terms of resources.

Defining and Using Frame-by-Frame Animation Resources

Frame-by-frame animation is often used when the content changes from frame to frame. This type of animation can be used for complex frame transitions—much like a kid's flip-book.

To define frame-by-frame resources, take the following steps:

- 1. Save each frame graphic as an individual drawable resource. It may help to name your graphics sequentially, in the order in which they are displayed—for example, frame1.png, frame2.png, and so on.
- 2. Define the animation set resource in an XML file within /res/drawable/ resource directory.
- 3. Load, start, and stop the animation programmatically.

Here's an example of a simple frame-by-frame animation resource file /res/drawable/juggle.xml that defines a simple three-frame animation that takes 1.5 seconds:

```
<?xml version="1.0" encoding="utf-8" ?>
<animation-list
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:oneshot="false">
    <item
        android:drawable="@drawable/splash1"
        android:duration="50" />
   <item
        android:drawable="@drawable/splash2"
        android:duration="50" />
   <item
        android:drawable="@drawable/splash2"
        android:duration="50" />
   <item
        android:drawable="@drawable/splash2"
        android:duration="50" />
   </item
        android:duration="50" />
   </item
        android:drawable="@drawable/splash2"
        android:duration="50" />
   </item
        android:drawable="@drawable/splash3"
        android:duration="50" />
   </item
        android:duration="50" />
```

Frame-by-frame animation set resources defined with <animation-list> are represented by the Drawable subclass AnimationDrawable. The following code retrieves an Animation-Drawable resource called juggle:

```
import android.graphics.drawable.AnimationDrawable;
...
AnimationDrawable jugglerAnimation = (AnimationDrawable)getResources().
getDrawable(R.drawable.juggle);
```

After you have a valid AnimationDrawable, you can assign it to a View on the screen and use the Animation methods to start and stop animation.

Defining and Using Tweened Animation Resources

Tweened animation features include scaling, fading, rotation, and translation. These actions can be applied simultaneously or sequentially and might use different interpolators.

Tweened animation sequences are not tied to a specific graphic file, so you can write one sequence and then use it for a variety of different graphics. For example, you can make moon, star, and diamond graphics all pulse using a single scaling sequence, or you can make them spin using a rotate sequence.

Graphic animation sequences can be stored as specially formatted XML files in the /res/anim directory and are compiled into the application binary at build time.

Here's an example of a simple animation resource file /res/anim/spin.xml that defines a simple rotate operation—rotating the target graphic counterclockwise four times in place, taking 10 seconds to complete:

If we go back to the example of a BitmapDrawable earlier, we can now add some animation simply by adding the following code to load the animation resource file spin.xml and set the animation in motion:

```
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;
...
ImageView flagImageView =
    (ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
...
Animation an =
    Animation an =
    AnimationUtils.loadAnimation(this, R.anim.spin);
flagImageView.startAnimation(an);
```

Now you have your graphic spinning. Notice that we loaded the animation using the base class object Animation. You can also extract specific animation types using the subclasses that match: RotateAnimation, ScaleAnimation, TranslateAnimation, and AlphaAnimation.

There are a number of different interpolators you can use with your tweened animation sequences.

Working with Menus

You can also include menu resources in your project files. Like animation resources, menu resources are not tied to a specific control but can be reused in any menu control.

Each menu resource (which is a set of individual menu items) is stored as a specially formatted XML files in the /res/menu directory and are compiled into the application package at build time.

Here's an example of a simple menu resource file /res/menu/speed.xml that defines a short menu with four items in a specific order:

```
<menu xmlns:android
   ="http://schemas.android.com/apk/res/android">
   <item
        android:id="@+id/start"
        android:title="Start!"
        android:orderInCategory="1"></item>
    <item
        android:id="@+id/stop"
        android:title="Stop!"
        android:orderInCategory="4"></item>
    <item
        android:id="@+id/accel"
        android:title="Vroom! Accelerate!"
        android:orderInCategory="2"></item>
    <item
        android:id="@+id/decel"
        android:title="Decelerate!"
        android:orderInCategory="3"></item>
</menu>
```

You can create menus using the Eclipse plug-in, which can access the various configuration attributes for each menu item. In the previous case, we set the title (label) of each menu item and the order in which the items display. Now, you can use string resources for those titles, instead of typing in the strings. For example:

```
<menu xmlns:android=
    "http://schemas.android.com/apk/res/android">
   <item
        android:id="@+id/start"
        android:title="@string/start"
        android:orderInCategory="1"></item>
   <item
        android:id="@+id/stop"
        android:title="@string/stop"
        android:orderInCategory="2"></item>
```

```
</menu>
```

To access the preceding menu resource called /res/menu/speed.xml, simply override the method onCreateOptionsMenu() in your application:

```
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.speed, menu);
   return true:
```

```
}
```

That's it. Now if you run your application and press the menu button, you see the menu. There are a number of other XML attributes that can be assigned to menu items. For a complete list of these attributes, see the Android SDK reference for menu resources at the website http://d.android.com/guide/topics/resources/menu-resource.html.You learn a lot more about menus and menu event handling in Chapter 7, "Exploring User Interface Screen Elements."

Working with XML Files

You can include arbitrary XML resource files to your project. You should store these XML files in the /res/xml directory, and they are compiled into the application package at build time.

The Android SDK has a variety of packages and classes available for XML manipulation. You learn more about XML handling in Chapter 10, "Using Android Data and Storage APIs," Chapter 11, "Sharing Data Between Applications with Content Providers," and Chapter 12, "Using Android Networking APIs." For now, we create an XML resource file and access it through code.

First, put a simple XML file in /res/xml directory. In this case, the file my pets.xml with the following contents can be created:

```
<?xml version="1.0" encoding="utf-8"?>
<pets>
    <pet name="Bit" type="Bunny" />
    <pet name="Nibble" type="Bunny" />
    <pet name="Stack" type="Bunny" />
   <pet name="Queue" type="Bunny" />
    <pet name="Heap" type="Bunny" />
   <pet name="Null" type="Bunny" />
   <pet name="Nigiri" type="Fish" />
    <pet name="Sashimi II" type="Fish" />
   <pet name="Kiwi" type="Lovebird" />
```

</pets>

Now you can access this XML file as a resource programmatically in the following manner:

```
XmlResourceParser myPets =
    getResources().getXml(R.xml.my pets);
```

Finally, to prove this is XML, here's one way you might churn through the XML and extract the information:

```
import org.xmlpull.v1.XmlPullParserException;
import android.content.res.XmlResourceParser;
. . .
int eventType = -1;
while (eventType != XmlResourceParser.END DOCUMENT) {
    if(eventType == XmlResourceParser.START DOCUMENT) {
        Log.d(DEBUG TAG, "Document Start");
    } else if(eventType == XmlResourceParser.START TAG) {
        String strName = myPets.getName();
        if(strName.equals("pet")) {
            Log.d(DEBUG TAG, "Found a PET");
            Log.d(DEBUG TAG,
                "Name: "+myPets.
                getAttributeValue(null, "name"));
            Log.d(DEBUG TAG,
                "Species: "+myPets.
                getAttributeValue(null, "type"));
        }
    }
   eventType = myPets.next();
}
Log.d(DEBUG TAG, "Document End");
```

Working with Raw Files

Your application can also include raw files as part of its resources. For example, your application might use raw files such as audio files, video files, and other file formats not supported by the Android Resource packaging tool aapt.

All raw resource files are included in the /res/raw directory and are added to your package without further processing.



Warning

All resources filenames must be lowercase and simple (letters, numbers, and underscores only). This also applies to raw file filenames even though the tools do not process these files other than to include them in your application package.

The resource filename must be unique to the directory and should be descriptive because the filename (without the extension) becomes the name by which the resource is accessed. You can access raw file resources and any resource from the /res/drawable directory (bitmap graphics files, anything not using the <resource> XML definition method). Here's one way to open a file called the_help.txt:

```
import java.io.InputStream;
...
InputStream iFile =
   getResources().openRawResource(R.raw.the help);
```

References to Resources

You can reference resources instead of duplicating them. For example, your application might want to reference a single string resource in multiple string arrays.

The most common use of resource references is in layout XML files, where layouts can reference any number of resources to specify attributes for layout colors, dimensions, strings, and graphics. Another common use is within style and theme resources.

Resources are referenced using the following format:

]resource_type/variable_name

Recall that earlier we had a string-array of soup names. If we want to localize the soup listing, a better way to create the array is to create individual string resources for each soup name and then store the references to those string resources in the string-array (instead of the text).

To do this, we define the string resources in the /res/strings.xml file like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Application Name</string>
        <string name="chicken_soup"
            >Organic Chicken Noodle</string>
            <string name="minestrone_soup"
            >Veggie Minestrone</string>
            <string name="chowder_soup"
            >New England Lobster Chowder</string>
</resources>
```

And then we can define a localizable string-array that references the string resources by name in the /res/arrays.xml file like this:

Ô

Тір

Save the strings.xml file first so that the string resources (which are picked up by the aapt and included in the R.java class) are defined prior to trying to save the arrays.xml file, which references those particular string resources. Otherwise, you might get the following error:

Error: No resource found that matches the given name.

You can also use references to make aliases to other resources. For example, you can alias the system resource for the OK string to an application resource name by including the following in your strings.xml resource file:

You learn more about all the different system resources available later in this chapter.



Тір

Much like string and integer arrays, you can create arrays of any type of resources using the <array> tag with child <item> tags, defining one item for each resource in the array. You can then load the array of miscellaneous resources using the obtainTypedArray() method of the Resource class. The typed array resource is commonly used for grouping and loading a bunch of Drawable resources with a single call. For more information, see the Android SDK documentation on typed array resources.

Working with Layouts

Much as web designers use HTML, user interface designers can use XML to define Android application screen elements and layout. A layout XML resource is where many different resources come together to form the definition of an Android application screen. Layout resource files are included in the /res/layout/ directory and are compiled into the application package at build time. Layout files might include many user interface controls and define the layout for an entire screen or describe custom controls used in other layouts.

Here's a simple example of a layout file (/res/layout/main.xml) that sets the screen's background color and displays some text in the middle of the screen (see Figure 6.3).

The main.xml layout file that displays this screen references a number of other resources, including colors, strings, and dimension values, all of which were defined in the strings.xml, colors.xml, and dimens.xml resource files. The color resource for the screen background color and resources for a TextView control's color, string, and text size follows:

```
<?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"
android:id="@+id/TextView01"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:text="@string/test_string"
android:textColor="@color/text_color"
android:gravity="center"
android:textSize="@dimen/text_size"></TextView>
</LinearLayout>
```

Figure 6.3 How the **main.xml** layout file displays in the emulator.

The preceding layout describes all the visual elements on a screen. In this example, a LinearLayout control is used as a container for other user interface controls—here, a single TextView that displays a line of text.



Тір

You can encapsulate common layout definitions in their own XML files and then include those layouts within other layout files using the <include> tag. For example, you can use the following <include> tag to include another layout file called /res/layout/mygreenrect.xml within the main.xml layout definition:

<include layout="@layout/mygreenrect"/>

Designing Layouts in Eclipse

Layouts can be designed and previewed in Eclipse using the Resource editor functionality provided by the Android plug-in (see Figure 6.4). If you click the project file /res/layout/main.xml (provided with any new Android project), you see a Layout tab, which shows you the preview of the layout, and a main.xml tab, which shows you the raw XML of the layout file.



Figure 6.4 Designing a layout file using Eclipse.

As with most user interface designers, the Android plug-in works well for your basic layout needs, enables you to create user interface controls such as TextView and Button controls easily, and enables setting the controls' properties in the Properties pane.



Тір

Moving the Properties pane to the far right of the workspace in Eclipse makes it easier to browse and set control properties when designing layouts.

Now is a great time to get to know the layout resource designer. Try creating a new Android project called ParisView (available as a sample project). Navigate to the /res/layout/main.xml layout file and double-click it to open it in the resource editor. It's quite simple by default, only a black (empty) rectangle and string of text.

Below in the Resource pane of the Eclipse perspective, you notice the Outline tab. This outline is the XML hierarchy of this layout file. By default, you see a LinearLayout. If you expand it, you see it contains one TextView control. Click on the TextView control. You see that the Properties pane of the Eclipse perspective now has all the properties available for that object. If you scroll down to the property called text, you see that it's set to a string resource variable @string/hello.



Тір

You can also select specific controls by clicking them in the layout designer preview area. The currently selected control is highlighted in red. We prefer to use the Outline view, so we can be sure we are clicking what we expect.

You can use the layout designer to set and preview layout control properties. For example, you can modify the TextView property called text Size by typing 18pt (a dimension). You see the results of your change to the property immediately in the preview area.

Take a moment to switch to the main.xml tab. You notice that the properties you set are now in the XML. If you save and run your project in the emulator now, you see similar results to what you see in the designer preview.

Now go back to the Outline pane. You see a green plus and a red minus button. You can use these buttons to add and remove controls to your layout file. For example, select the LinearLayout from the Outline view, and click the green button to add a control within that container object.

Choose the ImageView object. Now you have a new control in your layout. You can't actually see it yet because it is not fully defined.

Drag two PNG graphics files (or JPG) into your /res/drawable project directory, naming them flag.png and background.png. Now, browse the properties of your ImageView control, and set the Src property by clicking on the resource browser button labeled [...].You can browse all the Drawable resources in your project and select the flag resource you just added.You can also set this property manually by typing @drawable/flag.

Now, you see that the graphic shows up in your preview. While we're at it, select the LinearLayout object and set its background property to the background Drawable you added.

If you save the layout file and run the application in the emulator (see Figure 6.5) or on the phone, you see results much like you did in the resource designer preview pane.

Using Layout Resources Programmatically

Layouts, whether they are Button or ImageView controls, are all derived from the View class. Here's how you would retrieve a TextView object named TextView01:

```
TextView txt = (TextView)findViewById(R.id.TextView01);
```

You can also access the underlying XML of a layout resource much as you would any XML file. The following code retrieves the main.xml layout file for XML parsing:

```
XmlResourceParser myMainXml =
  getResources().getLayout(R.layout.main);
```



Figure 6.5 A layout with a LinearLayout, TextView, and ImageView, shown in the Android emulator.

Developers can also define custom layouts with unique attributes. We talk much more about layout files and designing Android user interfaces in Chapter 8, "Designing User Interfaces with Layouts."



Warning

Take care when providing alternative layout resources. Layout resources tend to be complicated, and the child controls within them are often referred to in code by name. Therefore, if you create an alternative layout resource, make sure each important control exists in the layout and is named the same. For example, if both layouts have a Button control, make sure its identifier (android:id) is the same in both the landscape and portrait mode alternative layout resources. You may include different controls in the layouts, but the important ones (those referred to and interacted with programmatically) should match in both layouts.

Working with Styles

Android user interface designers can group layout element attributes together in styles. Layout controls are all derived from the View base class, which has many useful attributes. Individual controls, such as Checkbox, Button, and TextView, have specialized attributes associated with their behavior. Styles are tagged with the <style> tag and should be stored in the /res/values/ directory. Style resources are defined in XML and compiled into the application binary at build time.



qiT

Styles cannot be previewed using the Eclipse Resource designer but they are displayed correctly in the emulator and on the device.

Here's an example of a simple style resource file /res/values/styles.xml containing two styles: one for mandatory form fields, and one for optional form fields on TextView and EditText objects:

Many useful style attributes are colors and dimensions. It would be more appropriate to use references to resources. Here's the styles.xml file again; this time, the color and text size fields are available in the other resource files colors.xml and dimens.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="mandatory text field style">
        <item name="android:textColor"
            >@color/mand text color</item>
       <item name="android:textSize"
            >@dimen/important text</item>
        <item name="android:textStyle">bold</item>
    </style>
    <style name="optional text field style">
        <item name="android:textColor"
            >@color/opt text color</item>
       <item name="android:textSize"
            >@dimen/unimportant text</item>
        <item name="android:textStyle">italic</item>
    </style>
</resources>
```

Now, if you can create a new layout with a couple of TextView and EditText text controls, you can set each control's style attribute by referencing it as such:

```
style="@style/name of style"
```

Here we have a form layout called /res/layout/form.xml that does that:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLavout
   xmlns:android=
        "http://schemas.android.com/apk/res/android"
    android: orientation="vertical"
    android:layout width="fill parent"
    android: layout height="fill parent"
    android:background="@color/background color">
    <TextView
        android:id="@+id/TextView01"
        style="@style/mandatory text field style"
        android:layout height="wrap content"
        android:text="@string/mand label"
        android:layout width="wrap content" />
    <EditText
        android:id="@+id/EditText01"
        style="@style/mandatory text field style"
        android:layout height="wrap content"
        android:text="@string/mand_default"
        android: layout width="fill parent"
        android:singleLine="true" />
    <TextView
        android:id="@+id/TextView02"
        style="@style/optional text field style"
        android:layout width="wrap content"
        android: layout height="wrap content"
        android:text="@string/opt label" />
    <EditText
        android:id="@+id/EditText02"
        style="@style/optional_text_field_style"
        android: layout height="wrap content"
        android:text="@string/opt default"
        android:singleLine="true"
        android:layout width="fill parent" />
    <TextView
        android:id="@+id/TextView03"
        style="@style/optional text field style"
        android:layout_width="wrap content"
        android:layout height="wrap content"
        android:text="@string/opt label" />
    <EditText
```

```
android:id="@+id/EditText03"
style="@style/optional_text_field_style"
android:layout_height="wrap_content"
android:text="@string/opt_default"
android:singleLine="true"
android:layout_width="fill_parent" />
inearLayout>
```

```
</LinearLayout>
```

The resulting layout has three fields, each made up of one TextView for the label and one EditText where the user can input text. The mandatory style is applied to the mandatory label and text entry. The other two fields use the optional style. The resulting layout would look something like Figure 6.6.

Mandatory Field: Must Type Here Optional Field: Can Type Here Optional Field: Can Type Here	езоurce Viewer
Must Type Here <i>Optional Field:</i> <i>Can Type Here</i> <i>Optional Field:</i> <i>Can Type Here</i>	Mandatory Field:
<i>Optional Field: Can Type Here Optional Field: Can Type Here</i>	Must Type Here
<i>Can Type Here Optional Field: Can Type Here</i>	Optional Field:
<i>Optional Field: Can Type Here</i>	Can Type Here
<i>Can Type Here</i>	Optional Field:
	<i>Can Type Here</i>



We talk more about styles in Chapter 7.

Using Style Resources Programmatically

Styles are applied to specific layout controls such as TextView and Button objects. Usually, you want to supply the style resource id when you call the control's constructor. For example, the style named myAppIsStyling would be referred to as R.style.myAppIsStyling.

Working with Themes

Themes are much like styles, but instead of being applied to one layout element at a time, they are applied to all elements of a given activity (which, generally speaking, means one screen).

Themes are defined in exactly the same way as styles. Themes use the <style> tag and should be stored in the /res/values directory. The only difference is that instead of applying that named style to a layout element, you define it as the theme attribute of an activity in the AndroidManifest.xml file.

We talk more about themes in Chapter 7.

Referencing System Resources

You can access system resources in addition to your own resources. The android package contains all kinds of resources, which you can browse by looking in the android.R subclasses. Here you find system resources for

- Animation sequences for fading in and out
- Arrays of email/phone types (home, work, and such)
- Standard system colors
- Dimensions for application thumbnails and icons
- Many commonly used drawable and layout types
- Error strings and standard button text
- System styles and themes

You can reference system resources the same way you use your own; set the package name to android. For example, to set the background to the system color for darker gray, you set the appropriate background color attribute to @android:color/darker_gray.

You can access system resources much like you access your application's resources. Instead of using your application resources, use the Android package's resources under the android.R class.

If we go back to our animation example, we could have used a system animation instead of defining our own. Here is the same animation example again, except it uses a system animation to fade in:

```
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;
...
ImageView flagImageView =
   (ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
...
Animation an = AnimationUtils.
   loadAnimation(this, android.R.anim.fade_in);
flagImageView.startAnimation(an);
```



Note

The default Android resources are provided as part of the Android SDK under the /platform_version>/data/res directory on newer SDK installations, and in the /tools/lib/res/default directory for older installations. Here you can examine all the drawable resources, full XML layout files, and everything else found in the android.R.* package.

Summary

Android applications rely on various types of resources, including strings, string arrays, colors, dimensions, drawable objects, graphics, animation sequences, layouts, styles, and themes. Resources can also be raw files. Many of these resources are defined with XML and organized into specially named project directories. Both default and alternative resources can be defined using this resource hierarchy.

Resources are compiled and accessed using the R. java class file, which is automatically generated when the application resources are compiled. Developers access application and system resources programmatically using this special class.

References and More Information

Android Dev Guide: Application Resources: http://d.android.com/guide/topics/resources/index.html Android Dev Guide: Resource Types: http://d.android.com/guide/topics/resources/available-resources.html