Creating New Views

The ability to extend existing Views, create composite widgets, and create unique new controls lets you create beautiful User Interfaces optimized for your particular workfl ow. Android lets you subclass the existing widget toolbox and implement your own View controls, giving you total freedom to tailor your User Interface to maximize the user experience.

When you design a User Interface, it's important to balance raw aesthetics and usability. With the power to create your own custom controls comes the temptation to rebuild all of them from scratch. Resist that urge. The standard widgets will be familiar to users from other Android applications. On small screens with users often paying limited attention, familiarity can often provide better usability than a slightly shinier widget.

Deciding on your approach when creating a new View depends on what you want to achieve:

❑ Modify or extend the appearance and/or behavior of an existing control when it already supplies the basic functionality you want. By overriding the event handlers and onDraw, but still calling back to the superclass's methods, you can customize the control without having to reimplement its functionality. For example, you could customize a TextView to display a set number of decimal points.

❑ Combine controls to create atomic, reusable widgets that leverage the functionality of several interconnected controls. For example, you could create a dropdown combo box by combining a TextView and a Button that displays a fl oating ListView when clicked.

❑ Create an entirely new control when you need a completely different interface that can't be achieved by changing or combining existing controls.

Modifying Existing Views

The toolbox includes a lot of common UI requirements, but the controls are necessarily generic. By customizing these basic Views, you avoid reimplementing existing behavior while still tailoring the User Interface, and functionality, of each control to your application's needs.

To create a new widget based on an existing control, create a new class that extends it — as shown in the following skeleton code that extends TextView:

```
import android.content.Context;
import android.util.AttributeSet;
import android.widget.TextView;
public class MyTextView extends TextView {
public MyTextView (Context context, AttributeSet attrs, int defStyle)
{
super(context, attrs, defStyle);
}
public MyTextView (Context context) {
super(context);
}
public MyTextView (Context context, AttributeSet attrs) {
super(context, attrs);
}
}
```

To override the appearance or behavior of your new View, override and extend the event handlers associated with the behavior you want to change.

In the following skeleton code, the onDraw method is overridden to modify the View's appearance, and the onKeyDown handler is overridden to allow custom key press handling:

```
public class MyTextView extends TextView {
public MyTextView (Context context, AttributeSet ats, int defStyle) {
super(context, ats, defStyle);
}
public MyTextView (Context context) {
super(context);
}
public MyTextView (Context context, AttributeSet attrs) {
super(context, attrs);
```

```
}
@Override
public void onDraw(Canvas canvas) {
[ ... Draw things on the canvas under the text ... ]
// Render the text as usual using the TextView base class.
super.onDraw(canvas);
[ ... Draw things on the canvas over the text ... ]
}
@Override
public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
[ ... Perform some special processing ... ]
[ ... based on a particular key press ... ]
// Use the existing functionality implemented by
// the base class to respond to a key press event.
return super.onKeyDown(keyCode, keyEvent);
}
}
```

The User Interface event handlers available within Views are covered in more detail later in this chapter.

Customizing Your To-Do List

The To-Do List example from Chapter 2 uses TextViews (within a List View) to display each item. You can customize the appearance of the list by creating a new extension of the Text View, overriding the onDraw method. In this example, you'll create a new TodoListItemView that will make each item appear as if on a paper pad. When complete, your customized To-Do List should look like Figure 4-2.

Figure 4-2

1. Create a new TodoListItemView class that extends TextView. Include a stub for overriding the onDraw method, and implement constructors that call a new init method stub.

```
package com.paad.todolist;
import android.content.Context;
import android.content.res.Resources;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.widget.TextView;
public class TodoListItemView extends TextView {
public TodoListItemView (Context context, AttributeSet ats, int ds) {
super(context, ats, ds);
init();
}
public TodoListItemView (Context context) {
super(context);
init();
}
public TodoListItemView (Context context, AttributeSet attrs) {
super(context, attrs);
init();
}
private void init() {
}
@Override
```

```
public void onDraw(Canvas canvas) {
// Use the base TextView to render the text.
super.onDraw(canvas);
}
}
```

2. Create a new colors.xml resource in the res/values folder. Create new color values for the paper, margin, line, and text colors.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<color name="notepad_paper">#AAFFFF99</color>
<color name="notepad_lines">#FF0000FF</color>
<color name="notepad_margin">#90FF0000</color>
<color name="notepad_text">#AA0000FF</color>
</resources>
```

3. Create a new dimens.xml resource fi le, and add a new value for the paper's margin width.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<dimen name="notepad_margin">30px</dimen>
</resources>
```

4. With the resources defi ned, you're ready to customize the TodoListItemView appearance. Create new private instance variables to store the Paint objects you'll use to draw the paper background and margin. Also create variables for the paper color and margin width values. Fill in the init method to get instances of the resources you created in the last two steps and create the Paint objects.

```
private Paint marginPaint;
private Paint linePaint;
private int paperColor;
private float margin;
private void init() {
// Get a reference to our resource table.
Resources myResources = getResources();
// Create the paint brushes we will use in the onDraw method.
marginPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
marginPaint.setColor(myResources.getColor(R.color.notepad_margin));
linePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
linePaint.setColor(myResources.getColor(R.color.notepad_lines));
// Get the paper background color and the margin width.
paperColor = myResources.getColor(R.color.notepad_paper);
margin = myResources.getDimension(R.dimen.notepad_margin);
}
```

5. To draw the paper, override onDraw, and draw the image using the Paint objects you created in Step 4. Once you've drawn the paper image, call the superclass's onDraw method, and let it draw the text as usual.

```
@Override
public void onDraw(Canvas canvas) {
// Color as paper
canvas.drawColor(paperColor);
// Draw ruled lines
canvas.drawLine(0, 0, getMeasuredHeight(), 0, linePaint);
canvas.drawLine(0, getMeasuredHeight(),
getMeasuredWidth(), getMeasuredHeight(),
linePaint);
// Draw margin
canvas.drawLine(margin, 0, margin, getMeasuredHeight(), marginPaint);
// Move the text across from the margin
canvas.save();
canvas.translate(margin, 0);
// Use the TextView to render the text.
super.onDraw(canvas);
canvas.restore();
}
```

6. That completes the TodoListItemView implementation. To use it in the To-Do List Activity, you need to include it in a new layout and pass that in to the Array Adapter constructor.

Start by creating a new todolist_item.xml resource in the res/layout folder. It will specify how each of the to-do list items is displayed. For this example, your layout need only consist of the new TodoListItemView, set to fi ll the entire available area.

```
<?xml version="1.0" encoding="utf-8"?>
<com.paad.todolist.TodoListItemView
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:padding="10dp"
android:scrollbars="vertical"
android:textColor="@color/notepad_text"
android:fadingEdge="vertical"
/>
```

7. Now open the ToDoList Activity class. The fi nal step is to change the parameters passed in to the ArrayAdapter in onCreate. Replace the reference to the default android.R.layout .simple_list_item_1 with the new R.layout.todolist_item layout created in Step 6.

```
final ArrayList<String> todoItems = new ArrayList<String>();
int resID = R.layout.todolist_item;
final ArrayAdapter<String> aa = new ArrayAdapter<String>(this, resID, todoItems);
myListView.setAdapter(aa);
```

Creating Compound Controls

Compound controls are atomic, reusable widgets that contain multiple child controls laid out and wired together.

When you create a compound control, you defi ne the layout, appearance, and interaction of the Views it contains. Compound controls are created by extending a ViewGroup (usually a Layout Manager). To create a new compound control, choose a layout class that's most suitable for positioning the child controls,
and extend it as shown in the skeleton code below:

```
public class MyCompoundView extends LinearLayout {
public MyCompoundView(Context context) {
super(context);
}
public MyCompoundView(Context context, AttributeSet attrs) {
super(context, attrs);
}
}
```

As with an Activity, the preferred way to design the UI for a compound control is to use a layout resource. The following code snippet shows the XML layout defi nition for a simple widget consisting of an Edit Text box and a button to clear it:

```
<?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">
<EditText
android:id="@+id/editText"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
/>
<Button
android:id="@+id/clearButton"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Clear"
/>
</LinearLayout>
```

To use this layout for your new widget, override its constructor to infl ate the layout resource using the inflate method from the LayoutInflate system service. The inflate method takes the layout resource and returns an infl ated View. For circumstances such as this where the returned View should be the class you're creating, you can pass in a parent and attach the result to it automatically, as shown in the next code sample.

The following code snippet shows the ClearableEditText class. Within the constructor it infl ates the layout resource created above and gets references to each of the Views it contains. It also makes a call to hookupButton that will be used to hookup the *clear text* functionality when the button is pressed.

```
public class ClearableEditText extends LinearLayout {
EditText editText;
Button clearButton;
public ClearableEditText(Context context) {
super(context);
// Inflate the view from the layout resource.
String infService = Context.LAYOUT_INFLATER_SERVICE;
LayoutInflater li;
li = (LayoutInflater)getContext().getSystemService(infService);
li.inflate(R.layout.clearable_edit_text, this, true);
// Get references to the child controls.
editText = (EditText)findViewById(R.id.editText);
clearButton = (Button)findViewById(R.id.clearButton);
// Hook up the functionality
hookupButton();
}
}
```

If you'd prefer to construct your layout in code, you can do so just as you would for an Activity. The following code snippet shows the ClearableEditText constructor overridden to create the same UI as is defi ned in the XML used in the earlier example:

```
public ClearableEditText(Context context) {
super(context);
// Set orientation of layout to vertical
setOrientation(LinearLayout.VERTICAL);
// Create the child controls.
editText = new EditText(getContext());
clearButton = new Button(getContext());
clearButton.setText("Clear");
// Lay them out in the compound control.
int lHeight = LayoutParams.WRAP_CONTENT;
int lWidth = LayoutParams.FILL_PARENT;
addView(editText, new LinearLayout.LayoutParams(lWidth, lHeight));
addView(clearButton, new LinearLayout.LayoutParams(lWidth, lHeight));
// Hook up the functionality
hookupButton();
}
```

Once the screen has been constructed, you can hook up the event handlers for each child control to provide the functionality you need. In this next snippet, the hookupButton method is fi lled in to clear the textbox when the button is pressed:

```
private void hookupButton() {
clearButton.setOnClickListener(new Button.OnClickListener() {
public void onClick(View v) {
editText.setText("");
}
});
}
```

Creating Custom Widgets and Controls

Creating completely new Views gives you the power to fundamentally shape the way your applications look and feel. By creating your own controls, you can create User Interfaces that are uniquely suited to your users' needs. To create new controls from a blank canvas, you extend either the View or SurfaceView classes.

The View class provides a Canvas object and a series of draw methods and Paint classes, to create a visual interface using raster graphics. You can then override user events like screen touches or key presses to provide interactivity. In situations where extremely rapid repaints and 3D graphics aren't required, the View base class offers a powerful lightweight solution.

The SurfaceView provides a canvas that supports drawing from a background thread and using openGL for 3D graphics. This is an excellent option for graphics-heavy controls that are frequently updated or display complex graphical information, particularly games and 3D visualizations.

This chapter introduces 2D controls based on the View class. To learn more about the SurfaceView class and some of the more advanced Canvas paint features available in Android, see Chapter 11.