Creating a New Visual Interface

The base View class presents a distinctly empty 100 · 100 pixel square. To change the size of the control and display a more compelling visual interface, you need to override the onMeasure and onDraw methods, respectively.

Within onMeasure, the new View will calculate the height and width it will occupy given a set of boundary conditions. The onDraw method is where you draw on the Canvas to create the visual interface.

The following code snippet shows the skeleton code for a new View class, which will be examined further in the following sections:

```
public class MyView extends View {
// Constructor required for in-code creation
public MyView(Context context) {
super(context);
}
// Constructor required for inflation from resource file
public MyView (Context context, AttributeSet ats, int defaultStyle) {
super(context, ats, defaultStyle );
}
//Constructor required for inflation from resource file
public MyView (Context context, AttributeSet attrs) {
super(context, attrs);
}
@Override
protected void onMeasure(int wMeasureSpec, int hMeasureSpec) {
int measuredHeight = measureHeight(hMeasureSpec);

int measuredWidth = measureWidth(wMeasureSpec);
// MUST make this call to setMeasuredDimension
// or you will cause a runtime exception when
// the control is laid out.
setMeasuredDimension(measuredHeight, measuredWidth);
}
private int measureHeight(int measureSpec) {
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
[ ... Calculate the view height ... ]
return specSize;
}
private int measureWidth(int measureSpec) {
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
[ ... Calculate the view width ... ]
return specSize;
}
@Override
protected void onDraw(Canvas canvas) {
[ ... Draw your visual interface ... ]
}
}
```

Note that the onMeasure **method calls** setMeasuredDimension**; you must always call this method within your overridden** onMeasure **method or your control will throw an exception when the parent container attempts to lay it out.**

Drawing Your Control

The onDraw method is where the magic happens. If you're creating a new widget from scratch, it's because you want to create a completely new visual interface. The Canvas parameter available in the onDraw method is the surface you'll use to bring your imagination to life.

Android provides a variety of tools to help draw your design on the Canvas using various Paint objects. The Canvas class includes helper methods to draw primitive 2D objects including circles, lines, rectangles, text, and Drawables (images). It also supports transformations that let you rotate, translate (move), and scale (resize) the Canvas while you draw on it.

Used in combination with Drawables and the Paint class (which offer a variety of customizable fi lls and pens), the complexity and detail that your control can render are limited only by the size of the screen and the power of the processor rendering it.

One of the most important techniques for writing effi cient code in Android is to avoid repetitive creation and destruction of objects. Any object created in your onDraw **method will be created and destroyed every time the screen refreshes. Improve effi ciency by making as many of these objects (in particular, instances of** Paint **and** Drawable**) class-scoped and moving their creation into the constructor.**

The following code snippet shows how to override the onDraw method to display a simple text string in the center of the control:

```
@Override
protected void onDraw(Canvas canvas) {
// Get the size of the control based on the last call to onMeasure.
int height = getMeasuredHeight();
int width = getMeasuredWidth();
// Find the center
int px = width/2;
int py = height/2;
// Create the new paint brushes.
// NOTE: For efficiency this should be done in
// the widget's constructor
Paint mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
mTextPaint.setColor(Color.WHITE);
// Define the string.
String displayText = "Hello World!";
// Measure the width of the text string.
float textWidth = mTextPaint.measureText(displayText);
// Draw the text string in the center of the control.
canvas.drawText(displayText, px-textWidth/2, py, mTextPaint);
}
```

Android offers a rich drawing library for the Canvas. Rather than diverge too far from the current topic, a more detailed look at the techniques available for drawing more complex visuals is included in Chapter 11.

Android does not currently support vector graphics. As a result, changes to any element of your canvas require repainting the entire canvas; modifying the color of a brush will not change your View's display until it is invalidated and redrawn. Alternatively, you can use OpenGL to render graphics; for more details, see the discussion on SurfaceView *in Chapter 11.*

Sizing Your Control

Unless you conveniently require a control that always occupies 100 · 100 pixels, you will also need to override onMeasure. The onMeasure method is called when the control's parent is laying out its child controls. It asks the question, "How much space will you use?" and passes in two parameters — widthMeasureSpec and heightMeasureSpec. They specify the space available for the control and some metadata describing that space.

Rather than return a result, you pass the View's height and width into the setMeasuredDimension method.

The following skeleton code shows how to override onMeasure. Note the calls to local method stubs calculateHeight and calculateWidth. They'll be used to decode the widthHeightSpec and heightMeasureSpec values and calculate the preferred height and width values.

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
int measuredHeight = measureHeight(heightMeasureSpec);
int measuredWidth = measureWidth(widthMeasureSpec);
setMeasuredDimension(measuredHeight, measuredWidth);
```

```
}
private int measureHeight(int measureSpec) {
// Return measured widget height.
}
private int measureWidth(int measureSpec) {
// Return measured widget width.
}
```

The boundary parameters, widthMeasureSpec and heightMeasureSpec, are passed in as integers for effi ciency reasons. Before they can be used, they fi rst need to be decoded using the static getMode and getSize methods from the MeasureSpec class, as shown in the snippet below:

```
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
```

Depending on the *mode* value, the *size* represents either the maximum space available for the control (in the case of AT_MOST), or the exact size that your control will occupy (for EXACTLY). In the case of UNSPECIFIED, your control does not have any reference for what the size represents.

By marking a measurement size as EXACT, the parent is insisting that the View will be placed into an area of the exact size specifi ed. The AT_MOST designator says the parent is asking what size the View would like to occupy, given an upper bound. In many cases, the value you return will be the same.

In either case, you should treat these limits as absolute. In some circumstances, it may still be appropriate to return a measurement outside these limits, in which case you can let the parent choose how to deal with the oversized View, using techniques such as clipping and scrolling.

The following skeleton code shows a typical implementation for handling View measurement:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
int measuredHeight = measureHeight(heightMeasureSpec);
int measuredWidth = measureWidth(widthMeasureSpec);
setMeasuredDimension(measuredHeight, measuredWidth);
}
private int measureHeight(int measureSpec) {
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
int result = 500; // Default size if no limits are specified.
if (specMode == MeasureSpec.AT_MOST) {
// Calculate the ideal size of your
// control within this maximum size.
// If your control fills the available // space return the outer bound.
result = specSize;
} else if (specMode == MeasureSpec.EXACTLY) {
// If your control can fit within these bounds return that value.
result = specSize;
}
return result;
}
private int measureWidth(int measureSpec) {
int specMode = MeasureSpec.getMode(measureSpec);
int specSize = MeasureSpec.getSize(measureSpec);
// Default size if no limits are specified.
int result = 500;
if (specMode == MeasureSpec.AT_MOST) {
// Calculate the ideal size of your control
// within this maximum size.
// If your control fills the available space
// return the outer bound.
result = specSize;
} else if (specMode == MeasureSpec.EXACTLY) {
// If your control can fit within these bounds return that value.
result = specSize;
}
return result;
}
```