



Las Clases Canvas y Paint

CONCEPTOS

La jerarquía de clases de Canvas:

```
public class Canvas extends Object
java.lang.Object -> android.graphics.Canvas
```

La jerarquía de clases de Paint:

```
public class Paint extends Object
java.lang.Object -> android.graphics.Paint
```

Constructores públicos de Canvas:

Canvas()	Construye un Canvas vacío.
Canvas(Bitmap bitmap)	Construye un Canvas con un bitmap específico.

Constructores públicos de Paint:

Paint()	Crea un nuevo Paint con configuración predeterminada.
Paint(int flags)	Crea un nuevo Paint con banderas específicas.
Paint(Paint paint)	Crea un nuevo Paint, inicializado con los atributos del parámetro paint.

Sobrescribir a onDraw ()

El paso más importante en la elaboración de una vista personalizada es reemplazar el método onDraw(). El parámetro de onDraw() es un objeto Canvas que el View puede utilizar para dibujarse. La clase Canvas define métodos para el dibujo de texto, líneas, mapas de bits, y muchas otras primitivas gráficas. Se pueden utilizar estos métodos en onDraw() para crear la interfaz de usuario personalizada (IU). Antes de llamar a cualquier método de dibujo, es necesario crear un objeto Paint.

Crear objetos de dibujo

El ambiente android.graphics divide el dibujo en dos áreas:

- Qué dibujar, a cargo del Canvas.
- Cómo dibujar, a cargo de Paint.

Por tanto, el Canvas proporciona un método para trazar una línea, mientras que Paint proporciona métodos para definir el color de esa línea. El Canvas tiene un método para dibujar un rectángulo, mientras que Paint define si rellena ese rectángulo con un color o dejarlo vacío. El Canvas define las formas que se pueden dibujar en la pantalla, mientras que Paint define el color, estilo, fuente, y así sucesivamente para cada figura que se dibuja.

Por lo tanto, antes de dibujar, se necesita para crear uno o más objetos Paint. Por ejemplo, el siguiente código PieChart hace esto en un método llamado init, que se llama desde el constructor:

```
private void init() {
    mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mTextPaint.setColor(mTextColor);
    if (mTextHeight == 0) {
        mTextHeight = mTextPaint.getTextSize();
    } else {
        mTextPaint.setTextSize(mTextHeight);
    }

    mPiePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
    mPiePaint.setStyle(Paint.Style.FILL);
    mPiePaint.setTextSize(mTextHeight);
}
```



```
mShadowPaint = new Paint(0);  
mShadowPaint.setColor(0xff101010);  
mShadowPaint.setMaskFilter(new BlurMaskFilter(8, BlurMaskFilter.Blur.NORMAL));  
:
```

La creación de objetos antes de utilizarlo permite una optimización importante. Los Views se redibujan con mucha frecuencia, y requieren inicialización intensiva. La creación de objetos de dibujo dentro del método `onDraw()` reduce significativamente el desempeño y puede hacer que la interfaz de usuario parezca lenta.

Manejo de Eventos

Para elaborar adecuadamente la vista personalizada, se necesita conocer su tamaño. Las vistas personalizadas complejas a menudo necesitan realizar múltiples cálculos de diseño, dependiendo del tamaño y la forma de su área de la pantalla. Nunca hacer suposiciones sobre el tamaño de la vista en la pantalla. Incluso si sólo una aplicación utiliza el View, esa aplicación debe manejar diferentes tamaños de pantalla, múltiples densidades de pantalla y diferentes relaciones de aspecto, tanto en modo vertical y horizontal.

A pesar de que View tiene muchos métodos para manejar medidas, la mayoría de ellos no necesita sobrescribirse. Si el View no necesita un control especial de su tamaño, sólo es necesario sobrescribir su método `onSizeChanged()`.

El método `onSizeChanged()` se invoca cuando a la vista se le asigna un tamaño por primera vez, y otra vez si el tamaño de la vista cambia por alguna razón. Calcular posiciones, dimensiones, y otros valores relacionados con el tamaño del View en `onSizeChanged()`, en lugar de volver a calcularlos cada vez que se dibujan. En el ejemplo `PieChart`, en el `onSizeChanged()` es donde el View `PieChart` calcula el rectángulo delimitador del gráfico circular y la posición relativa de la etiqueta de texto y otros elementos visuales.

Cuando al View se le asigna un tamaño, el controlador de la plantilla asume que el tamaño incluye el espaciado del View. Se deben manejar los valores de espaciado cuando se calcula el tamaño del View. El siguiente es un fragmento de `PieChart.onSizeChanged()` que muestra su uso:

```
// Account for padding  
float xpad = (float)(getPaddingLeft() + getPaddingRight());  
float ypad = (float)(getPaddingTop() + getPaddingBottom());  
if (mShowText) xpad += mTextWidth; // Account for the label  
float ww = (float)w - xpad;  
float hh = (float)h - ypad;  
float diameter = Math.min(ww, hh); // Figure out how big we can make the pie.
```

Si necesita un control más preciso sobre los parámetros de la plantilla del View, se implanta `onMeasure()`. Los parámetros del método son los valores `View.MeasureSpec` que el padre del View indica que tan grande sea, y que ese tamaño sea un máximo forzado o sólo una sugerencia. Como una optimización, estos valores se almacenan como números enteros, y se utilizan los métodos estáticos de `View.MeasureSpec` para descomprimir la información almacenada en cada entero.

Enseguida se muestra un ejemplo de la implantación de `onMeasure()`. Aquí, `PieChart` trata de que su área sea lo suficientemente grande para hacer que el pastel sea tan grande como su etiqueta:

```
@Override  
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
    // Try for a width based on our minimum  
    int minw = getPaddingLeft() + getPaddingRight() + getSuggestedMinimumWidth();  
    int w = resolveSizeAndState(minw, widthMeasureSpec, 1);  
  
    // Whatever the width ends up being, ask for a height that would let the pie  
    // get as big as it can
```



```

    int minh = MeasureSpec.getSize(w) - (int)mTextWidth + getPaddingBottom() +
getPaddingTop();
    int h = resolveSizeAndState(MeasureSpec.getSize(w) - (int)mTextWidth,
heightMeasureSpec, 0);

    setMeasuredDimension(w, h);
}

```

Hay tres cosas importantes a tener en cuenta en este código:

- Los cálculos tienen en cuenta el espaciado del View. Ello es responsabilidad del View.
- El método `resolveSizeAndState()` se utiliza para crear los valores finales de ancho y alto. Devuelve un valor `View.MeasureSpec` apropiada comparando el tamaño deseado del View con la especificación pasado a `onMeasure()`.
- El método `onMeasure()` no devuelve algún valor. En lugar de ello, el método comunica sus resultados llamando a `setMeasuredDimension()`. La invocación de este método es obligatoria. Si se omite esta llamada, la clase View lanza una excepción de tiempo de ejecución.

El Dibujo

Una vez creados los objetos y definido el código de medición, se implanta `onDraw()`. Cada View implanta a `onDraw()` de forma diferente, pero hay algunas operaciones comunes que la mayoría de Views comparten:

- Dibujar texto usando `drawText()`. Especificar el tipo de letra con `setTypeface()` y el color del texto con `setColor()`.
- Dibujar formas primitivas utilizando `drawRect()`, `drawOval()`, y `drawArc()`. Con `setStyle()` se cambian las formas si se rellenan, perfilan, o ambos.
- Dibujar formas más complejas usando la clase `Path`. Definir una forma agregando líneas y curvas a un objeto `Path`, y a continuación, dibujar la forma usando `drawPath()`. Al igual que con las formas primitivas, las rutas pueden perfilarse, rellenarse, o ambos dependiendo de `setStyle()`.
- Definir rellenos degradados creando objetos `LinearGradient`. Invocar a `setShader()` para utilizar el `LinearGradient` en formas rellenas.
- Dibujar mapas de bits utilizando `drawBitmap()`.

Por ejemplo, enseguida se muestra el código que dibuja el PieChart. Se utiliza una mezcla de texto, líneas y formas.

```

protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawOval(mShadowBounds, mShadowPaint); // Draw the shadow and the label text
    canvas.drawText(mData.get(mCurrentItem).mLabel, mTextX, mTextY, mTextPaint);
    for (int i = 0; i < mData.size(); ++i) { // Draw the pie slices
        Item it = mData.get(i);
        mPiePaint.setShader(it.mShader);
        canvas.drawArc(mBounds, 360-it.mEndAngle, it.mEndAngle - it.mStartAngle, true,
mPiePaint);
    }
    canvas.drawLine(mTextX, mPointerY, mPointerX, mPointerY, mTextPaint);
    canvas.drawCircle(mPointerX, mPointerY, mPointerSize, mTextPaint);
}

```

DESARROLLO

EJEMPLO 1.

En el siguiente ejemplo se emplea un Canvas para dibujar los ejes coordenados y calcular un origen relativo (0, 0):

1. Crear un nuevo proyecto Lienzo. En la carpeta `java/com.example.mipaquete`, abrir y modificar el archivo predeterminado `MainActivity.java` con el siguiente código:



```
import android.os.Bundle;
import android.app.Activity;
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Lienzo l = new Lienzo(this);
        setContentView(l);
    }
}
```

2. En la carpeta `java/com.example.mipaquete`, crear un nuevo archivo java `Lienzo.java` y agregar el siguiente código:

```
import android.content.*;
import android.graphics.*;
import android.view.View;
public class Lienzo extends View{
    Paint    p;
    int      x, y;
    public Lienzo(Context c){
        super(c);
    }
    protected void onDraw(Canvas c){
        super.onDraw(c);           // Canvas pinta atributos
        p = new Paint();           // Paint asigna atributos
        x = c.getWidth();           // También: getMeasuredWidth() o getRight(), x=480
        y = c.getHeight();          // También: getMeasuredHeight() o getBottom(), y=762
        p.setColor(Color.WHITE);    // Fondo blanco
        c.drawPaint(p);
        p.setColor(Color.BLACK);    // Texto negro
        p.setTextSize(20);
        c.drawText("x0=" + x/2 + ", y0=" + y/2, x/2 + 20, y/2-20, p);
        p.setColor(Color.rgb(0, 0, 255)); // Ejes azules
        c.drawLine(x/2, 0, x/2, y, p);
        c.drawLine(0, y/2, x, y/2, p);
    }
}
```

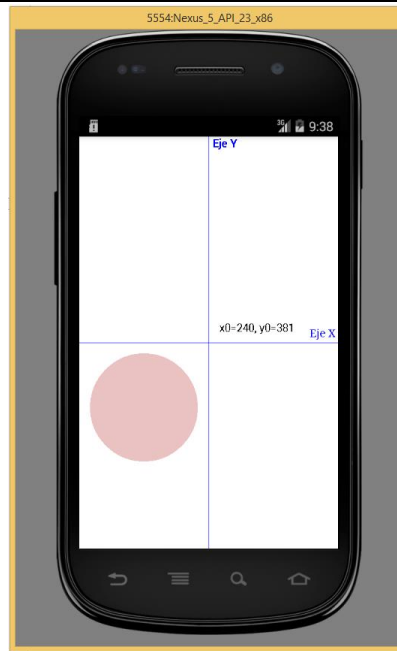
3. En el archivo anterior `Lienzo.java`, agregar las siguientes líneas al final del método `onDraw()`:

```
p.setTextAlign(Align.RIGHT);
p.setTypeface(Typeface.SERIF);
c.drawText("Eje X", x-5, y/2-10, p);
p.setTextAlign(Align.CENTER);
p.setTypeface(Typeface.DEFAULT_BOLD);
c.drawText("Eje Y", x/2+30, 20, p);
```

4. En el archivo anterior `Lienzo.java`, agregar las siguientes líneas al final del método `onDraw()` y variar el parámetro `alpha` del método `argb()` con diferentes valores:

```
p.setColor(Color.argb(100, 200, 100, 100));
c.drawCircle(x/2-120, y/2+120, 100, p);
```

5. Ejecutar la aplicación. La imagen debe ser similar a la siguiente:



6. En el fragmento de código anterior modificar lo siguiente:
 - a. Modificar los parámetros del método `setTextAlign()` con la alineación `Align.LEFT`.
 - b. Modificar los parámetros del método `setTextAlign()` con la alineación `ValueOf(String s)`.
 - c. Modificar los parámetros del método `setTypeface()` con las fuentes `DEFAULT`, `SANS_SERIF` y `MONOSPACE`.
7. En el archivo anterior, en el centro de cada cuadrante dibujar una de las figuras siguientes con sus métodos correspondientes: `drawOval()`, `drawRect()`, `drawRoundRect()`, `drawArc()`.
8. Mostrar la imagen correspondiente a los incisos 6 y 7.

EJEMPLO 2.

En el siguiente ejemplo se dibujan las funciones seno y coseno:

1. Crear un nuevo proyecto Lienzo2. En la carpeta `java/com.example.mipaquete`, abrir y modificar el archivo predeterminado `MainActivity.java` con el siguiente código:

```
import android.content.*;
import android.graphics.*;
import android.view.View;
public class Lienzo extends View{
    Paint    p;
    Path     r;
    int      x, y, x0, y0;
    public Lienzo(Context c){
        super(c);
    }
    protected void onDraw(Canvas c){
        super.onDraw(c);          // Canvas pinta atributos
        p = new Paint();          // Paint asigna atributos
        r = new Path();
        x = c.getWidth();         x0=x/2; // También: getMeasuredWidth(), o getRight(), x=480
        y = c.getHeight();        y0=y/2; // También: getMeasuredHeight(), o getBottom(), y=762
```

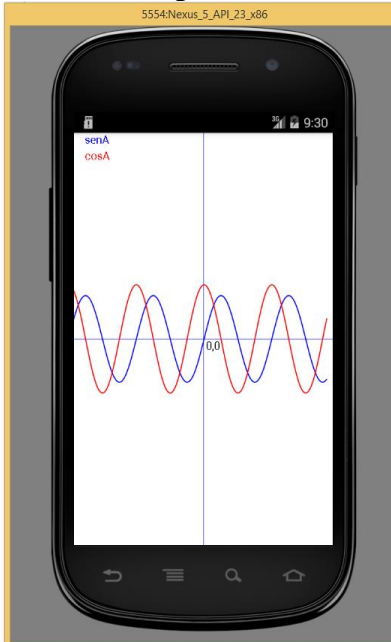


```
p.setColor(Color.WHITE);    // Fondo blanco
c.drawPaint(p);
p.setColor(Color.BLACK);    // Texto negro
p.setTextSize(20);
c.drawText("0,0", x0 + 5, y0 + 20, p);
p.setColor(Color.rgb(0, 0, 255)); // Ejes azules
c.drawLine(x0, 0, x0, y, p);
c.drawLine(0, y0, x, y0, p);
p.setColor(Color.BLUE);    c.drawText("senA", 20, 20, p);
p.setColor(Color.RED);    c.drawText("cosA", 20, 50, p);
p.setStyle(Paint.Style.STROKE);
p.setStrokeWidth(2);
p.setAntiAlias(true);

r = new Path();
x = getMeasuredWidth();
r.moveTo(0, 0); p.setColor(Color.BLUE);
for(int i=1; i<x; i++) r.lineTo(i, (float) Math.sin(i / 20f) * (-80f));
r.offset(-10, y0);        c.drawPath(r, p);

r = new Path();
r.moveTo(0, 0); p.setColor(Color.RED);
for(int i=1; i<x; i++) r.lineTo(i, (float) Math.cos(i / 20f) * (-100f));
r.offset(-10, y0);
c.drawPath(r, p);
}
}
```

2. Ejecutar la aplicación. La imagen debe ser similar a la siguiente:



3. Modificar la amplitud y el periodo de las dos señales, seno y coseno, del inciso 1.
4. Acotar las magnitudes de la amplitud y periodo, en los ejes coordenados del inciso 3.
5. Mostrar la imagen correspondiente a los incisos 3 y 4.