Personalización de componentes

Índice

En esta sesión veremos cómo crear vistas personalizadas para nuestras aplicaciones Android. En primer lugar trataremos el tema de los componentes compuestos, que nos permitirán crear nuevas vistas a partir de la agrupación de vistas ya existentes. A continuación presentaremos los componentes propios, un mecanismo mediante el cual podremos crear nuestras propias vistas desde cero. Por último, mostraremos de manera resumida cómo extender la funcionalidad de las vistas ya existentes, como por ejemplo los `TextView`, para conseguir disponer de nuevos controles en nuestra aplicación sin tener que empezar desde cero.

1. Componentes compuestos

Un componente compuesto está formado por un conjunto de vistas hijas que se tratan de manera atómica, como si fueran una única vista. Cuando se crea un componente compuesto se deben definir los siguientes elementos: el layout o disposición, la apariencia y la interacción con cada una de las vistas que contiene.

1.1. Crear un componente compuesto

Para crear un componente compuesto definimos una subclase de algún `ViewGroup` (normalmente de un objeto de tipo *layout*). Escogeremos una clase de tipo *layout* que se adapte más a la forma en la que queremos disponer las vistas de nuestro nuevo componente. El esqueleto de la subclase creada será el siguiente:

```
public class MiComponente extends LinearLayout {
        public MiComponente(Context context) {
                super(context);
        }

        public MiComponente(Context context, AttributeSet atts) {
                super(context, atts);
        }
}
```

Como en el caso de las actividades, la mejor forma de definir la disposición de las vistas de un componente compuesto es mediante un archivo XML de *layout* en los recursos de la aplicación. En el siguiente listado se muestra un ejemplo de componente compuesto formado por un campo de edición de texto y un botón al que más tarde le proporcionaremos la funcionalidad de eliminar el texto contenido en el campo de edición:

```
<?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/button"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Borrar"
        />
</LinearLayout>
```

Para usar este layout en nuestra nueva vista, sobrecargamos su constructor para que se inicialice su interfaz gráfica mediante el método `inflate` perteneciente al servicio de Android `LayoutInflate`, pasando como parámetro el identificador de recurso correspondiente al *layout*. En el siguiente código mostramos la clase `EdicionBorrable`, que se basa en el *layout* definido anteriormente.

```
public class EdicionBorrable extends LinearLayout {

        EditText editText;
        Button button;

        public EdicionBorrable(Context context) {
                super(context);

                // Creamos la interfaz a partir del layout
                String infService = Context.LAYOUT_INFLATER_SERVICE;
                LayoutInflater li;
                li =
(LayoutInflater)getContext().getSystemService(infService);
                li.inflate(R.layout.edicionborrable, this, true);

                // Obtenemos las referencias a las vistas hijas
                editText = (EditText)findViewById(R.id.editText);
                button = (Button)findViewById(R.id.button);
        }
}
```

1.2. Definir el componente compuesto mediante código

También es posible, como en el caso de las actividades, definir el *layout* de un componente compuesto mediante código. En el siguiente código mostramos cómo definir el *layout* de la clase `EdicionBorrable` sin utilizar un archivo XML:

```
public EdicionBorrable(Context context) {
        super(context);

        // Cambiamos la orientación del layout a vertical
        setOrientation(LinearLayout.VERTICAL);

        // Creamos las vistas hijas
        editText = new EditText(getContext());
        button = new Button(getContext());
        button.setText("Borrar");

        // Colocamos estas vistas en el control compuesto
        int lHeight = LayoutParams.WRAP_CONTENT;
        int lWidth = LayoutParams.FILL_PARENT;
        addView(editText, new LinearLayout.LayoutParams(lWidth, lHeight));
        addView(button, new LinearLayout.LayoutParams(lWidth, lHeight));
}
```

1.3. Añadir funcionalidad

Para añadir funcionalidad al componente compuesto debemos definir manejadores de eventos para sus vistas individuales. Esto se hará de la manera habitual. Por ejemplo, para hacer que en nuestro componente compuesto el botón borre el texto del cuadro de edición, lo único que tenemos que hacer es añadir el manejador adecuado al final del constructor, tras obtener la referencia a las vistas:

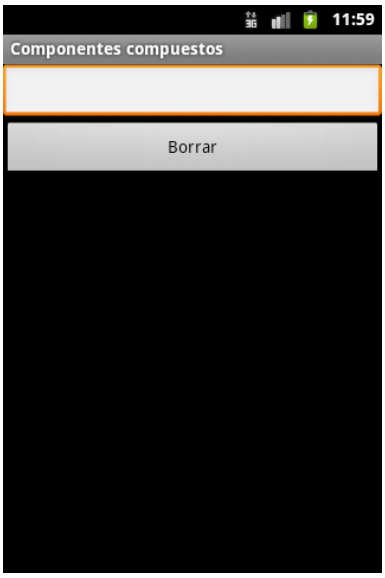
```
button.setOnClickListener(new Button.OnClickListener() {
        public void onClick(View v) {
                editText.setText("");
        }
});
```

1.4. Incluir el componente compuesto en nuestra actividad

Una vez definido el componente compuesto podemos añadirlo a la interfaz de nuestra actividad como si se tratara de una vista cualquiera. Por ejemplo, podemos añadir a nuestra aplicación una actividad cuya interfaz contendrá una única vista de tipo `EdicionBorrable`, con lo que su *layout* quedaría definido de la siguiente forma:

```
<?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"
    >
<es.ua.jtech.android.compuestos.EdicionBorrable
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
</LinearLayout>
```

Obsérvese que a la hora de añadir nuestro componente `EdicionBorrable` se ha incluido el espacio de nombres, que en nuestro caso concreto es `es.ua.jtech.android.compuestos`. Por lo demás, nuestro componente se tratará como una vista normal y corriente.



Nuestro ejemplo de componente compuesto

2. Componentes propios

Si no hay ningún componente predefinido que se adapte a nuestras necesidades, podemos crear un nuevo tipo de vista (`View`) en la que especificaremos exactamente qué es lo que queremos dibujar en la pantalla. El primer paso consistirá en crear una subclase de `View` en la que sobrescribiremos el método `onDraw`, que es el que define la forma en la que se dibuja el componente.

```
public class MiVista extends View {
        public MiVista(Context context) {
                super(context);
        }

        @Override
        protected void onDraw(Canvas canvas) {
                // TODO Definir como dibujar el componente
        }
}
```

2.1. Lienzo y pincel

El método `onDraw` recibe como parámetro el lienzo (`Canvas`) en el que deberemos dibujar. En este lienzo podremos dibujar diferentes tipos de elementos, como primitivas geométricas, texto e imágenes.

Importante

No confundir el `Canvas` de Android con el `Canvas` que existe en Java ME/SE. En Java ME/SE el `Canvas` es un componente de la interfaz, que equivaldría a `View` en Android, mientras que el `Canvas` de Android es más parecido al objeto `Graphics` de Java ME/SE, que encapsula el

contexto gráfico (o lienzo) del área en la que vamos a dibujar.

Además, para dibujar determinados tipos de elementos deberemos especificar también el tipo de pincel a utilizar (`Paint`), en el que especificaremos una serie de atributos como su color, grosor, etc.

Por ejemplo, para especificar un pincel que pinte en color rojo escribiremos lo siguiente:

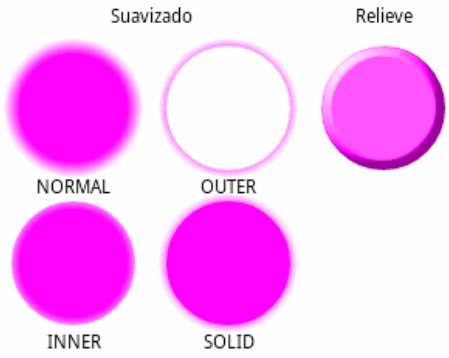
```
Paint p = new Paint();
p.setColor(Color.RED);
```

Las propiedades que podemos establecer en el pincel son:

- **Color plano**: Con `setARGB` o `setColor` se puede especificar el código ARGB del color o bien utilizar constantes con colores predefinidos de la clase `Color`.
- **Gradientes y shaders**: Se pueden rellenar las figuras utilizando shaders de gradiente o de bitmap. Para utilizar un *shader* tenemos el método `setShader`, y tenemos varios *shaders* disponibles, como distintos *shaders* de gradiente (`LinearShader`, `RadialShader`, `SweepShader`), `BitmapShader` para rellenar utilizando un mapa de bits como patrón, y `ComposeShader` para combinar dos *shaders* distintos.

Tipos de gradiente

- **Máscaras**: Nos sirven para aplicar un suavizado a los gráficos (`BlurMaskFilter`) o dar efecto de relieve (`EmbossMaskFilter`). Se aplican con `setMaskFilter`.

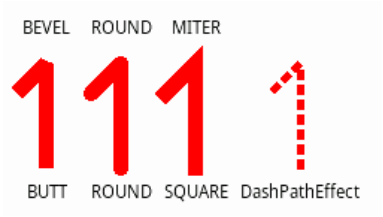


Mascaras de suavizado y relieve

- **Sombras**: Podemos crear efectos de sombra con `setShadowLayer`.
- **Filtros de color**: Aplica un filtro de color a los gráficos dibujados, alterando así su color original. Se aplica con `setColorFilter`.
- **Estilo de la figura**: Se puede especificar con `setStyle` que se dibuje sólo el trazo, sólo el relleno, o ambos.

Estilos de pincel

- **Estilo del trazo**: Podemos especificar el grosor del trazo (`setStrokeWidth`), el tipo de línea (`setPathEffect`), la forma de las uniones en las polilíneas (redondeada/`ROUND`, a inglete/`MITER`, o biselada/`BEVEL`, con `setStrokeJoin`), o la forma de las terminaciones (cuadrada/`SQUARE`, redonda/`ROUND` o recortada/`BUTT`, con `setStrokeCap`).



Tipos de trazo y límites

- **Antialiasing**: Podemos aplicar *antialiasing* con `setAntiAlias` a los gráficos para evitar el efecto *sierra*.
- **Dithering**: Si el dispositivo no puede mostrar los 16 millones de colores, en caso de haber un gradiente, para que el cambio de color no sea brusco, con esta opción (`setDither`) se mezclan pixels de diferentes colores para dar la sensación de que la transición entre colores es más suave.

Efecto dithering

- **Modo de transferencia**: Con `setXferMode` podemos cambiar el modo de transferencia con el que se dibuja. Por ejemplo, podemos hacer que sólo se dibuje encima de pixels que tengan un determinado color.
- **Estilo del texto**: Podemos también especificar el tipo de fuente a utilizar y sus atributos. Lo veremos con más detalle más adelante.

Una vez establecido el tipo de pincel, podremos utilizarlo para dibujar diferentes elementos en el lienzo, utilizando métodos de la clase `Canvas`.

En el lienzo podremos también establecer algunas propiedades, como el área de recorte (`clipRect`), que en este caso no tiene porque ser rectangular (`clipPath`), o transformaciones geométricas (`translate`, `scale`, `rotate`, `skew`, o `setMatrix`). Si queremos cambiar temporalmente estas propiedades, y luego volver a dejar el lienzo

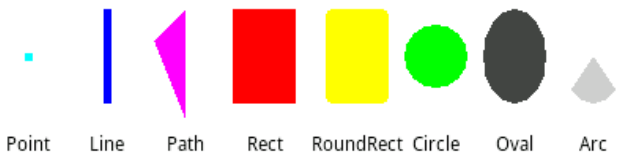
como estaba originalmente, podemos utilizar los métodos `save` y `restore`.

Vamos a ver a continuación como utilizar los métodos del lienzo para dibujar distintos tipos de primitivas geométricas.

2.2. Primitivas geométricas

En la clase `Canvas` encontramos métodos para dibujar diferentes tipos de primitivas geométricas. Estos tipos son:

- **Puntos**: Con `drawPoint` podemos dibujar un punto en las coordenadas X, Y especificadas.
- **Líneas**: Con `drawLine` dibujamos una línea recta desde un punto de origen hasta un punto destino.
- **Polilíneas**: Podemos dibujar una polilínea mediante `drawPath`. La polilínea se especificará mediante un objeto de clase `Path`, en el que iremos añadiendo los segmentos de los que se compone. Este objeto `Path` representa un contorno, que podemos crear no sólo a partir de segmentos rectos, sino también de curvas cuadráticas y cúbicas.
- **Rectángulos**: Con `drawRect` podemos dibujar un rectángulo con los límites superior, inferior, izquierdo y derecho especificados.
- **Rectángulos con bordes redondeados**: Es también posible dibujar un rectángulo con esquinas redondeadas con `drawRoundRect`. En este caso deberemos especificar también el radio de las esquinas.
- **Círculos**: Con `drawCircle` podemos dibujar un círculo dando su centro y su radio.
- **Óvalos**: Los óvalos son un caso más general que el del círculo, y los crearemos con `drawOval` proporcionando el rectángulo que lo engloba.
- **Arcos**: También podemos dibujar arcos, que consisten en un segmento del contorno de un óvalo. Se crean con `drawArc`, proporcionando, además de los mismos datos que en el caso del óvalo, los ángulos que limitan el arco.
- **Todo el lienzo**: Podemos también especificar que todo el lienzo se rellene de un color determinado con `drawColor` o `drawARGB`. Esto resulta útil para limpiar el fondo antes de empezar a dibujar.



Tipos de primitivas geométricas

A continuación mostramos un ejemplo de cómo podríamos dibujar una polilínea y un rectángulo:

```
Paint paint = new Paint();
paint.setStyle(Style.FILL);
paint.setStrokeWidth(5);
```

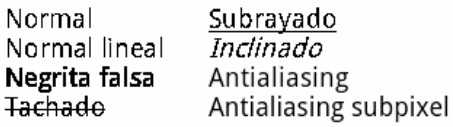
```
paint.setColor(Color.BLUE);

Path path = new Path();
path.moveTo(50, 130);
path.lineTo(50, 60);
path.lineTo(30, 80);

canvas.drawPath(path, paint);

canvas.drawRect(new RectF(180, 20, 220, 80), paint);
```

2.3. Cadenas de texto

Para dibujar texto podemos utilizar el método `drawText`. De forma alternativa, se puede utilizar `drawPosText` para mostrar texto especificando una por una la posición de cada carácter, y `drawTextOnPath` para dibujar el texto a lo largo de un contorno (`Path`).

Para especificar el tipo de fuente y sus atributos, utilizaremos las propiedades del objeto `Paint`. Las propiedades que podemos especificar del texto son:

- **Fuente**: Con `setTypeface` podemos especificar la fuente, que puede ser alguna de las fuentes predefinidas (*Sans Serif*, *Serif*, *Monoespaciada*), o bien una fuente propia a partir de un fichero de fuente. También podemos especificar si el estilo de la fuente será normal, cursiva, negrita, o negrita cursiva.
- **Tamaño**: Podemos establecer el tamaño del texto con `setTextSize`.
- **Anchura**: Con `setTextScaleX` podemos modificar la anchura del texto sin alterar la altura.
- **Inclinación**: Con `setTextSkewX` podemos aplicar un efecto de desencajado al texto, pudiendo establecer la inclinación que tendrán los carácteres.
- **Subrayado**: Con `setUnderlineText` podemos activar o desactivar el subrayado.
- **Tachado**: Con `setStrikeThruText` podemos activar o desactivar el efecto de tachado.
- **Negrita falsa**: Con `setFakeBoldText` podemos darle al texto un efecto de *negrita*, aunque la fuente no sea de este tipo.
- **Alineación**: Con `setTextAlign` podemos especificar si el texto de alinea al centro, a la derecha, o a la izquierda.
- **Subpixel**: Se renderiza a nivel de subpixel. El texto se genera a una resolución mayor que la de la pantalla donde lo vamos a mostrar, y para cada pixel real se habrán generado varios pixels. Si aplicamos *antialiasing*, a la hora de mostrar el pixel real, se determinará un nivel de gris dependiendo de cuantos pixels ficticios estén activados. Se consigue un aspecto de texto más suavizado.
- **Texto lineal**: Muestra el texto con sus dimensiones reales de forma lineal, sin ajustar los tamaños de los caracteres a la cuadrícula de pixels de la pantalla.
- **Contorno del texto**: Aunque esto no es una propiedad del texto, el objeto `Paint` también nos permite obtener el contorno (`Path`) de un texto dado, para así poder aplicar al texto los mismos efectos que a cualquier otro contorno que dibujemos.



Efectos del texto

Con esto hemos visto como dibujar texto en pantalla, pero para poderlo ubicar de forma correcta es importante saber el tamaño en pixels del texto a mostrar. Vamos a ver ahora cómo obtener estas métricas.

Las métricas se obtendrán a partir del objeto `Paint` en el que hemos definido las propiedades de la fuente a utilizar. Mediante `getFontMetrics` podemos obtener una serie de métricas de la fuente actual, que nos dan las distancias recomendadas que debemos dejar entre diferentes líneas de texto:

- `ascent`: Distancia que asciende la fuente desde la línea de base. Para texto con espaciado sencillo es la distancia que se recomienda dejar por encima del texto. Se trata de un valor negativo.
- `descent`: Distancia que baja la fuente desde la línea de base. Para texto con espaciado sencillo es la distancia que se recomienda dejar por debajo del texto. Se trata de un valor positivo.
- `leading`: Distancia que se recomienda dejar entre dos líneas consecutivas de texto.
- `bottom`: Es la máxima distancia que puede bajar un símbolo desde la línea de base. Es un valor positivo.
- `top`: Es la máxima distancia que puede subir un símbolo desde la línea de base. Es un valor negativo.

Los anteriores valores son métricas generales de la fuente, pero muchas veces necesitaremos saber la anchura de una determinada cadena de texto, que ya no sólo depende de la fuente sino también del texto. Tenemos una serie de métodos con los que obtener este tipo de información:

- `measureText`: Nos da la anchura en pixels de una cadena de texto con la fuente actual.
- `breakText`: Método útil para cortar el texto de forma que no se salga de los márgenes de la pantalla. Se le proporciona la anchura máxima que puede tener la línea, y el método nos dice cuántos carácteres de la cadena proporcionada caben en dicha línea.
- `getTextWidths`: Nos da la anchura individual de cada carácter del texto proporcionado.
- `getTextBounds`: Nos devuelve un rectángulo con las dimensiones del texto, tanto anchura como altura.

2.4. Imágenes

Podemos también dibujar en nuestro lienzo imágenes que hayamos cargado como `Bitmap`. Esto se hará utilizando el método `drawBitmap`.

También podremos realizar transformaciones geométricas en la imagen al mostrarla en el lienzo lienzo con `drawBitmap`, e incluso podemos dibujar el *bitmap* sobre una malla poligonal con `drawBitmapMesh`

2.5. Drawables

También podemos dibujar objetos de tipo *drawable* en nuestro lienzo, esta vez mediante el método `draw` definido en la clase `Drawable`. Esto nos permitirá mostrar en nuestro componente cualquiera de los tipos disponibles de *drawables*, tanto definidos en XML como de forma programática.

2.6. Medición del componente

Al crear un nuevo componente, además de sobreescribir el método `onDraw`, es buena idea sobreescribir también el método `onMeasure`. Este método será invocado por el sistema cuando vaya a ubicarlo en el *layout*, para asignarle un tamaño. Para cada dimensión (altura y anchura), nos pasa dos parámetros:

- **Tamaño**: Tamaño en píxeles solicitado para la dimensión (altura o anchura).
- **Modo**: Puede ser `EXACTLY`, `AT_MOST`, o `UNSPECIFIED`. En el primer caso indica que el componente debe tener exactamente el tamaño solicitado, el segundo indica que como mucho puede tener ese tamaño, y el tercero nos da libertad para decidir el tamaño.

Antes de finalizar `onMeasure`, deberemos llamar obligatoriamente a `setMeasuredDimension(width, height)` proporcionando el tamaño que queramos que tenga nuestro componente. Una posible implementación sería la siguiente:

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
  int widthMode = MeasureSpec.getMode(widthMeasureSpec);
  int widthSize = MeasureSpec.getSize(widthMeasureSpec);
  int heightMode = MeasureSpec.getMode(heightMeasureSpec);
  int heightSize = MeasureSpec.getSize(heightMeasureSpec);

  int width = DEFAULT_SIZE;
  int height = DEFAULT_SIZE;

  switch(widthMode) {
    case MeasureSpec.EXACTLY:
      width = widthSize;
      break;
    case MeasureSpec.AT_MOST:
      if(width > widthSize) {
        width = widthSize;
      }
      break;
  }

  switch(heightMode) {
    case MeasureSpec.EXACTLY:
      height = heightSize;
      break;
    case MeasureSpec.AT_MOST:
```

```
      if(height > heightSize) {
        height = heightSize;
      }
      break;
  }

  this.setMeasuredDimension(width, height);
}
```

Podemos ver que tenemos unas dimensiones preferidas por defecto para nuestro componente. Si nos piden unas dimensiones exactas, ponemos esas dimensiones, pero si nos piden unas dimensiones como máximo, nos quedamos con el mínimo entre nuestra dimensión preferida y la que se ha especificado como límite máximo que puede tener.

2.7. Atributos propios

Si creamos un nuevo tipo de vista, es muy probable que necesitemos parametrizarla de alguna forma. Por ejemplo, si queremos dibujar una gráfica que nos muestre un porcentaje, necesitaremos proporcionar un valor numérico como porcentaje a mostrar. Si vamos a crear la vista siempre de forma programática esto no es ningún problema, ya que basta con incluir en nuestra clase un método que establezca dicha propiedad.

Sin embargo, si queremos que nuestro componente se pueda añadir desde el XML, será necesario poder pasarle dicho valor como atributo. Para ello en primer lugar debemos declarar los atributos propios en un fichero */res/values/attrs.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <declare-styleable name="Grafica">
                <attr name="percentage" format="integer"/>
        </declare-styleable>
</resources>
```

En el XML donde definimos el *layout*, podemos especificar nuestro componente utilizando como nombre de la etiqueta el nombre completo (incluyendo el paquete) de la clase donde hemos definido la vista:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app=
"http://schemas.android.com/apk/res/es.ua.jtech.android.grafica"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<es.ua.jtech.grafica.android.GraficaView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:percentage="60"
 />
</LinearLayout>
```

Podemos fijarnos en que para declarar el atributo propio hemos tenido que especificar el espacio de nombres en el que se encuentra, mediante el atributo `xmlns:app` del `LinearLayout`. En dicho espacio de nombres deberemos especificar el paquete que

hemos declarado en `AndroidManifest.xml` para la aplicación (en nuestro caso `es.ua.jtech.android.grafica`).

En el siguiente código vemos cómo acceder al valor de un atributo para nuestro componente propio desde el código:

```
public GraficaView(Context context) {
  super(context);
}

public GraficaView(Context context, AttributeSet attrs, int defStyle) {
  super(context, attrs, defStyle);
  this.init(attrs);
}

public GraficaView(Context context, AttributeSet attrs) {
  super(context, attrs);
  this.init(attrs);
}

private void init(AttributeSet attrs) {
  TypedArray ta = this.getContext().obtainStyledAttributes(attrs,
                                                R.styleable.Grafica);
  this.percentage = ta.getInt(R.styleable.Grafica_percentage, 0);
}
```

En primer lugar podemos ver que debemos definir todos los posibles constructores de las vistas, ya que cuando se cree desde el XML se invocará uno de los que reciben la lista de atributos especificados. Una vez recibamos dicha lista de atributos, deberemos obtener el conjunto de atributos propios mediante `obtainStyledAttributes`, y posteriormente obtener los valores de cada atributo concreto dentro de dicho conjunto.

2.8. Actualización del contenido

Es posible que en un momento dado cambien los datos a mostrar y necesitemos actualizar el contenido que nuestro componente está dibujando en pantalla. Podemos forzar que se vuelva a dibujar llamando al método `invalidate` de nuestra vista (`View`).

Esto podemos utilizarlo también para crear animaciones. De hecho, para crear una animación simplemente deberemos cambiar el contenido del lienzo conforme pasa el tiempo. Una forma de hacer esto es simplemente cambiar mediante un hilo o temporizadores propiedades de los objetos de la escena (como sus posiciones), y forzar a que se vuelva a redibujar el contenido del lienzo cada cierto tiempo.

Sin embargo, si necesitamos contar con una elevada tasa de refresco, como por ejemplo en el caso de un videojuego, será recomendable utilizar una vista de tipo `SurfaceView`.

3. Modificar vistas existentes

Existe una alternativa más simple para crear vistas propias: si ya existe una vista en Android que se parezca al nuevo control que deseamos crear, podemos simplemente

extender la funcionalidad de dicho componente y modificar aquellas partes de su comportamiento que deseamos modificar mediante sobrecarga. De esta forma podemos ahorrar también mucho código.

3.1. Extendiendo la funcionalidad de un TextView

El primer paso consistirá en crear una clase que herede de la clase de la vista cuya funcionalidad queremos extender. En el siguiente ejemplo mostramos el esqueleto de una clase llamada `MiTextView` y que heredará de `TextView`:

```
public class MiTextView extends TextView {

        public MiTextView (Context context, AttributeSet attrs, int
defStyle) {
                super(context, attrs, defStyle);
        }

        public MiTextView (Context context) {
                super(context);
        }

        public MiTextView (Context context, AttributeSet attrs) {
                super(context, attrs);
        }
}
```

El siguiente paso consistiría en sobrecargar aquellos manejadores de evento cuyo comportamiento queremos que difiera del de la clase base. El siguiente código amplia la clase `MiTextView` mostrada en el ejemplo anterior, añadiendo la sobrecarga de el manejador `onDraw`, lo cual permitirá modificar la forma en la que la vista se mostrará en la pantalla, y sobrecargando también el manejador `onKeyDown`:

```
public class MiTextView extends TextView {

        public MiTextView (Context context, AttributeSet ats, int
defStyle) {
                super(context, ats, defStyle);
        }

        public MiTextView (Context context) {
                super(context);
        }

        public MiTextView (Context context, AttributeSet attrs) {
                super(context, attrs);
        }

        @Override
        public void onDraw(Canvas canvas) {
                // Primero dibujamos en el canvas bajo el texto...

                // ... luego mostramos el texto de la manera habitual
                // haciendo uso de la clase base...
                super.onDraw(canvas);

                // ... y por último dibujamos cosas sobre el texto
        }
```

```
        @Override
        public boolean onKeyDown(int keyCode, KeyEvent keyEvent) {
                // Primero realizamos las acciones que sean oportunas
                // según la tecla pulsada...

                // ...y a continuación hacemos también uso de la clase
base
                return super.onKeyDown(keyCode, keyEvent);
        }
}
```