Drawables, estilos y temas

Índice

En esta sesión trataremos varias herramientas de Android que nos permitirán personalizar la interfaz de nuestras aplicaciones. En primer lugar hablaremos de los *Drawables*, recursos de tipo imagen que pueden ser usados de varias formas distintas: para mostrarlos tal cual en una actividad, para definir el fondo de una determinada vista, etc. A cotinuación hablaremos de estilos y temas en Android. Los estilos y temas siguen la misma filosofía que el lenguaje CSS en el caso del desarrollo web: separar el diseño del contenido. De esta forma seríamos capaces de modificar el aspecto de un conjunto de vistas o el aspecto global de la aplicación sin necesidad de modificar ningún archivo de código o de layout.

1. Elementos drawables

Un *drawable* es un tipo de recurso que puede ser dibujado en pantalla. Podremos utilizarlos para especificar el aspecto que van a tener los diferentes componentes de la interfaz, o partes de éstos. Estos *drawables* podrán ser definidos en XML o de forma programática. Entre los diferentes tipos de *drawables* existentes encontramos:

- **Color**: Rellena el lienzo de un determinado color.
- **Gradiente**: Rellena el lienzo con un gradiente.
- **Forma** (*shape*): Se pueden definir una serie de primitivas geométricas básicas como *drawables*.
- **Imágen** (*bitmap*): Una imagen se comporta como *drawable*, ya que podrá ser dibujada y referenciada de la misma forma que el resto.
- **Nine-patch**: Tipo especial de imagen PNG que al ser escalada sólo se escala su parte central, pero no su marco.
- **Animación**: Define una animación por fotogramas, como veremos más adelante.
- **Capa** (*layer list*): Es un *drawable* que contiene otros *drawables*. Cada uno especificará la posición en la que se ubica dentro de la capa.
- **Estados** (*state list*): Este *drawable* puede mostrar diferentes contenidos (que a su vez son *drawables*) según el estado en el que se encuentre. Por ejemplo sirve para definir un botón, que se mostrará de forma distinta según si está normal, presionado, o inhabilitado.
- **Niveles** (*level list*): Similar al anterior, pero en este caso cada *item* tiene asignado un valor numérico (nivel). Al establecer el nivel del *drawable* se mostrará el *item* cuyo nivel sea mayor o igual que el indicado.
- **Transición** (*transition*): Nos permite mostrar una transición de un *drawable* a otro mediante un fundido.
- **Inserción** (*inset*): Ubica un *drawable* dentro de otro, en la posición especificada.
- **Recorte** (*clip*): Realiza un recorte de un *drawable*.
- **Escala** (*scale*): Cambia el tamaño de un *drawable*.

Todos los *drawables* derivan de la clase `Drawable`. Esta nos permite que todos ellos puedan ser utilizados de la misma forma, independientemente del tipo del que se trate. Se puede consultar la lista completa de *drawables* y su especificación en la siguiente

dirección:

http://developer.android.com/guide/topics/resources/drawable-resource.html

Por ejemplo, vamos a definir un *drawable* que muestre un rectángulo rojo con borde azul, creando un fichero XML de nombre `rectangulo.xml` en el directorio */res/drawable/*. El fichero puede tener el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?> <shape
xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="rectangle">
         <solid android:color="#ff0000"/>
         <stroke android:width="2dp" android:color="#0000ff"
            android:dashWidth="10dp" android:dashGap="5dp"/>
</shape>
```

Podremos hacer referencia a este rectángulo desde el código mediante `R.drawable.rectangulo` y mostrarlo en la interfaz asignándolo a un componente de alto nivel como por ejemplo `ImageView`, o bien hacer referencia a él desde un atributo del XML mediante `@drawable/rectangulo`. Por ejemplo, podríamos especificar este *drawable* en el atributo `android:background` de la etiqueta `Button` dentro de nuestro *layout*, para que así el botón pase a tener como aspecto una forma rectangular de color rojo y con borde azul. De la misma forma podríamos darle al botón el aspecto de cualquier otro tipo de *drawable* de los vistos anteriormente. A continuación vamos a ver con más detalle los tipos de *drawables* más interesantes.

Nota

Como en el caso de otros recursos, podemos definir diferentes variantes del directorio de *drawables*, para así definir diferentes versiones de los *drawables* para los distintos tipos de dispositivos, por ejemplo según su densidad, tamaño, o forma de pantalla.

1.1. Colores

Los *ColorDrawables* son los *drawables* más sencilllos. Permiten definir una imagen simplemente a partir de un único color sólido. Se definen usando la etiqueta `color` en XML, dentro de la carpeta de *drawables* en los recursos de la aplicación. A continuación mostramos un ejemplo de drawable de color rojo

```
<color
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:color="#FF0000"
/>
```

1.2. Formas

Ya hemos visto anteriormente un ejemplo de *drawable* de tipo forma, pero en esta sección examineramos este tipo de elementos en más detalle. Mediante el elemento `shape` podremos definir formas simples indicando sus dimensiones, fondo y bordes.

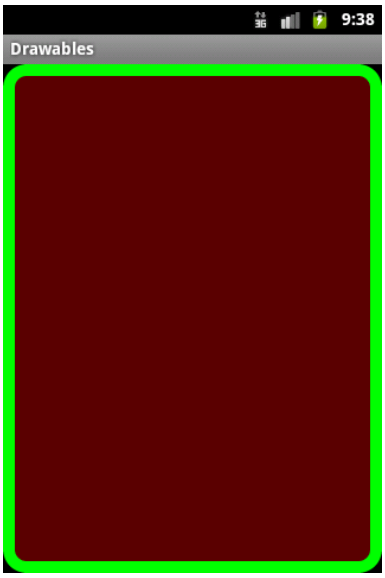
Cada forma será de un tipo determinado. El tipo se especifica mediante un atributo del elemento `shape`. Los posibles valores que este atributo `shape` podrá tomar son los siguientes:

- `oval`: como su propio nombre indica este tipo permitirá definir formas de tipo oval.
- `rectangle`: permite el uso de un subnodo `corners` con un atributo `radius` para establecer bordes redondeados.
- `ring`: podemos definir el radio interno y la anchura del anillo mediante los atributos `innerRadius` y `thickness`, respectivamente. También es posible establecer estos valores en función de la anchura del *drawable* por medio de los atributos `innerRadiusRatio` y `thicknessRatio`.

Para especificar cómo será el borde de la figura podemos utilizar el elemento `stroke`. En concreto, deberemos hacer uso de los atributos `width` y `color`. Otro nodo que podemos utilizar es `padding`, que permitirá desplazar la figura en el lienzo, para que no aparezca centrada. De todas formas, el nodo más habitual suele ser `solid`, que mediante el atributo `color` nos permite establecer el color de fondo.

A continuación mostramos un ejemplo de forma, el aspecto de la cual se puede ver en la siguiente figura:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="rectangle">
        <solid android:color="#f0600000"/>
        <stroke
                android:width="10dp"
                android:color="#00FF00"/>
        <corners
                android:radius="15dp" />
        <padding
                android:left="10dp"
                android:top="10dp"
                android:right="10dp"
        android:bottom="10dp"
        />
</shape>
```



Ejemplo de drawable: forma rectangular

1.3. Gradientes

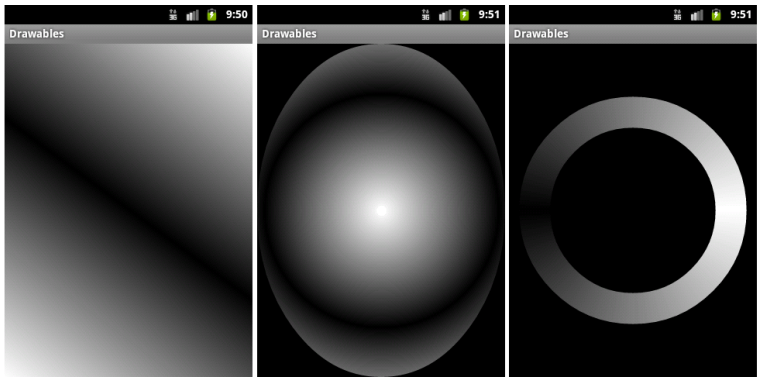
Un *GradientDrawable* permite incluir gradientes más o menos complejos en nuestra actividad. Se definen mediante una etiqueta `gradient` en XML. Cada *drawable* de este tipo requiere al menos el uso de los atributos `startColor` y `endColor`, que indican los dos colores entre los que se va a realizar la transición. También es posible el uso de un atributo `middleColor`, por lo que la transición se puede llegar a realizar entre tres colores diferentes. El atributo `type` permite definir el tipo de gradiente:

- `linear`: es el tipo por defecto. Muestra una transición directa desde `startColor` a `endColor`, con un ángulo que puede ser definido mediante el atributo `angle`.
- `radial`: dibuja un gradiente circular en el que se se produce una transición desde `startColor` en la parte más externa a `endColor` en el centro. Requiere el uso del atributo `gradientRadius` para indicar el radio del círculo a través del cual se producirá la transición. También es posible utilizar de manera opcional los atributos `centerX` y `centerY` para desplazar el centro del círculo. El atributo `gradientRadius` está definido en píxeles, por lo que será necesario definir un *drawable* de este estilo diferente para cada posible resolución de pantalla.
- `sweep`: el gradiente se muestra en el límite externo de la figura padre (normalmente un anillo).

El siguiente código muestra un ejemplo con tres gradientes diferentes. El resultado se puede contemplar en la siguiente figura:

```
<!-- Rectangulo con gradiente lineal -->
<?xml version="1.0" encoding="utf-8"?>
        <shape xmlns:android="http://schemas.android.com/apk/res/android"
                android:shape="rectangle"
```

```
                android:useLevel="false">
                <gradient
                        android:startColor="#ffffff"
                        android:endColor="#ffffff"
                        android:centerColor="#000000"
                        android:useLevel="false"
                        android:type="linear"
                        android:angle="45"
                        />
        </shape>
<!-- Oval gradiente radial -->
<?xml version="1.0" encoding="utf-8"?>
        <shape xmlns:android="http://schemas.android.com/apk/res/android"
                android:shape="oval"
                android:useLevel="false">
                <gradient
                        android:type="radial"
                        android:startColor="#ffffff"
                        android:endColor="#ffffff"
                        android:centerColor="#000000"
                        android:useLevel="false"
                        android:gradientRadius="300"
                />
        </shape>
<!-- Anillo con gradiente de tipo sweep -->
<?xml version="1.0" encoding="utf-8"?>
        <shape xmlns:android="http://schemas.android.com/apk/res/android"
                android:shape="ring"
                android:useLevel="false"
                android:innerRadiusRatio="3"
                android:thicknessRatio="8">
                <gradient
                        android:startColor="#ffffff"
                        android:endColor="#ffffff"
                        android:centerColor="#000000"
                        android:useLevel="false"
                        android:type="sweep"
                />
</shape>
```



Ejemplos de drawable a partir de gradientes

1.4. Imágenes

Las imágenes que introduzcamos en los directorios de recursos de tipo *drawable* (*/res/drawable/*) podrán ser tratadas igual que cualquier otro tipo de *drawable*. Por

ejemplo, si introducimos en dicho directorio una imagen `titulo.png`, podremos hacer referencia a ella en los atributos de los XML mediante `@drawable/titulo` (no se pone la extensión), o bien desde el código mediante `R.drawable.titulo`.

Las imágenes se encapsulan en la clase `Bitmap`. Los *bitmaps* pueden ser mutables o inmutables, según si se nos permite modificar el valor de sus pixels o no respectivamente.

Si el *bitmap* se crea a partir de un *array* de pixels, de un recurso con la imagen, o de otro *bitmap*, tendremos un *bitmap* inmutable.

Si creamos el *bitmap* vacío, simplemente especificando su altura y su anchura, entonces será mutable (en este caso no tendría sentido que fuese inmutable ya que sería imposible darle contenido). También podemos conseguir un *bitmap* mutable haciendo una copia de un *bitmap* existente mediante el método `copy`, indicando que queremos que el *bitmap* resultante sea mutable.

Para crear un *bitmap* vacío, a partir de un *array* de pixels, o a partir de otro *bitmap*, tenemos una serie de métodos estáticos `createBitmap` dentro de la clase `Bitmap`.

Para crear un *bitmap* a partir de un fichero de imagen (GIF, JPEG, o PNG, siendo este último el formato recomendado) utilizaremos la clase `BitmapFactory`. Dentro de ella tenemos varios métodos con prefijo `decode` que nos permiten leer las imágenes de diferentes formas: de un *array* de bytes en memoria, de un flujo de entrada, de un fichero, de una URL, o de un recurso de la aplicación. Por ejemplo, si tenemos una imagen (`titulo.png`) en el directorio de *drawables* podemos leerla como `Bitmap` de la siguiente forma:

```
Bitmap imagen = BitmapFactory.decodeResource(getResources(),
    R.drawable.titulo);
```

Al crear un *bitmap* a partir de otro, podremos realizar diferentes transformaciones (escalado, rotación, etc).

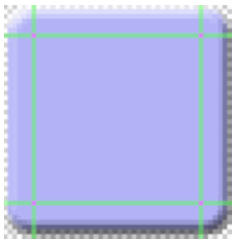
Una vez no se vaya a utilizar más el *bitmap*, es recomendable liberar la memoria que ocupa. Podemos hacer esto llamando a su método `recycle`.

1.5. Imágenes nine-patch

Como hemos comentado anteriormente, utilizaremos los *drawables* para especificar el aspecto que queremos que tengan los componentes de la interfaz. La forma más flexible de definir este aspecto es especificar una imagen propia. Sin embargo, encontramos el problema de que los componentes normalmente no tendrán siempre el mismo tamaño, sino que Android los "estirará" según su contenido y según los parámetros de *layout* especificados (es decir, si deben ajustarse a su contenido o llenar todo el espacio disponible). Esto es un problema, ya que si siempre especificamos la misma imagen como aspecto para estos componentes, al estirarla veremos que ésta se deforma, dando un aspecto terrible a nuestra aplicación, como podemos ver a continuación:

Imagen estirada

Sin embargo, tenemos un tipo especial de imágenes PNG llamadas *nine-patch* (llevan extensión `.9.png`), que nos permitirán evitar este problema. Normalmente la parte central de nuestros componentes es homogénea, por lo que no pasa nada si se estira. Sin embargo, los bordes si que contienen un mayor número de detalles, que no deberían ser deformados, especialmente las esquinas. Las imágenes *nine-patch* se dividen en 9 regiones: la parte central, que puede ser escalada en cualquier dirección, las esquinas, que nunca pueden escaladas, y los bordes, que sólo pueden ser escalados en su misma dirección (horizontal o vertical). A continuación vemos un ejemplo de dicha división:

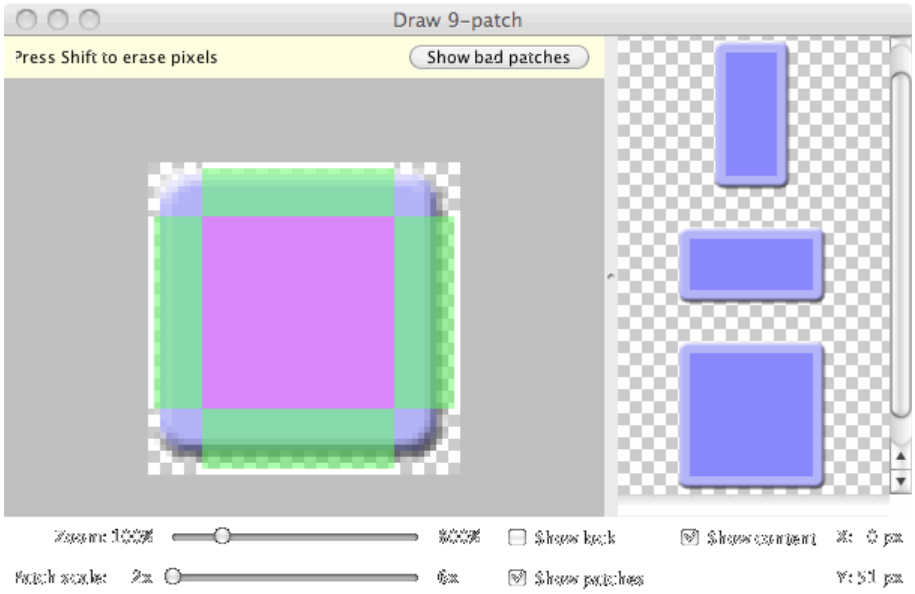


Parches de la imagen

Si ponemos una imagen de este tipo como *drawable* de fondo para un botón, veremos que siempre se mostrará con el aspecto correcto, independientemente de su contenido:

Aplicación de nine-patch a un botón

Podemos crear este tipo de imágenes con la herramienta `draw9patch` que podemos encontrar en el subdirectorio `tools` del SDK de Android. Lo único que necesitaremos es arrastrar el PNG que queramos tratar como *nine-patch*, y añadir una serie de píxeles en el marco de la imagen para marcar las regiones:



Herramienta draw9patch

La fila de píxeles superior y la columna izquierda indican las zonas de la imagen que son flexibles y que se pueden ampliar si es necesario repitiendo su contenido. En el caso de la fila superior, indica que se pueden estirar en la horizontal, mientras que los del lateral izquierdo corresponden a la vertical.

Opcionalmente podemos especificar en la fila inferior y en la columna derecha la zona que utilizaremos como contenido. Por ejemplo, si utilizamos la imagen como marco de un botón, esta será la zona donde se ubicará el texto que pongamos en el botón. Marcando la casilla *Show content* veremos en el lateral derecho de la herramienta una previsualización de la zona de contenido.

1.6. Lista de estados

Siguiendo con el ejemplo del botón, encontramos ahora un nuevo problema. Los botones no deben tener siempre el mismo aspecto de fondo, normalmente cambiarán de aspecto cuando están pulsados o seleccionados, sin embargo sólo tenemos la posibilidad de especificar un único *drawable* como fondo. Para poder personalizar el aspecto de todos los estados en los que se encuentra el botón tenemos un tipo de *drawable* llamado *state list drawable*. Se define en XML, y nos permitirá especificar un *drawable* diferente para cada estado en el que se puedan encontrar los componentes de la interfaz, de forma que en cada momento el componente mostrará el aspecto correspondiente a su estado actual.

Por ejemplo, podemos especificar los estados de un botón (no seleccionado, seleccionado, y pulsado) de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
```

```
        <!-- presionado -->
    <item android:state_pressed="true"
          android:drawable="@drawable/boton_pressed" />
    <!-- seleccionado -->
    <item android:state_focused="true"
          android:drawable="@drawable/boton_selected" />
    <!-- no seleccionado -->
    <item android:drawable="@drawable/boton_normal" />
</selector>
```

Los *drawables* especificados para cada estado pueden ser de cualquier tipo (por ejemplo imágenes, *nine-patch*, o formas definidas en XML).

Un *drawable* similar es el de tipo *level list*, pero en este caso los diferentes posibles *drawables* a mostrar se especifican para un rango de valores numéricos. ¿Para qué tipos de componentes de la interfaz podría resultar esto de utilidad?

1.7. Definición por capas

Es posible crear un *drawable* a partir de la composición de capas de otros *drawables* diferentes. Para ello se utiliza el elemento `layer-list`. Este elemento contendrá subnodos de tipo `item`, cuyo atributo `drawable` indicará a qué *drawable* se está haciendo referencia. Los elementos se añadirán en orden, quedando el primer elemento en el fondo de la pila.

```
<?xml version="1.0" encoding="utf-8"?>
        <layer-list
xmlns:android="http://schemas.android.com/apk/res/android">
                <item android:drawable="@drawable/bottomimage"/>
                <item android:drawable="@drawable/image2"/>
                <item android:drawable="@drawable/image3"/>
                <item android:drawable="@drawable/topimage"/>
        </layer-list>
```

1.8. Animación por fotogramas

Este tipo de *drawable* nos permite definir una animación a partir de diferentes fotogramas, que deberemos especificar también como *drawables*, además del tiempo en milisegundos que durará el fotograma. Se definen en XML de la siguiente forma:

```
<animation-list
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
  <item android:drawable="@drawable/spr0" android:duration="50" />
  <item android:drawable="@drawable/spr1" android:duration="50" />
  <item android:drawable="@drawable/spr2" android:duration="50" />
</animation-list>
```

Además, la propiedad `oneshot` nos indica si la animación se va a reproducir sólo una vez o en bucle infinito. Al ponerla como `false` especificamos que se reproduzca de forma continuada.

Desde el código, podremos obtener la animación de la siguiente forma, considerando que

la hemos guardado en un fichero `animacion.xml`:

```
AnimationDrawable animFotogramas =
getResources().getDrawable(R.drawable.animacion);
```

De forma alternativa, podríamos haberla definido de forma programática de la siguiente forma:

```
BitmapDrawable f1 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite0);
BitmapDrawable f2 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite1);
BitmapDrawable f3 = (BitmapDrawable)getResources()
    .getDrawable(R.drawable.sprite2);
AnimationDrawable animFotogramas = new AnimationDrawable();

animFotogramas.addFrame(f1, 50);
animFotogramas.addFrame(f2, 50);
animFotogramas.addFrame(f3, 50);

animFotogramas.setOneShot(false);
```

Nota

La diferencia entre `Bitmap` y `BitmapDrawable` reside en que en el primer caso simplemente tenemos una imagen, mientras que en el segundo lo que tenemos es un *drawable* que encapsula una imagen, es decir, se le podrá proporcionar a cualquier componente que acepte *drawables* en general como entrada, y concretamente lo que dibujará será la imagen (`Bitmap`) que contiene.

Para que comience la reproducción deberemos llamar al método `start` de la animación:

```
animFotogramas.start();
```

De la misma forma, podemos detenerla con el método `stop`:

```
animFotogramas.stop();
```

Importante

El método `start` no puede ser llamado desde el método `onCreate` de nuestra actividad, ya que en ese momento el *drawable* todavía no está vinculado a la vista. Si lo que queremos es que se ponga en marcha nada más cargarse la actividad, el lugar idóneo para invocarlo es el evento `onWindowFocusChanged`. Lo recomendable será llamar a `start` cuando obtengamos el foco, y a `stop` cuando lo perdamos.

1.9. Definición programática

Vamos a suponer que tenemos un `ImageView` con identificador `visor` y un drawable de nombre `rectangulo`. Normalmente especificaremos directamente en el XML el *drawable* que queremos mostrar en el `ImageView`. Para ello deberemos añadir el atributo `android:src = "@drawable/rectangulo"` en la definición del `ImageView`.

Podremos también obtener una referencia a dicha vista y mostrar en ella nuestro

rectángulo especificando el identificador del *drawable* de la siguiente forma:

```
ImageView visor = (ImageView)findViewById(R.id.visor);
visor.setImageResource(R.drawable.rectangulo);
```

Otra alternativa para mostrarlo es obtener primero el objeto `Drawable` y posteriormente incluirlo en el `ImageView`:

```
Drawable rectangulo = this.getResources()
    .getDrawable(R.drawable.rectangulo);
visor.setImageDrawable(rectangulo);
```

Estas primitivas básicas también se pueden crear directamente de forma programática. En el paquete `android.graphics.drawable.shape` podemos encontrar clases que encapsulan diferentes formas geométricas. Podríamos crear el rectángulo de la siguiente forma:

```
RectShape r = new RectShape();
ShapeDrawable sd = new ShapeDrawable(r);
sd.getPaint().setColor(Color.RED);
sd.setIntrinsicWidth(100);
sd.setIntrinsicHeight(50);

visor.setImageDrawable(sd);
```

2. Estilos y temas

Los estilos permiten modificar el aspecto de una vista o una ventana. Con un estilo podemos definir propiedades como altura, padding, color de fuente, tamaño de fuente, y muchas más cosas. Un estilo se define en un archivo XML separado de la definición del layout. La filosofía de los estilos en Android es la misma que la del lenguaje CSS en el caso del diseño web: permiten separar el diseño del contenido.

Para crear un estilo simplemente utilizamos un elemento XML `style` con un atributo `name` y uno o más elementos `item`. El elemento `item` debe incluir un atributo `name` para especificar el atributo al que hace referencia (como color o fuente) y su contenido será el valor que se le asigna a dicho atributo.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <style name="NombreEstilo">
                <item name="NombreAtributo">value</item>
        </style>
</resources>
```

Utilizando un estilo como el anterior podríamos hacer que a siguiente definición de layout:

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textColor="#00FF00"
    android:typeface="monospace"
    android:text="@string/hello" >
```

quede de la sigiente forma:

```
<TextView
        style="@style/EstiloFuente"
        android:text="@string/hello" />
```

siempre y cuando hubiéramos definido el estilo `EstiloFuente` en un archivo XML separado dentro de */res/values/* con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <style name="EstiloFuente">
                <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

Un **tema** es un estilo que se aplica a una aplicación o actividad completa en lugar de a una vista individual. Cuando un estilo se aplica a una actividad, se aplicará a cada vista de la misma las propiedades que sean compatibles con ella. Por ejemplo, todas las propiedades relacionadas con el texto, como el color o el tamaño de la fuente, se aplicarían a las vistas que contuvieran texto. Más adelante veremos cómo aplicar un estilo como tema.

2.1. Herencia

La herencia es un mecanismo que nos va a permitir crear estilos fácilmente, extendiendo estilos base. Para ello hacemos uso del atributo `parent`. El siguiente código muestra un ejemplo de estilo base en el que se establecen ciertas propiedades para el texto, y cómo se aplica la herencia para construir otro estilo a partir de éste pero con el texto más pequeño:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <style name="TextoBase">
                <item name="android:textSize">14sp</item>
                <item name="android:textColor">#111</item>
        </style>
        <style name="TextoPequenyo" parent="TextoBase">
                <item name="android:textSize">8sp</item>
        </style>
</resources>
```

Se puede aplicar la herencia tantas veces como sea necesario. Por ejemplo, sería posible definir un nuevo estilo a partir del estilo `TextoPequenyo` definido en el ejemplo anterior, también mediante herencia.

2.2. Cómo aplicar un estilo

Existen dos formas de aplicar un estilo:

- A una vista individual, añadiendo un atributo `style` al elemento XML de la vista en

el archivo de layout correspondiente.
- A una aplicación o actividad de manera global, añadiendo el atributo `android:theme` o a un elemento `activity` o al elemento `application` dentro del *Manifest* de la aplicación.

Cuando se aplica un estilo a una vista determinada, ésta será la única a la que se aplicará. Por otra parte, si se aplica a un `ViewGroup`, hemos de tener en cuenta que los elementos `View` que contenga **no** heredarán las propiedades definidas en el estilo. Sin embargo, sería posible aplicar el estilo a todas las vistas, aunque para ello sería necesario aplicarlo como un tema.

2.3. Selección de un tema basada en la versión de la plataforma

Las versiones más modernas de Android incluyen nuevos temas que podríamos querer aplicar a nuestras aplicaciones, pero en ese caso tendríamos que intentar conservar la compatibilidad con versiones anteriores del sistema. Esto se puede conseguir mediante un tema que utilice herencia y carpetas de recursos con sufijos para escoger un archivo de estilo u otro según la versión.

Por ejemplo, nuestra aplicación podría declarar un estilo que herede del estilo ligero por defecto de Android. Para ello, crearíamos el archivo */res/values/styles.xml* y añadiríamos lo siguiente:

```
<style name="LightThemeSelector" parent="android:Theme.Light">
    ...
</style>
```

Por otra parte, podríamos desear que en el caso en el que el dispositivo estuviera ejecutando la versión 3.0 de Android, éste usara un tema especial de esta versión. Para ello podríamos crear el archivo */res/values-v11/styles.xml* con el siguiente contenido:

```
<style name="LightThemeSelector" parent="android:Theme.Holo.Light">
    ...
</style>
```

2.4. Utilizando los estilos y temas definidos en el sistema

La plataforma Android proporciona una gran cantidad de temas y estilos que podemos aplicar a nuestras propias aplicaciones. Se puede encontrar una referencia a todos estos estilos en la clase `R.style`. Para usar cualquier estilo definido dentro de dicha clase sustituimos cada subrayado por un punto. Por ejemplo, podríamos aplicar el estilo `Theme_NoTitleBar` de la siguiente forma: `"@android:style/Theme.NoTitleBar"`. Otros ejemplos podrían ser los siguientes, que producirían como resultado una actividad en forma de diálogo o en forma de ventana semitransparente respectivamente:

```
<activity android:theme="@android:style/Theme.Dialog">
<activity android:theme="@android:style/Theme.Translucent">
```

Desafortunadamente la clase `R.style` no está bien documentada y los estilos no están descritos. Por lo tanto, la mejor manera de conocer cómo están definidos es mirando directamente el código fuente. El código fuente nos dirá qué propiedades se definen para cada estilo.

Para tener una referencia de qué propiedades o atributos podemos definir para cada estilo o tema (como `WindowBackground` o `TextAppearance`) podemos o bien consultar la clase `R.attr` o bien la descripción de la clase correspondiente a la vista que nos interese en la documentación de Android.

Nota:

Todo el código fuente se puede encontrar en *http://source.android.com/source/downloading.html*