Introducción al diseño de interfaces gráficas en Android

Índice

En esta sesión comenzaremos a tratar el tema de la interfaz gráfica de las aplicaciones en Android. Pero antes de empezar hemos de tener en cuenta que la terminología de Android referida a interfaces gráficas puede resultar un poco extraña al principio. Examinemos alguno de los conceptos que trataremos en mayor profundidad a lo largo de la sesión:

- Las **Vistas** son la base del desarrollo de interfaces gráficas en Android. Todos los elementos gráficos, comunmente llamados *widgets* o *componentes* en otros entornos, son subclases de `View`. Otros elementos más complejos, como agrupaciones de vistas o layouts también heredan de esta clase. Así pues, en Android, un botón será una vista, así como también un campo de edición de texto.
- Los **Grupos de Vistas** son extensiones de la vista genérica. Subclases de `ViewGroup`, pueden contener múltiples vistas hijas. Los layouts son extensiones de los grupos de vistas.
- Los **Layouts** son medios por los que organizar vistas en la pantalla del dispositivo. Existen diferentes tipos de *layout* que nos permitirán disponer los elementos de la pantalla de diversas maneras.
- Cada una de las pantallas o ventanas de nuestra aplicación se corresponderá con una **Actividad**. Las actividades ya han sido tratadas en detalle en sesiones anteriores. En Android son el equivalente de los formularios. Para mostrar una interfaz de usuario la operación que se debe realizar es asignar una vista (normalmente un layout) a una actividad.

1. Vistas

Todos los componentes visuales en Android son una subclase de la clase `View`. No se les llama *widgets* para no confundirlos con las aplicaciones de tipo widget que se pueden mostrar en la ventana inicial de Android. En sesiones anteriores ya hemos conocido algunas vistas: el botón (clase `Button`) y la etiqueta de texto (clase `TextView`).

1.1. Crear interfaces de usuario con vistas

Cuando se inicia una actividad, lo hace con una pantalla temporalmente vacía sobre la que deberemos colocar todos los elementos gráficos de su interfaz. Para asignar el interfaz de usuario se hace uso del método `setContentView`, pasando como parámetro una instancia de la clase `View` o de alguna de sus subclases. Este método también acepta como parámetro un identificador de recurso, correspondiente a un archivo XML con una descripción de interfaz gráfica. Este segundo método suele ser el más habitual.

Usar recursos de tipo layout permite separar la capa de presentación de la capa lógica, proporcionando la suficiente flexibilidad para cambiar la interfaz gráfica sin necesidad de modificar ni una línea de código. Definir la interfaz gráfica por medio de recursos en lugar de mediante código permite también especificar diferentes layouts en función de diferentes configuraciones de hardware o incluso que se pueda modificar la interfaz en tiempo de ejecución cuando se produzca algún evento, como por ejemplo cambiar la

orientación de la pantalla.

En el siguiente código vemos como inicializar la interfaz gráfica a partir de un layout definido en los recursos de la aplicación. Normalemente el recurso consistirá en un archivo XML almacenado en la carpeta `/res/layout/`. En este ejemplo se supone que se está haciendo uso del archivo *milayout.xml*, el cual se encuentra precisamente en dicha carpeta:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.milayout);
        TextView miTexto = (TextView)findViewById(R.id.texto);
}
```

En el ejemplo anterior también se puede observar cómo acceder desde el código de la actividad a cualquiera de las vistas definidas en el layout, por medio de una llamada al método `findViewById`. En este caso se crea una instancia de la clase `TextView`, que se corresponderá con el elemento `TextView` del layout *milayout*, con lo que podremos acceder a la vista desde el código para modificar el texto que muestra, poder asociarle manejadores de eventos, etc. El método `findViewById` recibe como parámetro el identificador de la vista en el layout. El identificador de una vista se especifica mediante su atributo `android:id` y tiene la siguiente sintaxis: `android:id="@+id/[IDENTIFICADOR]"`, donde `[IDENTIFICADOR]` es el identificador que le queremos asignar al elemento.

Como se ha comentado anteriormente, la otra alternativa para crear la interfaz gráfica de una actividad es construirla directamente desde el código fuente. En el siguiente ejemplo vemos cómo asignar una interfaz gráfica compuesta únicamente por un `TextView` a una actividad cualquiera:

```
@Override
public void onCreate(Bundle sagedInstaceState) {
        super.onCreate(savedInstanceState);

        textView texto = new TextView(this);
        setContentView(texto);

        texto.setText("Hola Mundo!");
}
```

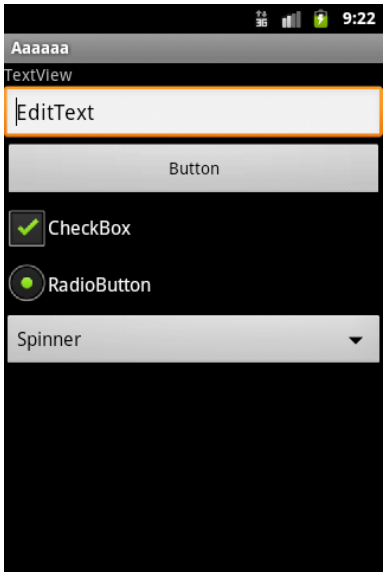
El mayor inconveniente es que el método `setContentView` tan solo acepta como parámetro una única vista, así que si queremos una interfaz más compleja deberemos pasar como parámetro algún layout que deberemos haber construido manualmente en el código.

1.2. Las vistas de Android

Android proporciona una gran variedad de vistas básicas para poder diseñar interfaces de usuario de manera sencilla. Usando estas vistas podremos desarrollar aplicaciones

consistentes visualmente con las del resto del sistema, aunque también es posible modificarlas o ampliar sus funcionalidades. Algunas de las vistas más utilizadas son:

- **TextView**: una etiqueta de sólo lectura. Permite texto multilínea, formateado de cadenas, etc.
- **EditText**: una caja de texto editable. Permite texto multilínea, texto flotante, etc.
- **ListView**: un grupo de vistas que crea y administra una lista vertical de vistas, correspondiéndose cada una de ellas a una fila de la lista.
- **Spinner**: un cuadro de selección que permite seleccionar un elemento de una lista de posibles opciones. Se muestra como un botón que al ser pulsado visualiza el listado de posibles opciones.
- **Button**: un botón normal.
- **CheckBox**: un botón con dos posibles estados representado por un recuadro que puede estar marcado o no.
- **RadioButton**: un grupo de botones con dos posibles estados. Uno de estos grupos muestra al usuario un conjunto de opciones de las cuales sólo una puede estar activa simultáneamente.
- **SeekBar**: una barra de desplazamiento, que permite escoger visualmente un valor dentro de un rango entre un valor inicial y final.



Vistas básicas de Android

Estos son sólo unos ejemplos. Android pone a nuestra disposición vistas más avanzadas, incluyendo selectores de fecha, cajas de texto con autocompletado, mapas, galerías y pestañas. En *http://developer.android.com/guide/tutorials/views/index.html* se puede consultar un listado de los widgets disponibles en Android.

2. Layouts

Los layouts son subclases de `ViewGroup` utilizadas para posicionar diferentes vistas en nuestra interfaz gráfica. Los layouts se pueden anidar con el objetivo de conseguir crear layouts más complejos. Algunos de los layouts disponibles en Android son:

- **FrameLayout**: es el layout más simple. Lo único que hace es añadir cada vista a la esquina superior izquierda. Si se añaden varias vistas se van colocando una encima de la otra, de tal forma que las vistas superiores ocultan a las inferiores.
- **LinearLayout**: alinea diferentes vistas ya sea en una línea horizontal o una línea vertical. Un `LinearLayout` vertical consiste en una columna de vistas, mientras que un `LinearLayout` horizontal no es más que una fila de vistas. Este layout permite establecer un peso mediante la propiedad `weight` a cada elemento, lo que controlará el tamaño relativo de cada elemento en la vista.
- **RelativeLayout**: es el layout nativo más flexible, permitiendo definir la posición de cada una de sus vistas de manera relativa a la posición de los demás o a los bordes de la pantalla.
- **TableLayout**: permite disponer un conjunto de vistas en una rejilla formada por varias filas y columnas. Es posible permitir que las filas o columnas crezcan o disminuyan de tamaño.
- **Gallery**: muestra una lista horizontal de vistas en la que se puede navegar mediante un scroll.

La documentación de Android describe en detalle las características y propiedades de cada layout. Esta documentación se puede consultar en *http://developer.android.com/guide/topics/ui/layout-objects.html*.

2.1. Utilizando layouts

La manera habitual de hacer uso de layouts es mediante un fichero XML definido como un recurso de la aplicación, en la carpeta */res/layout/*. Este fichero debe contener un elemento raíz, el cual podrá contener de manera anidada tantos layouts y vistas como sea necesario. A continuación se muestra un ejemplo de layout en el que un `TetView` es colocado sobre un `EditText` usando un `LinearLayout` vertical:

```
<?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">
        <TextView
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Introduce un texto"
        />
        <EditText
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Escribe el texto aquí"
        />
</LinearLayout>
```

Para cada elemento se debe suministrar un valor para los atributos `layout_width` y `layout_height`. Se podría utilizar una altura o anchura exacta en píxeles, pero no es aconsejable, ya que nuestra aplicación podría ejecutarse en terminales con diferentes resoluciones o tamaños de pantalla. Los valores más habituales para estos atributos, y que además permiten que haya independencia del hardware, son los usados en este ejemplo: `fill_parent` y `wrap_content`.

El valor `wrap_content` limita el tamaño de una vista al mínimo requerido para poder mostrar sus contenidos. Por su parte, el valor `fill_parent` expande la vista para que ocupe todo el tamaño disponible en su vista padre (o en la pantalla, si se trata del elemento raíz). En el ejemplo anterior el layout situado como elemento raíz ocupará toda la pantalla. Las dos vistas dentro del layout ocuparán toda la anchura disponible, mientras que su altura será tan sólo la necesaria para mostrar sus respectivos textos.

En el caso en el que sea estrictamente necesario por algún motivo (ya que es algo que se desaconseja en general) es posible implementar un layout en el propio código fuente. Cuando asignamos vistas a layouts por medio de código, es importate hacer uso de `LayoutParameters` por medio del método `setLayoutParams`, o pasándolos como parámetro en la llamada a `addView`, tal como se muestra en el siguiente ejemplo:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);

TextView texto = new TextView(this);
EditText edicion = new EditText(this);

text.setText("Introduce un texto");
edicion.setText("Escribe el texto aquí");

int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LienarLayout.LayoutParams.WRAP_CONTENT;

ll.addView(texto, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(edicion, new LinearLayout.layoutParams(lHeight, lWidth));

setContentView(ll);
```

2.2. Optimizar layouts

El proceso mediante el cual se rellena la pantalla correspondiente a una actividad de elementos gráficos es costoso computacionalmente. El añadir un nuevo layout o una nueva vista puede tener un gran impacto en la latencia a la hora de navegar por las diferentes actividades de nuestra aplicación.

Suele ser una buena práctica, en general, utilizar layouts tan simples como sea posible. Para conseguir una interfaz más eficiente podemos seguir los siguientes consejos:

- Evitar anidamientos innecesarios: no introduzcas un layout dentro de otro a menos que sea estrictamente necesario. Un `LinearLayout` dentro de un `FrameLayout`, usando ambos el valor `fill_parent` para su altura y su anchura, no producirá ningún cambio en el aspecto final de la interfaz y su único efecto será aumentar el coste

computacional de construir dicha interfaz. Buscar layouts redundantes, sobre todo si has hecho varios cambios en la interfaz.
- Evitar usar demasiadas vistas: cada vista que se añade a la interfaz requiere recursos y tiempo de ejecución.
- Evitar anidamientos profundos: debido a que los layouts pueden ser anidados de cualquier manera y sin limitaciones es fácil caer en la tentación de construir estructuras complejas, con muchos anidamientos. Aunque no exista un límite para el nivel de anidamiento, deberemos intentar que sea lo más bajo posible.

Para ayudarnos en la tarea de optimizar nuestros layouts disponemos del comando `layoutopt`, incluido en el SDK de Android y que puede ser ejecutado en la línea de comandos. Para analizar un layout o conjunto de layouts ejecutamos el comando pasándole como parámetro el nombre de un recurso de tipo layout o una carpeta de recursos de este tipo. El comando nos mostrará diferentes recomendaciones para mejorar nuestros layouts.

3. Uso básico de vistas y layouts

En esta sección veremos algunos detalles sobre cómo utilizar algunas de las vistas y layouts proporcionados por Android. Se debe tener en cuenta que esto no es más que una introducción, y que es posible encontrar una descripción más detallada de todos estos elementos en la documentación de Android.

3.1. TextView

Se corresponde con una etiqueta de texto simple, que sirve evidentemente para mostrar un texto al usuario. Su atributo más importante es `android:text`, cuyo valor indica el texto a mostrar por pantalla. También pueden ser de interés los atributos `android:textColor` y `android:textSize`. Una curiosidad es que es posible hacer el `TextView` editable por medio del atributo booleano `android:editable`; sin embargo, esta vista no está preparada para este tipo de acción. Sólo se podrá editar texto a través de su subclase `EditText`.

Dentro del código los métodos más utilizados son `setText` y `appendText`, que permiten modificar el texto del `TextView` o añadir texto adicional durante la ejecución del programa. En ambos casos el parámetro recibido será una cadena. Para acceder a su texto usaremos el método `getText`.

3.2. EditText

`EditText` no es más que una subclase de `TextView` que está preparada para la edición de texto. Al pulsar sobre la vista la interfaz de Android mostrará un teclado para poder introducir nuestros datos. En el código se maneja igual que un `TextView` (por medio de

los métodos `getText` o `setText`, por ejemplo).

3.3. Button

Esta vista representa un botón normal y corriente, con un texto asociado. Para cambiar el texto del botón usaremos su atributo `android:text`.

La forma más habitual de interactuar con un botón es pulsarlo para desencadenar un evento. Para que nuestra aplicación realice una determinada acción cuando el botón sea pulsado, deberemos implementar un manejador para el evento *OnClick*. Veamos un ejemplo. Supongamos una actividad cuyo layout viene definido por el siguiente fichero XML:

```
<?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">
        <TextView
                android:id="@+id/texto"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Texto"
        />
        <Button
                android:id="@+id/boton"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Púlsame"
        />
</LinearLayout>
```

En el método `onCreate` de la actividad podríamos incluir el siguiente código para que cuando se pulsara el botón se modificara el texto del `TextView`:

```
public class MiActividad extends Activity {
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        setContentView(R.layout.miLayout);

                TextView texto = (TextView)findViewById(R.id.texto);
        Button boton = (Button)findViewById(R.id.boton);
        boton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                texto.setText("Botón pulsado");
            }
        });
    }
}
```

También es posible definir el manejador para el click del ratón en el propio fichero XML, a través del atributo `android:onClick`. El valor de este atributo será el nombre del método que se encargará de manejar el evento. Según la especificación del botón definido en el siguiente ejemplo:

```
<Button
```

```
     android:layout_height="wrap_content"
     android:layout_width="wrap_content"
     android:text="@string/textoBoton"
     android:onClick="manejador" />
```

cada vez que se pulse el botón se lanzará el método `manejador`, cuya cabecera deberá ser la siguiente:

```
public void manejador(View vista) {
        // Hacer algo
}
```

3.4. CheckBox

Se trata de un botón que puede encontrarse en dos posibles estados: seleccionado o no seleccionado. Este tipo de botones, al contrario que en el caso de los pertenecientes a la vista `RadioButton`, son totalmente independientes unos de otros, por lo que varios de ellos pueden encontrarse seleccionados al mismo tiempo. Dentro del código se manejan individualmente. Los métodos más importantes para manejar vistas de tipo `CheckBox` son `isChecked`, que devuelve un booleano indicando si el botón está seleccionado, y `setChecked`, que recibe un valor booleano como parámetro y sirve para indicar el estado del botón.

Veamos otro ejemplo. Supongamos que el layout de nuestra actividad se define 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">
        <TextView
                android:id="@+id/texto"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:text="Texto"
        />
        <CheckBox
                android:id="@+id/micheckbox"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Púlsame"
                android:checked="false"
        />
</LinearLayout>
```

Obsérvese como se hace uso del atributo `android:text` del `CheckBox` para establecer el texto que acompañará al botón. También se ha hecho uso del atributo `android:checked` para establecer el estado inicial del botón. Para hacer que nuestra actividad reaccione a la pulsación del `CheckBox` implementamos el manejador del evento *OnClick* para el mismo:

```
public class MiActividad extends Activity {
        TextView texto;
        CheckBox miCheckbox;
```

```
     protected void onCreate(Bundle icicle) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
                texto = (TextView)findViewById(R.id.texto);
                miCheckbox = (CheckBox)findViewById(R.id.micheckbox);
                miCheckbox.setOnClickListener(new OnClickListener() {
                    public void onClick(View v) {
                        // Realizamos una acción u otra según si el botón
está pulsado o no
                        if (miCheckbox.isChecked()) {
                            texto.setText("activado");
                        } else {
                            texto.setText("desactivado");
                        }
                    }
                });
        }
}
```

3.5. RadioButton

Un `RadioButton`, al igual que un `CheckButton`, tiene dos estados (seleccionado y no seleccionado). La diferencia con el anterior es que una vez que el botón está en el estado de seleccionado no puede cambiar de estado por intervención directa del usuario. Los elementos de tipo `RadioButton` suelen agruparse normalmente en un elemento de tipo `RadioGroup`, de tal forma que la activación de uno de los botones del grupo supondrá la desactivación de los demás.

En el siguiente ejemplo vemos como definir un grupo formado por dos botones de radio en el archivo XML de layout de una actividad:

```
<RadioGroup
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
      <RadioButton android:id="@+id/radio_si"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Sí" />
      <RadioButton android:id="@+id/radio_no"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="No" />
</RadioGroup>
```

Para que se realice una determinada acción al pulsar uno de los botones de radio deberemos implementar el manejador del evento *OnClick*:

```
public class MiActividad extends Activity {
        boolean valor = true;
        RadioButton botonSi, botonNo;

     protected void onCreate(Bundle icicle) {
             super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

             botonSi = (RadioButton)findViewById(R.id.radio_si);
             botonNo = (RadioButton)findViewById(R.id.radio_no);
```

```
                botonSi.setOnClickListener(new OnClickListener() {
                    public void onClick(View v) {
                        valor = true;
                    }
                });

                botonNo.setOnClickListener(new OnClickListener() {
                    public void onClick(View v) {
                        valor = false;
                    }
                });
        }
}
```

3.6. Spinner

Un `Spinner` es una vista que permite escoger uno de entre una lista de elementos. Es el equivalente a un cuadro de selección de un formulario normal. Construir un `Spinner` en Android requiere varios pasos, que vamos a ver a continuación.

- En primer lugar añadimos el `Spinner` a nuestro layout. Los dos atributos específicos de este tipo de vista que hemos añadido al ejemplo son `drawSelectorOnTop`, que dibujará un control encima de la opción seleccionada, y `android:prompt`, que indica la cadena de texto a mostrar cuando todavía no se ha seleccionado ninguna opción. Obsérvese como en este caso su valor se ha definido a partir de una cadena definida en el archivo *strings.xml* de los recursos de la aplicación. Esto es así porque no es posible asignarle una cadena como valor a este atributo.

```
<Spinner
        android:id="@+id/spinner"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:drawSelectorOnTop="true"
        android:prompt="@string/eligeopcion"
    />
```

- Añadimos la cadena para el *prompt* en */res/values/strings.xml*:

```
<string name="eligeopcion">Elige!</string>
```

- A continuación debemos indicar cuáles serán las opciones disponibles en el `Spinner`. Esto se hará también mediante los recursos de la aplicación. En concreto, crearemos un fichero en */res/values/* al que llamaremos *arrays.xml* y que contendrá lo siguiente:

```
<resources>
        <string-array name="opciones">
                <item>Mensual</item>
                <item>Trimestral</item>
                <item>Semestral</item>
                <item>Anual</item>
        </string-array>
</resources>
```

- Finalmente añadiremos el código necesario en el método `onCreate` de la actividad para que se asocien las opciones al `Spinner`. Para ello se utiliza un objeto de la clase

`ArrayAdapter`, que asocia cada cadena de nuestro `string-array` a una vista que será mostrada en el `Spinner`. Al método `createFromResource` le pasamos el contexto de la aplicación, el identificador del `string-array` y un identificador para indicar el tipo de vista que queremos asociar a cada elemento seleccionable. El método `setDropDownViewResource` se utiliza para definir el tipo de layout que se utilizará para mostrar la lista completa de opciones. Finalmente se asocia el adaptador al `Spinner`:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Spinner s = (Spinner) findViewById(R.id.spinner);
    ArrayAdapter adaptador = ArrayAdapter.createFromResource(
            this, R.array.opciones, android.R.layout.simple_spinner_item);
    adaptador.setDropDownViewResource(
                    android.R.layout.simple_spinner_dropdown_item);
    s.setAdapter(adaptador);
}
```

Con respecto a los eventos relacionados con el `Spinner`, hemos de tener en cuenta que Android no soporta el manejo de eventos para sus elementos individuales. Si intentamos crear un manejador para el evento *OnClick* para cada opción por separado, por ejemplo, se producirá un error en tiempo de ejecución. En lugar de ello deberemos definir un manejador de evento global para el `Spinner`, que tendrá el siguiente aspecto:

```
spinner.setOnItemSelectedListener(new OnItemSelectedListener() {
        public void onItemSelected(AdapterView<?> arg0, View arg1,
                                   int arg2, long arg3) {
                // Hacer algo
        }

        public void onNothingSelected(AdapterView<?> arg0) {
                // Hacer algo
        }
});
```

3.7. LinearLayout

Como se ha comentado anteriormente se trata de un layout que organiza sus componentes en una única fila o una única columna, según éste sea horizontal o vertical, respectivamente. Para establecer la orientación podemos utilizar o bien el atributo `android:orientation` en la definición del layout en el archivo XML (tomando como valor `horizontal` o `vertical`, siendo el primero el valor por defecto) o mediante el método `setOrientation` desde el código.

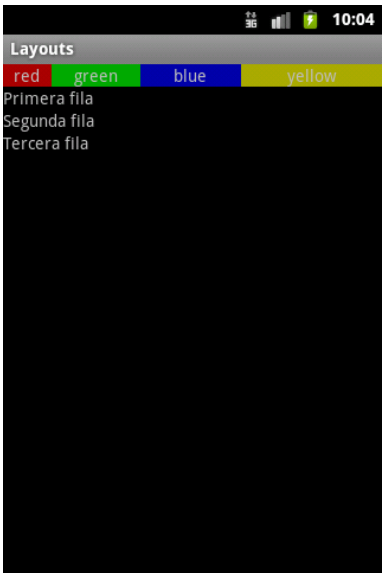
Un ejemplo de layout de este tipo sería:

```
<?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">
```

```
  <LinearLayout
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <TextView
          android:text="red"
          android:gravity="center_horizontal"
          android:background="#aa0000"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="1"/>
      <TextView
          android:text="green"
          android:gravity="center_horizontal"
          android:background="#00aa00"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="2"/>
      <TextView
          android:text="blue"
          android:gravity="center_horizontal"
          android:background="#0000aa"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="3"/>
      <TextView
          android:text="yellow"
          android:gravity="center_horizontal"
          android:background="#aaaa00"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:layout_weight="4"/>
  </LinearLayout>

  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="Primera fila"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
    <TextView
        android:text="Segunda fila"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
    <TextView
        android:text="Tercera fila"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
  </LinearLayout>

</LinearLayout>
```



Ejemplo de LinearLayout

En este ejemplo podemos ver el uso de un par de atributos interesantes. En primer lugar tenemos `android:layout-weight`, que va a permitir que las diferentes vistas dentro del `LinearLayout` ocupen una porción del espacio disponible proporcional a su peso. Por otra parte tenemos `android:gravity`, que permite alinear el texto de una vista. En este caso se ha centro el texto horizontalmente.

3.8. TableLayout

Se trata de un layout que organiza sus elementos como una rejilla dividida en filas y columnas. Se compone de un conjunto de elementos de tipo `TableRow` que representan a las diferentes filas de la tabla. Cada fila a su vez se compone de un conjunto de vistas. Existe la posibilidad de tener filas con diferente número de vistas; en este caso la tabla tendrá tantas columnas como celdas tenga la fila que contenga más vistas.

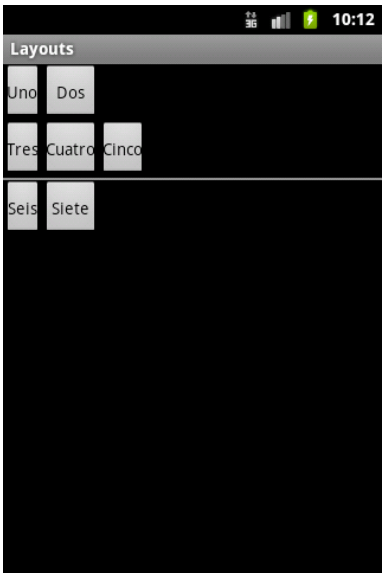
La anchura de una columna vendrá definida por la fila que contenta la vista de máxima anchura en dicha columna. Esto quiere decir que no tenemos libertad para asignar valor al atributo `android:layout_width`, que debería valer `match_parent`. Sí que podemos definir la altura, aunque lo más habitual será utilizar el valor `wrap_content` para el atributo `android:layout_height`. Si no se indica lo contrario esos serán los valores por defecto.

Aunque no podamos controlar la anchura de una columna, si que podemos tener columnas de anchura dinámica, usando los métodos `setColumnShrinkable()` y `setColumnStretchable()`. El primero permitirá disminuir la anchura de una columna con tal de que su correspondiente tabla pueda caber correctamente en su elemento padre. Por su parte, el segundo permitirá aumentar la anchura de una columna para hacer que la tabla pueda ocupar todo el espacio que tenga disponible. Hay que tener en cuenta que se

pueden utilizar ambos métodos para una misma columna; en este caso la anchura de la columna siempre se incrementará hasta que se utilice todo el espacio disponible, pero nunca más. Otra opción disponible es ocultar una columna por medio del método `setColumnCollapsed`.

El siguiente ejemplo muestra un `TableLayout` definido como recurso de la aplicación por medio de un fichero XML. Como se puede observar en el código se ha introducido un elemento `View` cuya única función es de servir de separador visual entre la segunda y la tercera fila:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TableRow>
        <Button
            android:text="Uno"
            android:padding="3dip" />
        <Button
            android:text="Dos"
            android:padding="3dip" />
    </TableRow>
        <TableRow>
        <Button
            android:text="Tres"
            android:padding="3dip" />
        <Button
            android:text="Cuatro"
            android:padding="3dip" />
        <Button
            android:text="Cinco"
            android:padding="3dip" />
    </TableRow>
    <View
        android:layout_height="2dip"
        android:background="#FF909090" />
    <TableRow>
        <Button
            android:text="Seis"
            android:padding="3dip" />
        <Button
            android:text="Siete"
            android:padding="3dip" />
    </TableRow>
</TableLayout>
```



Ejemplo de TableLayout

3.9. RelativeLayout

Este layout es muy flexible; permite indicar la posición de un elemento en función de otros y de los bordes de la pantalla. Enumerar todos sus posibles atributos alargaría innecesariamente esta sección, así que mostraremos un ejemplo que deje claro su funcionamiento, dejando al lector la tarea de acudir a la documentación de Android en el caso de que necesite profundizar más:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/etiqueta"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Escribe"/>
    <EditText
        android:id="@+id/entrada"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_below="@id/etiqueta"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entrada"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dip"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/ok"
```

```
        android:layout_alignTop="@id/ok"
        android:text="Cancelar" />
</RelativeLayout>
```

Ejemplo de RelativeLayout

4. Interfaces independientes de densidad y resolución

Durante el primer año de vida del sistema Android tan solo existía un único terminal que hiciera uso de este sistema, por lo que a la hora de diseñar una interfaz gráfica no era necesario tener en mente diferentes configuraciones de hardware. A principios del 2010 se produjo una avalancha en el mercado de nuevos dispositivos basados en Android. Una consecuencia de esta variedad fue un aumento también en las posibles configuraciones de resolución o densidad.

A la hora de diseñar una interfaz gráfica para nuestra aplicación Android es importante tener en cuenta que ésta podría llegar a ejecutarse en una gran variedad de configuraciones hardware distintas, con diferentes resoluciones (HVGA, QVGA o WVGA) y densidades de pantalla (3.2 pulgadas, 3.7, 4, etc.). Los tablets cada vez están más de moda, y es posible que otro tipo de dispositivos incluyan un sistema Android en el futuro.

En esta sección trataremos algunas cuestiones relativas a cómo diseñar nuestras interfaces gráficas teniendo todo esto en cuenta.

4.1. Múltiples archivos de recurso

En la primera sesión del módulo de Android tratamos el tema de los recursos de la

aplicación. Vimos que para determinados tipos de recurso era posible crear una estructura paralela de directorios que permitiera almacenar recursos a utilizar con diferentes configuraciones de hardware. En este apartado vemos diferentes sufijos que podemos añadir al nombre de la carpeta *layout* o *drawable* dentro de los recursos de la aplicación para definir diferentes interfaces según la configuración de nuestra pantalla.

- **Tamaño de la pantalla**: tamaño de la pantalla relativa a un terminal de tamaño "estándar":
  - `small`: una pantalla con un tamaño menor de 3.2 pulgadas.
  - `medium`: para dispositivos con un tamaño de pantalla típico.
  - `large`: para pantallas de un tamaño significativamente mayor que la de un terminal típico, como la pantalla de una tablet o un netbook.
- **Densidad**: se refiere a la densidad en píxeles de la pantalla. Normalmente se mide en puntos por pulgada (*dots per inch* o *dpi*). Se calcula en función de la resolución y el tamaño físico de la pantalla:
  - `ldpi`: usado para almacenar recursos para baja densidad pensados para pantallas con densidades entre 100 y 140dpi.
  - `mdpi`: usado para pantallas de densidad media entre 140 y 180dpi.
  - `hdpi`: usado para pantalla de alta densidad, entre 190 y 250dpi.
  - `nodpi`: usado para recursos que no deberían ser escalados, sea cual sea la densidad de la pantalla donde van a ser mostrados.
- **Relación de aspecto** (aspect ratio): se trata de la relación de la altura con respecto a la anchura de la pantalla:
  - `long`: usado para pantallas que son mucho más anchas que las de los dispositivos estándar.
  - `notlong`: usado para terminales con una relación de aspecto estándar.

Cada uno de estos sufijos puede ser utilizado de manera independiente o en combinación con otros. A continuación tenemos un par de ejemplos de carpetas de recursos para layouts que hacen uso de alguno de los listados anteriormente:

```
/res/layout-small-long/
/res/layout-large/
/res/drawable-hdpi/
```

4.2. Indicar las configuraciones de pantalla soportadas por nuestra aplicación

En el caso de alguna aplicación puede que no sea posible optimizar la interfaz para todas y cada una de las configuraciones de pantalla existentes. En estos casos puede ser útil utilizar el elemento `supports-screens` en el *Manifest* de la aplicación para especificar en qué pantallas puede ésta ser ejecutada. En el siguiente código se muestra un ejemplo:

```
<supports-screens
        android:smallScreens="false"
        android:normalScreens="true"
        android:largeScreens="true"
```

```
        android:anyDensity="true"
/>
```

Un valor de `false` en alguno de estos atributos forzará a Android a mostrar la interfaz por medio de un modo de compatibilidad que consistirá básicamente en realizar un escalado. Esto no suele funcionar siempre de manera correcta, haciendo aparecer elementos extraños o artefactos en la interfaz.

4.3. Prácticas a seguir para conseguir interfaces independientes de la resolución

En esta sección resumimos algunas técnicas que podremos utilizar en nuestras aplicaciones para que puedan ser visualizadas de manera correcta en la inmensa mayoría de las configuraciones hardware. Pero antes de empezar hemos de tener en cuenta que la consideración más importante a tener en cuenta es que nunca debemos presuponer cuál será el tamaño de la pantalla del terminal en el que se ejecutará nuestra aplicación. La regla de oro es crear layouts para diferentes clases de pantallas (pequeña, normal y grande) y para diferentes resoluciones (baja, media y alta).

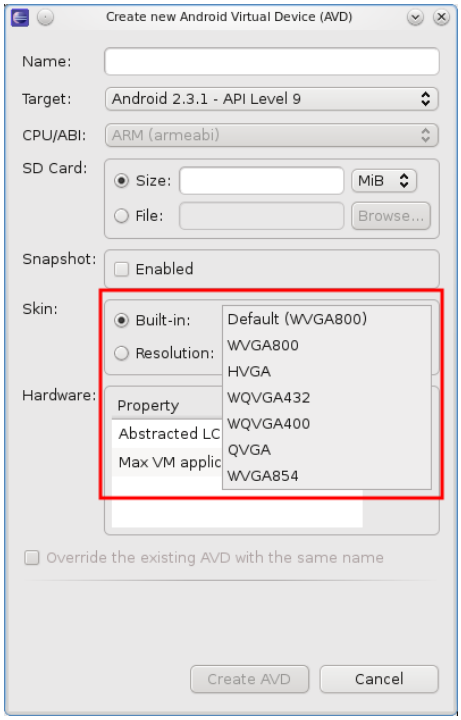
La primera regla es **no utilizar nunca tamaños absolutos basados en número de píxeles**. Esto se aplica tanto a layouts, como a vistas y fuentes. En particular se debería evitar usar el `AbsoluteLayout`, que se basa en la especificación de la disposición de sus elementos a partir de coordenadas en píxeles. Haciendo uso del `RelativeLayout` se pueden generar interfaces suficientemente complejas, evitando el uso de coordenadas absolutas, por lo que es una mejor solución.

A la hora de determinar el tamaño de una vista o un layout utilizaremos valores como `fill_parent` o `wrap_content`. También podría ser posible asignar valores a los atributos `layout-width` y `layout-height` medidos en píxeles independientes de la densidad (dp) o en píxeles independientes de la escala (sp). Esto mismo podría ser aplicado a las fuentes. Las medidas independientes de la densidad o de la escala son un medio por el que se puede especificar el tamaño de los componentes de nuestra interfaz de tal forma que al mostrarse en diferentes configuraciones hardware se escalen para seguir mostrando el mismo aspecto. Por ejemplo, un dp equivale a un píxel en una pantalla de 160dpi. Una línea de 2dp de anchura aparecerá como una línea de 3 píxeles de anchura en una pantalla de 240dpi.

Por supuesto no debemos olvidar **utilizar sufijos** para crear una estructura paralela de directorios de recursos, tal como se explicó en la sección anterior. Como mínimo se deberían definir las siguientes carpetas con sus recursos correspondientes:

```
/res/drawable-ldpi/
/res/drawable-mdpi/
/res/drawable-hdpi/
/res/layout-small/
/res/layout-normal/
/res/laout-large/
```

Por último, no debemos olvidar **probar nuestra aplicación en la mayor variedad de dispositivos posibles**. En este sentido podemos hacer uso de la emulación y de Android SDK, ya que es poco práctico hacer estas pruebas en dispositivos reales. Hemos de recordar que a la hora de crear un dispositivo virtual para probar nuestras aplicaciones Android podemos especificar diferentes configuraciones de pantalla, tanto en cuanto a resolución como en cuanto a densidad. La forma más sencilla de probar diferentes configuraciones es hacer uso de los skins que incorpora el AVD Manager del SDK. Crea un dispositivo virtual para cada opción de skin disponible y prueba tu aplicación en todos ellos.



Selección de skins en el AVD del SDK de Android