Clase 4

Curso Desarrollo en ANDROID V2.0

Emiliano Gonzalez

Pedro Coronel

Asunción – Paraguay

2013

Controles de selección

Adaptadores en Android (*adapters*)

Un adaptador representa algo así como una interfaz común al modelo de datos que existe por detrás de todos los controles de selección. Dicho de otra forma, todos los controles de selección accederán a los datos que contienen a través de un adaptador.
Además de proveer de datos a los controles visuales, el adaptador también será responsable de generar a partir de estos datos las vistas específicas que se mostrarán dentro del control de selección. Por ejemplo, si cada elemento de una lista estuviera formado a su vez por una imagen y varias etiquetas, el responsable de generar y establecer el contenido de todos estos "*sub-elementos*" a partir de los datos será el propio adaptador.
Android proporciona de serie varios tipos de adaptadores sencillos, aunque podemos extender su funcionalidad fácilmente para adaptarlos a nuestras necesidades. Los más comunes son los siguientes:

- `ArrayAdapter`. Es el más sencillo de todos los adaptadores, y provee de datos a un control de selección a partir de un array de objetos de cualquier tipo.
- `SimpleAdapter`. Se utiliza para mapear datos sobre los diferentes controles definidos en un fichero XML de layout.
- `SimpleCursorAdapter`. Se utiliza para mapear las columnas de un cursor sobre los diferentes elementos visuales contenidos en el control de selección.

Para no complicar excesivamente los tutoriales, por ahora nos vamos a conformar con describir la forma de utilizar un `ArrayAdapter` con los diferentes controles de selección disponibles. Más adelante aprenderemos a utilizar el resto de adaptadores en contextos más específicos.
Veamos cómo crear un adaptador de tipo ArrayAdapter para trabajar con un array genérico de java:

```
1  final String[] datos =
2          new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};
3
4  ArrayAdapter<String> adaptador =
5          new ArrayAdapter<String>(this,
6              android.R.layout.simple_spinner_item, datos);
```

Comentemos un poco el código. Sobre la primera línea no hay nada que decir, es tan sólo la definición del array java que contendrá los datos a mostrar en el control, en este caso un array sencillo con cinco cadenas de caracteres. En la segunda línea creamos el adaptador en sí, al que pasamos 3 parámetros:

1. El *contexto*, que normalmente será simplemente una referencia a la actividad donde se crea el adaptador.
2. El ID del *layout* sobre el que se mostrarán los datos del control. En este caso le pasamos el ID de un layout predefinido en Android (`android.R.layout.simple_spinner_item`), formado únicamente por un control `TextView`, pero podríamos pasarle el ID de cualquier layout de nuestro proyecto con cualquier estructura y conjunto de controles, más adelante veremos cómo.
3. El *array* que contiene los datos a mostrar.

   Con esto ya tendríamos creado nuestro adaptador para los datos a mostrar y ya tan sólo nos quedaría asignar este adaptador a nuestro control de selección para que éste mostrase los datos en la aplicación.

Control Spinner

Las listas desplegables en Android se llaman `Spinner`. Funcionan de forma similar al de cualquier control de este tipo, el usuario selecciona la lista, se muestra una especie de lista emergente al usuario con todas las opciones disponibles y al seleccionarse una de ellas ésta queda fijada en el control. Para añadir una lista de este tipo a nuestra aplicación podemos utilizar el código siguiente:

```
1  <Spinner android:id="@+id/CmbOpciones"
2      android:layout_width="fill_parent"
3      android:layout_height="wrap_content" />
```

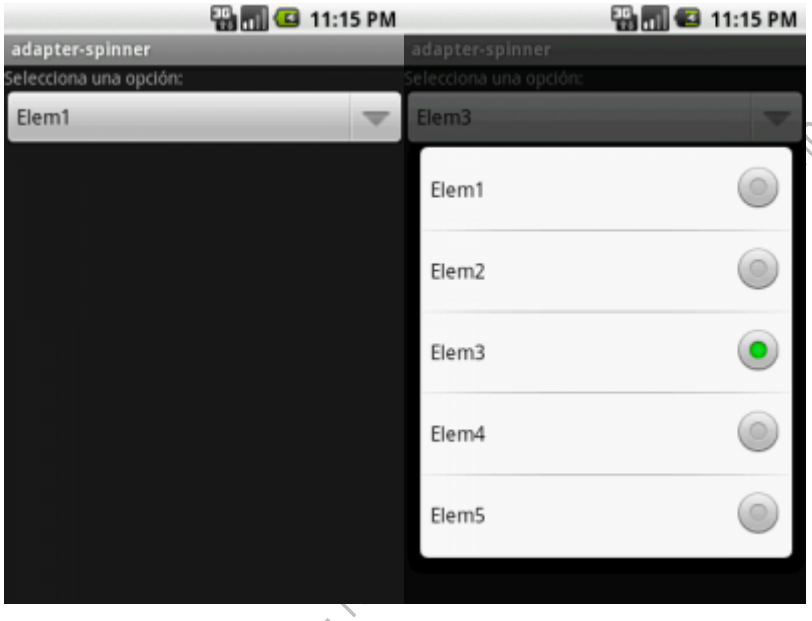
Poco vamos a comentar de aquí ya que lo que nos interesan realmente son los datos a mostrar. En cualquier caso, las opciones para personalizar el aspecto visual del control (fondo, color y tamaño de fuente) son las mismas ya comentadas para los controles básicos.

Para enlazar nuestro adaptador (y por tanto nuestros datos) a este control utilizaremos el siguiente código java:

```
1  final Spinner cmbOpciones = (Spinner)findViewById(R.id.CmbOpciones);
2
3  adaptador.setDropDownViewResource(
4          android.R.layout.simple_spinner_dropdown_item);
5
6  cmbOpciones.setAdapter(adaptador);
```

Comenzamos como siempre por obtener una referencia al control a través de su ID. Y en la última línea asignamos el adaptador al control mediante el método `setAdapter()`. ¿Y la segunda línea para qué es? Cuando indicamos en el apartado anterior cómo construir un adaptador vimos cómo uno de los parámetros que le pasábamos era el ID del layout que utilizaríamos para visualizar los elementos del control. Sin embargo, en el caso del

control `Spinner`, este layout tan sólo se aplicará al elemento seleccionado en la lista, es decir, al que se muestra directamente sobre el propio control cuando no está desplegado. Sin embargo, antes indicamos que el funcionamiento normal del control `Spinner` incluye entre otras cosas mostrar una lista emergente con todas las opciones disponibles. Pues bien, para personalizar también el aspecto de cada elemento en dicha lista emergente tenemos el método `setDropDownViewResource(`*ID_layout*`)`, al que podemos pasar otro ID de layout distinto al primero sobre el que se mostrarán los elementos de la lista emergente. En este caso hemos utilizado otro layout predefinido an Android para las listas desplegables (`android.R.layout.simple_spinner_dropdown_item`).
Con estas simples lineas de código conseguiremos mostrar un control como el que vemos en las siguientes imágenes:



Como se puede observar en las imágenes, la representación del elemento seleccionado (primera imagen) y el de las opciones disponibles (segunda imagen) es distinto, incluyendo el segundo de ellos incluso algún elemento gráfico a la derecha para mostrar el estado de cada opción. Como hemos comentado, esto es debido a la utilización de dos layouts diferentes para uno y otros elementos.

En cuanto a los eventos lanzados por el control `Spinner`, el más comunmente utilizado será el generado al seleccionarse una opción de la lista desplegable, `onItemSelected`. Para capturar este evento se procederá de forma similar a lo ya visto para otros controles anteriormente, asignadole su controlador mediante el método `setOnItemSelectedListener()`:

```
1    cmbOpciones.setOnItemSelectedListener(
2            new AdapterView.OnItemSelectedListener() {
3            public void onItemSelected(AdapterView<?> parent,
```

```
                android.view.View v, int position, long id) {
                    lblMensaje.setText("Seleccionado: " + datos[position]);
            }

            public void onNothingSelected(AdapterView<?> parent) {
                lblMensaje.setText("");
            }
    });
```

Para este evento definimos dos métodos, el primero de ellos (`onItemSelected`) que será llamado cada vez que se selecciones una opción en la lista desplegable, y el segundo (`onNothingSelected`) que se llamará cuando no haya ninguna opción seleccionada (esto puede ocurrir por ejemplo si el adaptador no tiene datos).

ListView.

Un control `ListView` muestra al usuario una lista de opciones seleccionables directamente sobre el propio control, sin listas emergentes como en el caso del control `Spinner`. En caso de existir más opciones de las que se pueden mostrar sobre el control se podrá por supuesto hacer `scroll` sobre la lista para acceder al resto de elementos.

Para empezar, veamos como podemos añadir un control ListView a nuestra interfaz de usuario:

```
<ListView android:id="@+id/LstOpciones"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

Una vez más, podremos modificar el aspecto del control utilizando las propiedades de fuente y color. Definiremos primero un array con nuestros datos de prueba, crearemos posteriormente el adaptador de tipo `ArrayAdapter` y lo asignaremos finalmente al control mediante el método `setAdapter()`:

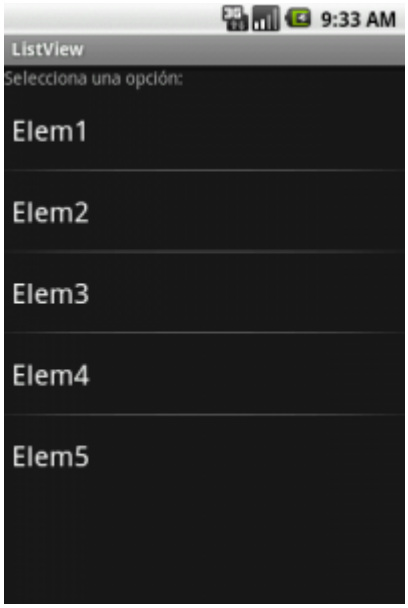
```
final String[] datos =
    new String[]{"Elem1","Elem2","Elem3","Elem4","Elem5"};

ArrayAdapter<String> adaptador =
    new ArrayAdapter<String>(this,
        android.R.layout.simple_list_item_1, datos);

ListView lstOpciones = (ListView)findViewById(R.id.LstOpciones);

lstOpciones.setAdapter(adaptador);
```

En este caso, para mostrar los datos de cada elemento hemos utilizado otro layout genérico de Android para los controles de tipo `ListView` (`android.R.layout.simple_list_item_1`), formado únicamente por un `TextView` con unas dimensiones determinadas. La lista creada quedaría como se muestra en la imagen siguiente:



Como se puede comprobar el uso básico del control `ListView` es completamente análogo al ya comentado para el control `Spinner`.

Si quisiéramos realizar cualquier acción al pulsarse sobre un elemento de la lista creada tendremos que implementar el evento `onItemClick`. Veámos cómo con un ejemplo:

```
lstOpciones.setOnItemClickListener(new OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> a, View v, int position, long id) {
        //Acciones necesarias al hacer click
    }
});
```

Hasta aquí todo sencillo. Pero, ¿y si necesitamos mostrar datos más complejos en la lista? ¿qué ocurre si necesitamos que cada elemento de la lista esté formado a su vez por varios elementos? Pues vamos a provechar este artículo dedicado a los `ListView` para ver cómo podríamos conseguirlo, aunque todo lo que comentaré es extensible a otros controles de selección.

Para no complicar mucho el tema vamos a hacer que cada elemento de la lista muestre por ejemplo dos líneas de texto a modo de título y subtítulo con formatos diferentes (por supuesto se podrían añadir muchos más elementos, por ejemplo imágenes, *checkboxes*, etc).

En primer lugar vamos a crear una nueva clase java para contener nuestros datos de prueba. Vamos a llamarla `Titular` y tan sólo va a contener dos atributos, título y subtítulo.

```
package py.com.cursoandroid;

public class Titular
{
    private String titulo;
    private String subtitulo;
```

```
7
8       public Titular(String tit, String sub){
9           titulo = tit;
10          subtitulo = sub;
11      }
12
13      public String getTitulo(){
14          return titulo;
15      }
16
17      public String getSubtitulo(){
18          return subtitulo;
19      }
20  }
```

En cada elemento de la lista queremos mostrar ambos datos, por lo que el siguiente paso será crear un layout XML con la estructura que deseemos. En mi caso voy a mostrarlos en dos etiquetas de texto (`TextView`), la primera de ellas en negrita y con un tamaño de letra un poco mayor. Llamaremos a este layout "*listitem_titular.xml*":

```
1   <?xml version="1.0" encoding="utf-8"?>
2
3   <LinearLayout
4     xmlns:android="http://schemas.android.com/apk/res/android"
5     android:layout_width="wrap_content"
6     android:layout_height="wrap_content"
7     android:orientation="vertical">
8
9   <TextView android:id="@+id/LblTitulo"
10      android:layout_width="fill_parent"
11      android:layout_height="wrap_content"
12      android:textStyle="bold"
13      android:textSize="20px" />
14
15  <TextView android:id="@+id/LblSubTitulo"
16      android:layout_width="fill_parent"
17      android:layout_height="wrap_content"
18      android:textStyle="normal"
19      android:textSize="12px" />
20
21  </LinearLayout>
```

Ahora que ya tenemos creados tanto el soporte para nuestros datos como el layout que necesitamos para visualizarlos, lo siguiente que debemos hacer será indicarle al adaptador cómo debe utilizar ambas cosas para generar nuestra interfaz de usuario final. Para ello vamos a crear nuestro propio adaptador extendiendo de la clase `ArrayAdapter`.

```
1   class AdaptadorTitulares extends ArrayAdapter {
2
3
```

```
4         Activity context;
5
6             AdaptadorTitulares(Activity context) {
7                 super(context, R.layout.listitem_titular, datos);
8                 this.context = context;
9             }
10
11            public View getView(int position, View convertView, ViewGroup parent) {
12            LayoutInflater inflater = context.getLayoutInflater();
13            View item = inflater.inflate(R.layout.listitem_titular, null);
14
15            TextView lblTitulo = (TextView)item.findViewById(R.id.LblTitulo);
16            lblTitulo.setText(datos[position].getTitulo());
17
18            TextView lblSubtitulo = (TextView)item.findViewById(R.id.LblSubTitulo);
19            lblSubtitulo.setText(datos[position].getSubtitulo());
20
21            return(item);
22        }
      }
```

Analicemos el código anterior. Lo primero que encontramos es el constructor para nuestro adaptador, al que sólo pasaremos el contexto (que será la actividad desde la que se crea el adaptador). En este constructor tan sólo guardaremos el contexto para nuestro uso posterior y llamaremos al constructor padre tal como ya vimos al principio de este artículo, pasándole el ID del layout que queremos utilizar (en nuestro caso el nuevo que hemos creado, "listitem_titular") y el array que contiene los datos a mostrar.

Posteriormente, redefinimos el método encargado de generar y rellenar con nuestros datos todos los controles necesarios de la interfaz gráfica de cada elemento de la lista. Este método es `getView()`.
El método getView() se llamará cada vez que haya que mostrar un elemento de la lista. Lo primero que debe hacer es "*inflar*" el layout XML que hemos creado. Esto consiste en consultar el XML de nuestro layout y crear e inicializar la estructura de objetos java equivalente. Para ello, crearemos un nuevo objeto `LayoutInflater` y generaremos la estructura de objetos mediante su método `inflate(id_layout)`.
Tras esto, tan sólo tendremos que obtener la referencia a cada una de nuestras etiquetas como siempre lo hemos hecho y asignar su texto correspondiente según los datos de nuestro array y la posición del elemento actual (parámetro `position` del método `getView()`).
Una vez tenemos definido el comportamiento de nuestro adaptador la forma de proceder en la actividad principal será análoga a lo ya comentado, definiremos el array de datos de prueba, crearemos el adaptador y lo asignaremos al control mediante `setAdapter()`:

```
datos =
    new Titular[]{
        new Titular("Título 1", "Subtítulo largo 1"),
        new Titular("Título 2", "Subtítulo largo 2"),
        new Titular("Título 3", "Subtítulo largo 3"),
        new Titular("Título 4", "Subtítulo largo 4"),
        new Titular("Título 5", "Subtítulo largo 5")};

//...
//...

AdaptadorTitulares adaptador =
        new AdaptadorTitulares(this);

ListView lstOpciones = (ListView)findViewById(R.id.LstOpciones);

lstOpciones.setAdapter(adaptador);
```

Hecho esto, y si todo ha ido bien, nuestra nueva lista debería quedar como vemos en la imagen siguiente:

GridView

El control `GridView` de Android presenta al usuario un conjunto de opciones seleccionables distribuidas de forma tabular, o dicho de otra forma, divididas en filas y columnas. Dada la naturaleza del control ya podéis imaginar sus propiedades más importantes, que paso a enumerar a continuación:

- `android:numColumns`, indica el número de columnas de la tabla o "`auto_fit`" si queremos que sea calculado por el propio sistema operativo a partir de las siguientes propiedades.
- `android:columnWidth`, indica el ancho de las columnas de la tabla.

- `android:horizontalSpacing`, indica el espacio horizontal entre celdas.
- `android:verticalSpacing`, indica el espacio vertical entre celdas.
- `android:stretchMode`, indica qué hacer con el espacio horizontal sobrante. Si se establece al valor "`columnWidth`" este espacio será absorbido a partes iguales por las columnas de la tabla. Si por el contrario se establece a "`spacingWidth`" será absorbido a partes iguales por los espacios entre celdas.

  Veamos cómo definiríamos un `GridView` de ejemplo en nuestra aplicación:

```
<GridView android:id="@+id/GridOpciones"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:numColumns="auto_fit"
    android:columnWidth="80px"
    android:horizontalSpacing="5px"
    android:verticalSpacing="10px"
    android:stretchMode="columnWidth" />
```

Una vez definida la interfaz de usuario, la forma de asignar los datos desde el código de la aplicación es completamente análoga a la ya comentada tanto para las listas desplegables como para las listas estáticas: creamos un array genérico que contenga nuestros datos de prueba, declaramos un adaptador de tipo ArrayAdapter pasándole en este caso un layout genérico (`simple_list_item_1`, compuesto por un simple `TextView`) y asociamos el adaptador al control `GridView` mediante su método `setAdapter()`:

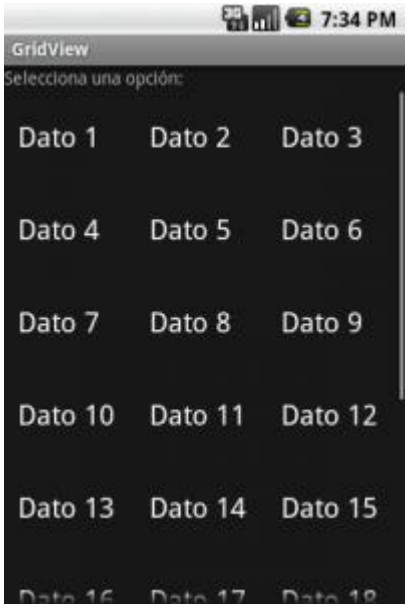
```
private String[] datos = new String[25];
//...
for(int i=1; i<=25; i++)
        datos[i-1] = "Dato " + i;

ArrayAdapter<String> adaptador =
        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, datos

final GridView grdOpciones = (GridView)findViewById(R.id.GridOpciones);

grdOpciones.setAdapter(adaptador);
```

Por defecto, los datos del array se añadirán al control `GridView` ordenados por filas, y por supuesto, si no caben todos en la pantalla se podrá hacer *scroll* sobre la tabla. Vemos en una imagen cómo queda nuestra aplicación de prueba:



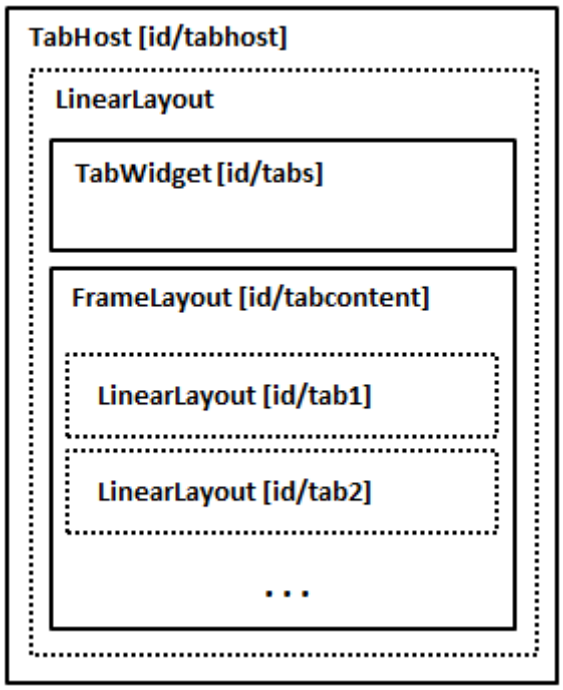
En cuanto a los eventos disponibles, el más interesante vuelve a ser el lanzado al seleccionarse una celda determinada de la tabla: `onItemSelected`. Este evento podemos capturarlo de la misma forma que hacíamos con los controles `Spinner` y `ListView`. Veamos un ejemplo de cómo hacerlo:

```
1  grdOpciones.setOnItemSelectedListener(
2      new AdapterView.OnItemSelectedListener() {
3              public void onItemSelected(AdapterView<?> parent,
4                  android.view.View v, int position, long id) {
5                      lblMensaje.setText("Seleccionado: " + datos[position]);
6              }
7
8              public void onNothingSelected(AdapterView<?> parent) {
9                  lblMensaje.setText("");
10             }
11 });
```

Tab Layout

En Android, el elemento principal de un conjunto de pestañas será el control `TabHost`. Éste va a ser el contenedor principal de nuestro conjunto de pestañas y deberá tener obligatoriamente como id el valor "`@android:id/tabhost`". Dentro de éste vamos a incluir un `LinearLayout` que nos servirá para distribuir verticalmente las secciones principales del layout: la sección de pestañas en la parte superior y la sección de contenido en la parte inferior. La sección de pestañas se representará mediante un elemento `TabWidget`, que deberá tener como id el valor "`@android:id/tabs`", y como contenedor para el contenido de las pestañas añadiremos un `FrameLayout` con el id obligatorio "`@android:id/tabcontent`". Por último, dentro del `FrameLayout` incluiremos el contenido de cada pestaña, normalmente cada uno dentro de su propio layout principal (en este caso se ha utilizado `LinearLayout`) y con un id

único que nos permita posteriormente hacer referencia a ellos fácilmente (en este caso se ha utilizado por ejemplo los ids "`tab1`", "`tab2`", …). A continuación se representa de forma gráfica toda la estructura descrita.



Si traducimos esta estructura a nuestro fichero de layout XML tendríamos lo siguiente:

```
1   <TabHost android:id="@android:id/tabhost"
2       android:layout_width="match_parent"
3       android:layout_height="match_parent">
4
5       <LinearLayout
6           android:orientation="vertical"
7           android:layout_width="fill_parent"
8           android:layout_height="fill_parent" >
9
10              <TabWidget android:layout_width="match_parent"
11                      android:layout_height="wrap_content"
12                      android:id="@android:id/tabs" />
13
14              <FrameLayout android:layout_width="match_parent"
15                      android:layout_height="match_parent"
16                      android:id="@android:id/tabcontent" >
17
18                  <LinearLayout android:id="@+id/tab1"
19                      android:orientation="vertical"
20                      android:layout_width="match_parent"
21                      android:layout_height="match_parent" >
22                      <TextView android:id="@+id/textView1"
23                          android:text="Contenido Tab 1"
24                          android:layout_width="wrap_content"
25                          android:layout_height="wrap_content" />
```

```
26          </LinearLayout>
27
28              <LinearLayout android:id="@+id/tab2"
29                  android:orientation="vertical"
30                  android:layout_width="match_parent"
31                  android:layout_height="match_parent" >
32                  <TextView android:id="@+id/textView2"
33                      android:text="Contenido Tab 2"
34                      android:layout_width="wrap_content"
35                      android:layout_height="wrap_content" />
36          </LinearLayout>
37
38           </FrameLayout>
39      </LinearLayout>
40
41  </TabHost>
```

Como se puede ver, como contenido de las pestañas tan sólo se ha añadido por simplicidad una etiqueta de texto con el texto "*Contenido Tab NºTab*". Esto nos permitirá ver que el conjunto de pestañas funciona correctamente cuando ejecutemos la aplicación. Con esto ya tendríamos montada toda la estructura de controles necesaria para nuestra interfaz de pestañas. Sin embargo, como ya dijimos al principio del artículo, con esto no es suficiente. Necesitamos asociar de alguna forma cada pestaña con su contenido, de forma que el control se comporte correctamente cuando cambiamos de pestaña. Y esto tendremos que hacerlo mediante código en nuestra actividad principal.

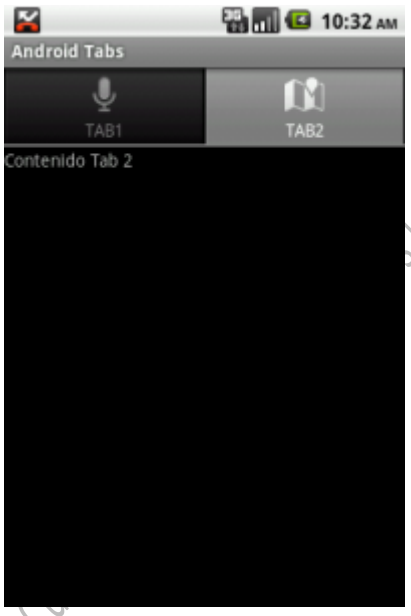
Empezaremos obteniendo una referencia al control principal `TabHost` y preparándolo para su configuración llamando a su método `setup()`. Tras esto, crearemos un objeto de tipo `TabSpec` para cada una de las pestañas que queramos añadir mediante el método `newTabSpec()`, al que pasaremos como parámetro una etiqueta identificativa de la pestaña (en mi caso de ejemplo "*mitab1*", "*mitab2*", …). Además, también le asignaremos el layout de contenido correspondiente a la pestaña llamando al método `setContent()`, e indicaremos el texto y el icono que queremos mostrar en la pestaña mediante el método `setIndicator(texto, icono)`. Veamos el código completo.


```
1   Resources res = getResources();
2
3   TabHost tabs=(TabHost)findViewById(android.R.id.tabhost);
4   tabs.setup();
5
6   TabHost.TabSpec spec=tabs.newTabSpec("mitab1");
7   spec.setContent(R.id.tab1);
8   spec.setIndicator("TAB1",
9           res.getDrawable(android.R.drawable.ic_btn_speak_now));
10  tabs.addTab(spec);
```

```
11
12    spec=tabs.newTabSpec("mitab2");
13    spec.setContent(R.id.tab2);
14    spec.setIndicator("TAB2",
15            res.getDrawable(android.R.drawable.ic_dialog_map));
16    tabs.addTab(spec);
17
18    tabs.setCurrentTab(0);
```

Si vemos el código, vemos por ejemplo como para la primera pestaña creamos un objeto `TabSpec` con la etiqueta "*mitab1*", le asignamos como contenido uno de los `LinearLayout` que incluimos en la sección de contenido (en este caso `R.id.tab1`) y finalmente le asignamos el texto "*TAB1*" y el icono `android.R.drawable.ic_btn_speak_now` (Éste es un icono incluido con la propia plataforma Android. Si no existiera en vuestra versión podéis sustituirlo por cualquier otro icono). Finalmente añadimos la nueva pestaña al control mediante el método `addTab()`.

Si ejecutamos ahora la aplicación tendremos algo como lo que muestra la siguiente imagen, donde podremos cambiar de pestaña y comprobar cómo se muestra correctamente el contenido de la misma.



En cuanto a los eventos disponibles del control `TabHost`, aunque no suele ser necesario capturarlos, podemos ver a modo de ejemplo el más interesante de ellos, `OnTabChanged`, que se lanza cada vez que se cambia de pestaña y que nos informa de la nueva pestaña visualizada. Este evento podemos implementarlo y asignarlo mediante el método `setOnTabChangedListener()` de la siguiente forma:

```
1   tabs.setOnTabChangedListener(new OnTabChangeListener() {
2       @Override
3       public void onTabChanged(String tabId) {
```

```
4            Log.i("AndroidTabsDemo", "Pulsada pestaña: " + tabId);
5        }
6    });
```

En el método `onTabChanged()` recibimos como parámetro la etiqueta identificativa de la pestaña (no su ID), que debimos asignar cuando creamos su objeto `TabSpec` correspondiente. Para este ejemplo, lo único que haremos al detectar un cambio de pestaña será escribir en el log de la aplicación un mensaje informativo con la etiqueta de la nueva pestaña visualizada. Así por ejemplo, al cambiar a la segunda pestaña recibiremos el mensaje de log: "*Pulsada pestaña: mitab2*".