Capítulo 8
Listas

Práctica 8.1: Hola mundo

Para ir calentando, empezaremos con una práctica sencilla en la que haremos una actividad con un botón que, al ser pulsado, mostrará un mensaje (toast) saludando.

1. Lanza el entorno de desarrollo y crea un proyecto nuevo:

   - Nombre de la aplicación: Hola mundo
   - Dominio de la compañía: DM2E
   - Mínimo SDK: API 15
   - Una blank activity, con la configuración por defecto.

2. Revisa el código que el asistente ha creado por nosotros y asegúrate de que lo entiendes todo.
3. Lanza la aplicación en un AVD o en tu móvil.

La actividad predefinida no es muy emocionante. Vamos a cambiarla para añadir un poco de interactividad:

1. Elimina la etiqueta del layout.
2. Añade el botón:

   ```
   <Button android:text="@string/pulsame"
           android:layout_width="match_parent"
           android:layout_height="match_parent" />
   ```

3. La cadena @string/hello_world (de la etiqueta) ya no se utiliza. Elimínala de strings.xml.
4. Añade una nueva cadena con identificador pulsame y texto _¡Púlsame!_
5. Lanza la aplicación y comprueba el cambio.
6. De momento, el botón no hace nada. Añade un evento:

   ```
   <Button ...
           android:onClick="onPulsame" />
   ```

7. Añade en la clase de la actividad el método que recibirá la notificación.

   ```
   public void onPulsame(View view) {
           Toast toast = Toast.makeText(
           getApplicationContext(),
           R.string.saludo,
           Toast.LENGTH_SHORT);
           toast.show();
   }
   ```

8. Añade en strings.xml una nueva cadena con el identificador saludo y texto ¡Hola, mundo!.

9. Lanza la aplicación y comprueba que, al pulsar el botón, aparece el mensaje.

Puedes "limpiar" la aplicación eliminando el menú que añadió el asistente y que no estamos utilizando para nada:

1. Elimina de la clase principal los métodos onCreateOptionsMenu() y onOptionsItemSelected() (y retira los import's que ya no se necesiten).
2. Elimina el recurso del menú, menu_main.xml.
3. Elimina la cadena action_settings de strings.xml.

Práctica 8.2: Cambiando el color

Utilizando el método factoría Toast.makeText() conseguimos un objeto Toast cómodamente. Podemos modificar la posición de la notificación usando los métodos setGravity() y setMargin() antes de mostrarla, e incluso refinar el texto con setText().

Si queremos hacer cambios más radicales, tendremos que acceder a la vista (widget) interna del toast y manipularla. Por ejemplo, para que el texto salga de color rojo, antes de hacer visible la notificación podemos obtener su vista interna, acceder a la etiqueta y cambiarla el color.

1. Haz una copia del proyecto.
2. Añade, antes de la línea toast.show() el siguiente código:

```
View v = toast.getView();
TextView tv = (TextView) v.findViewById(android.R.id.message);
tv.setTextColor(Color.RED);
```

Práctica 8.3: Reloj emergente

En la práctica anterior, utilizamos el método factoría para conseguir un toast inicializado, y accedimos a su contenido para manipularlo.

En lugar de eso, podemos construir manualmente el toast (con new) y establecerle por código la vista que queramos que contenga. La clase Toast se encargará de superponer nuestra vista sobre la actividad actual, y de hacerla desaparecer un tiempo después.

En esta práctica, pondremos como vista el widget AnalogClock, que pinta un reloj de agujas.

1. Haz una copia del proyecto de la práctica anterior.
2. Renombra el paquete principal a DM2E.relojemergente.
3. Modifica la cadena app_name y pon Reloj emergente. Borra la cadena saludo, que ya no utilizaremos.
4. Quita el código del evento asociado al botón y crea y configura la notificación manualmente. En concreto, construye un nuevo objeto con new, y luego establecele como vista un android.widget.AnalogClock.

```
public void onPulsame(View view) {
        Context c = getApplicationContext();
```

```
            Toast toast = new Toast(c);
            toast.setDuration(Toast.LENGTH_LONG);
            AnalogClock clock = new AnalogClock(c);
            toast.setView(clock);
            toast.show();
    }
```

Práctica 8.4: Mensaje de alerta desde un layout

Si quisiéramos poner notificaciones más elaboradas (por ejemplo, un icono y un texto), podríamos construir vistas más complejas igual que hemos hecho en la práctica anterior. Sin embargo, construir el interfaz por código va en contra de la filosofía de desarrollo para Android. Cuando sea posible, es preferible utilizar ficheros de recursos de tipo layout para ello. En esta práctica, crearemos un layout que no estará directamente asociado a una actividad, sino que lo usaremos para construir a partir de él el contenido del toast.

1. Haz una copia del proyecto de la práctica 8.1 y renombra el paquete principal a DM2E.toastconlayout.
2. Cambia el título de la actividad principal a Toast con layout.
3. Crea un nuevo fichero de layout:

    - Llámalo mitoast.xml
    - Usa un LinearLayout.
    - Añade una imagen. Para no tener que buscar una y añadirla en el proyecto, utilizaremos una de las predefinidas de Android, @android:drawable/ic_dialog_info.
    - Añade una etiqueta con la cadena @string/saludo que definimos en la práctica original.
    - Ponle a la imagen un pequeño margen a la derecha para que se separe del texto (8dp bastarán).

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal"
android:layout_width="match_parent"
android:layout_height="match_parent">
<ImageView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginRight="8dp"
android:src="@android:drawable/ic_dialog_info"
/>
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="@string/saludo"
android:layout_gravity="center_vertical"
/>
</LinearLayout>
```

4. Modifica el código del método onPulsame() para que se cree una notificación manualmente, como hicimos en la práctica 8.3.

5. Obtén el objeto que es capaz de generar una vista a partir del identificador de un layout.
6. Pídele que construya la vista a partir del layout anterior.
7. Establécelo como vista del toast que acabas de crear
8. Muestra el toast.

```
public void onPulsame(View view) {
        Context c = getApplicationContext();
        Toast toast = new Toast(c);
        toast.setDuration(Toast.LENGTH_LONG);
        LayoutInflater inflater = getLayoutInflater();
        View v = inflater.inflate(R.layout.mitoast, null, false);
        toast.setView(v);
        toast.show();
}
```

1. Modifica el layout y añade un AnalogClock.
2. Lanza la aplicación.
3. Pulsa el botón y cierra la aplicación antes de que el toast desaparezca.
4. ¿Qué ocurre? ¿Por qué? ¿Cómo lo solucionarías?

Práctica 8.5: Lista básica de elementos

Las listas resultan ser un control muy habitual en las aplicaciones de Android, pero requieren una programación algo más compleja de lo habitual. Hay varias cuestiones que hay que tener en cuenta:

- El objetivo de las listas es mostrar múltiples elementos ("items").
- Cada elemento se muestra a través de un componte visual diferente. Por tanto, las listas son contenedores de otros controles: un botón es un widget independiente, pero una lista tendrá dentro a otros.
- La visualización de cada elemento de la lista puede requerir más de un elemento, mezclando así múltiples etiquetas, imágenes, etcétera.
- En un determinado momento, no todos los items serán visibles, por lo que podríamos querer que la creación de los componentes de cada item no se construyan hasta que no sea imprescindible.
- Los items de una lista pueden provenir de datos con estructuras complejas (más allá de un mero texto) por lo que se desea independizar el componente visual (ListView) de los propios datos y su origen.

Todo lo anterior hace que, formalmente, las ListView se consideren layouts, que son construídas a partir de lo que Android denomina adaptadores. Un adaptador es un objeto que implementa un determinado interfaz (Adapter), y a través del cual la ListView es capaz de obtener cada uno de los componentes gráficos que debe mostrar. El método más importante es getView(), que devuelve la vista (control) que mostrará el elemento i-ésimo de la vista.

Android proporciona varios adaptadores. En esta práctica, utilizaremos ArrayAdapter que recibe un array de objetos de tipo T (es una clase genérica) y para cada uno construye un TextView en el que muestra el resultado de ejecutar toString() sobre él. Por tanto, al "conectar" un ListView con un ArrayAdapter veremos la lista de los elementos del array que le hayamos dado al adaptador.

1. Crea un nuevo proyecto y llámalo ListViewBasico. Como título pon ListView básico, y ponlo en el paquete DM2E.listviewbasico.
2. Modifica el layout principal para eliminar la etiqueta añadida por el asistente y pon como único widget un ListView. Fíjate que aquí no establecemos su contenido.

```
<RelativeLayout ... >
<ListView android:id="@+id/listView"
android:layout_width="match_parent"
android:layout_height="match_parent" />
</RelativeLayout>
```

3. La clase ArrayAdapter necesita saber cómo queremos que se muestre la etiqueta que creará para cada item. Espera que lo hagamos indicando un layout que esté formado por un TextView. En el método getView() "inflará" el layout y establecerá el texto del elemento correspondiente.

Crea un layout de nombre basiclistitemview con un TextView:

```
<TextView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
</TextView>
```

4. En el onCreate(), tras establecer la vista con el layout:

    a) Crea un array de 100 Integer, y añade los números del 1 al 10.
    b) Crea un ArrayAdapter<Integer>, y pásale en el constructor el array anterior.
    c) Busca la ListView en la actividad.
    d) Establece como adaptador el ArrayAdapter creado.

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Integer[] datos = new Integer[100];
        for (int i = 0; i < 100; ++i) {
                datos[i] = i + 1;
        }
        ArrayAdapter<Integer> aa;
        aa = new ArrayAdapter<Integer>(this,
                R.layout.basiclistitemview, datos);
        ListView lv = (ListView) findViewById(R.id.listView);
        lv.setAdapter(aa);
} // onCreate
```

5. Prueba la aplicación.

El constructor de ArrayAdapter recibe tres parámetros:

1. El objeto que se usará como contexto para la creación de las vistas.
2. El layout que se “inflará” para cada item. Debe ser un TextView.

3. El array de elementos de tipo T (o, alternativamente, un List<T>) del que se obtendrán los items.

Observa que los elementos aparecen muy juntos, y su tamaño no sería práctico si quisiéramos permitir al usuario que seleccionara alguno. Además, tener que crear el layout incluso para unos items tan simples como un mero texto resulta laborioso.

Android proporciona un layout predefinido que ahorra la creación del layout, y que además deja espacio suficiente para poder seleccionar cada elemento en función del tipo y configuración del dispositivo.

1. Modifica la llamada al constructor del ArrayAdapter y en el segundo parámetro pasa android.R.layout.simple_list_item_1. Es un layout predefinido que contiene un TextView configurado para dejar margen suficiente como para poder seleccionarse.
2. Elimina el layout que hiciste manualmente, basiclistitemview.
3. Lanza la aplicación y observa las diferencias.

Práctica 8.6: ListView con dos etiquetas

Como hemos visto, Android proporciona layout predefinidos[1]. En esta ocasión, usaremos android.R.layout.simple_list_item_2, que incluye dos etiquetas.

1. Haz una copia de la práctica anterior:

   a) Llámala ListViewConDosEtiquetas.
   b) Renombra el paquete a DM2E.listviewdosetiquetas.

2. Modifica la llamada al constructor de ArrayAdapter para que utilice el nuevo layout predefinido.

3. Ejecuta la práctica. ¿Qué ocurre?

El layout simple_list_item_2 es un RelativeLayout que tiene dentro los dos TextView. Cuando el ArrayAdapter "infla" el layout para mostrar un item, no se encuentra un TextView, sino el RelativeLayout, y no puede establecerle el texto, haciendo saltar una excepción.

1. En la invocación al constructor del ArrayAdapter, antes del último parámetro con el array pasa como parámetro android.R.id.text1.
2. Ejecuta la práctica de nuevo. Observa que vuelve a funcionar, pero el aspecto es ligeramente diferente respecto a la práctica anterior.

Como es la clase ArrayAdapter quien se preocupa de crear la vista de cada item, aunque hayamos pasado un layout que es capaz de mostrar dos etiquetas no podemos a priori especificar qué queremos que se muestre en la segunda.

Para conseguirlo, necesitaremos hacer una subclase de ArrayAdapter, sobreescribir el método getView() y construir nosotros la vista del item.

1. Crea una clase Libro que tenga dos atributos de tipo String, titulo y autor.

[1] Puedes verlos en http://developer.android.com/reference/android/R.layout.html

```
class Libro {
   public Libro(String t, String a) {
      titulo = t;
      autor = a;
   }
   public String titulo;
   public String autor;
} // class Libro
```

2. Crea una clase MiArrayAdapter que herede de ArrayAdapter<Libro>:

   a) Crea dos constructores que reciban un contexto, y o bien un array de libros, o bien un java.util.List.
   b) Sobreescribe el método getView(). En él, construye una nueva vista a partir del layout predefinido, obtén el libro que hay que mostrar en la posición solicitada, y accede a las etiquetas del layout para poner el título y el autor (android.R.id.text1 y text2).

```
class MiArrayAdapter extends ArrayAdapter<Libro> {
public MiArrayAdapter(Context context, Libro[] libros) {
super(context, 0, libros);
}
public MiArrayAdapter(Context context,
java.util.List<Libro> libros) {
super(context, 0, libros);
}
@Override
public View getView(int position, View convertView,
ViewGroup parent) {
Libro l = getItem(position);
View v;
v = LayoutInflater.from(getContext()).inflate(
android.R.layout.simple_list_item_2,
parent,
false);
TextView tv;
tv = (TextView) v.findViewById(android.R.id.text1);
tv.setText(l.titulo);
if (l.autor != null) {
tv = (TextView) v.findViewById(android.R.id.text2);
tv.setText(l.autor);
}
return v;
} // getView
} // class MiArrayAdapter
```

3. Modifica el onCreate() para crear una lista con varios libros, y asociárselo a un MiArrayAdapter que asocies al ListView.

```
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        java.util.ArrayList<Libro> libros;
        libros = new java.util.ArrayList<Libro>();
        libros.add(new Libro("Don Quijote de la Mancha",
                "Miguel de Cervantes"));
        libros.add(new Libro("La Celestina", "Fernando de Rojas"));
        libros.add(new Libro("Rinconete y Cortadillo",
                "Miguel de Cervantes"));
        libros.add(new Libro("El Lazarillo de Tormes", null));
        libros.add(new Libro("La Galatea", "Miguel de Cervantes"));
        MiArrayAdapter aa;
        aa = new MiArrayAdapter(this, libros);
        ListView lv = (ListView) findViewById(R.id.listView);
        lv.setAdapter(aa);
} // onCreate
```

En el getView() hemos ignorado el segundo parámetro, convertView. Internamente, ListView puede reutilizar vistas. Si detecta que la vista que se construyó para un determinado item ya no es necesaria, y necesita una vista nueva para un item diferente, puede llamar a getView() pasando como segundo parámetro la vista antigua para ser "reciclada" con el nuevo item. Por eficiencia, es conveniente por tanto comprobar si el segundo parámetro no es null. En ese caso, en lugar de crear una vista nueva ("inflándola" desde el layout) se debería utilizar esa, accediendo a sus elementos y manipulando su contenido. Al final, devolveremos como resultado del método el mismo objeto que hemos recibido en el segundo parámetro.

```
public View getView(int position, View convertView,
ViewGroup parent) {
Libro l = getItem(position);
View v;
if (convertView == null)
v = LayoutInflater.from(getContext()).inflate(
android.R.layout.simple_list_item_2,
parent,
false);
else
v = convertView;
TextView tv;
tv = (TextView) v.findViewById(android.R.id.text1);
tv.setText(l.titulo);
tv = (TextView) v.findViewById(android.R.id.text2);
if (l.autor != null)
tv.setText(l.autor);
else
tv.setText("");
return v;
} // getView
```

Práctica 8.7: El patrón View Holder

En la práctica anterior, nos preocupamos de reciclar la vista anterior que pudiera habernos llegado en el segundo parámetro de getView(). Sin embargo, tenemos que seguir utilizando findViewById() para acceder a sus elementos internos. Una solución es utilizar el patrón ViewHolder[2]. Consiste en aprovechar que la clase View es capaz de guardar un objeto arbitrario (con setTag() y getTag() que pueda ser utilizado a discreción por el programador. La idea es crear una nueva clase ViewHolder en el que guardemos los dos TextView. Cuando nos veamos obligados a crear una nueva vista, buscaremos los TextView con findViewById() y los guardaremos en un objeto de nuestra clase ViewHolder que conservaremos en la vista a través del setTag(). En futuras invocaciones a getView(), si convertView no es null, en lugar de utilizar el lento findViewById() accederemos con getTag() al ViewHolder y recogeremos de ahí los elementos. A costa de un pequeño consumo adicional de memoria mejoramos el rendimiento sensiblemente.

1. Haz una copia de la práctica anterior.
2. Añade una clase interna a MiArrayAdapter:

    a) Llámala ViewHolder.
    b) Añade dos atributos públicos de tipo TextView y llámalos titulo y autor.

```
class ViewHolder {
   public TextView titulo;
   public TextView autor;
}
```

3. Modifica el método getView() para que:

    a) Si convertView es null, se cree un objeto ViewHolder, se inicialice con los TextView de la nueva vista recién creada y se guarde en ella con setTag().
    b) Si se está reciclando una vista anterior, se acceda con getTag() al objeto ViewHolder y se evite así buscar los TextView por id.

```
public View getView(int position, View convertView,
ViewGroup parent) {
Libro l = getItem(position);
View v;
ViewHolder vh;
if (convertView == null) {
v = LayoutInflater.from(getContext()).inflate(
android.R.layout.simple_list_item_2,
parent,
false);
vh = new ViewHolder();
vh.titulo = (TextView) v.findViewById(android.R.id.text1);
vh.autor = (TextView) v.findViewById(android.R.id.text2);
```

[2] http://developer.android.com/training/improving-layouts/smooth-scrolling.html#ViewHolder
http://arteksoftware.com/androids-built-in-list-item-layouts/

```
v.setTag(vh);
}
else {
v = convertView;
vh = (ViewHolder) v.getTag();
}
vh.titulo.setText(l.titulo);
if (l.autor != null)
vh.autor.setText(l.autor);
else
vh.autor.setText("");
return v;
} // getView
```

Práctica 8.8: Diferentes tipos de item

En la práctica anterior, los libros anónimos utilizan una vista en la que hay dos etiquetas, aunque una, la del autor, no se utilice. Podríamos querer usar para ellos la vista con una única etiqueta que usamos en la práctica 8.5. En ese caso, al crear la vista del item en el getView() decidiremos si "inflamos" el layout android.R.layout.simple_list_item_1 (en los libro anónimos) o el android.R.layout.simple_list_item_2 (en todos los demás).

El problema que surge es que las vistas las reutilizamos. Si Android nos pide que creemos la vista para un libro con autor conocido y nos da una vista para reutilizar, ésta tendrá que ser del tipo simple_list_item_2 o no nos valdrá y se perderá la oportunidad de reciclaje. La solución pasa por indicar a Android que tenemos varios tipos de vista, y permitirle saber de qué tipo es cada una.

En concreto, sobreescribiremos los siguientes métodos del adaptador:

- getViewTypeCount(): devuelve el número de "tipos de vista" que podría mostrar la lista. En nuestro caso devolveremos dos.

- getItemViewType(int position): debe devolver un número entre 0 y getViewTypeCount() - 1 indicando el "tipo de vista" que se utiliza para mostrar el item situado en la posición del parámetro. En nuestro caso, devolveremos 0 para indicar que es una vista normal (de un libro con autor) y 1 si es la de un libro anónimo.

Luego, en getView() tendremos, obviamente, que crear la vista que corresponda cuando convertView sea null. Lo importante es que, cuando no lo sea, sabremos que el tipo de vista que nos han dado es precisamente el que necesitamos. En este caso eso significa que no hay que preocuparse más de borrar la etiqueta del autor cuando vayamos a mostrar un usuario anónimo.

1. Haz una copia de la práctica anterior.
2. Añade a la clase MiArrayAdapter los métodos getViewTypeCount() y getItemViewType() tal y como se ha descrito antes.
3. Modifica getView() para construir el tipo de vista que corresponda dependiendo de si convertView es null. Elimina el else del if donde se establecía el autor, dado que ahora sabremos que la segunda etiqueta nunca existirá cuando el libro sea anónimo.

```
public int getViewTypeCount() {
```

```
return 2;
}
public int getItemViewType(int position) {
return (getItem(position).autor == null)?1:0;
}
public View getView(int position, View convertView,
ViewGroup parent) {
Libro l = getItem(position);
View v;
ViewHolder vh;
if (convertView == null) {
int layoutId;
if (l.autor == null)
layoutId = android.R.layout.simple_list_item_1;
else
layoutId = android.R.layout.simple_list_item_2;
v = LayoutInflater.from(getContext()).inflate(
layoutId,
parent,
false);
vh = new ViewHolder();
vh.titulo = (TextView) v.findViewById(android.R.id.text1);
if (l.autor != null)
vh.autor = (TextView) v.findViewById(android.R.id.text2);
v.setTag(vh);
}
else {
v = convertView;
vh = (ViewHolder) v.getTag();
}
vh.titulo.setText(l.titulo);
//tv = (TextView) v.findViewById(android.R.id.text2);
if (l.autor != null)
vh.autor.setText(l.autor);
// Si el autor es null no tenemos que preocuparnos, porque
// la vista tendrá siempre una única etiqueta.
return v;
} // getView
```

Práctica 8.9: Pulsación sobre elementos

Si en las prácticas anteriores pulsas sobre cualquier elemento verás un pequeño efecto que proporciona feedback, pero nada más.

Es posible añadir código para reaccionar ante las pulsaciones de los eventos. En este caso, se hace a través del ListView, no del adaptador como hemos hecho en las prácticas anteriores. Ten en cuenta que el adaptador sirve para hacer de puente entre las vistas y los datos (un array o una lista, en el caso del ArrayAdapter), pero es responsabilidad del ListView preocuparse de la interacción.

La clase ListView proporciona métodos del tipo setOn*Listener(), para añadir oyentes a diferentes eventos que pueden ocurrir sobre la lista. Los que nos interesan aquí son:

- setOnItemClickListener(): recibe un OnItemClickListener (interfaz interno de AdapterVew, superclase de ListView). El interfaz fuerza a la implementación de un único método, onItemClick(), que recibe información sobre el evento.

- setOnItemLongClickListener(): recibe un OnItemLongClickListener. En este caso el método a implementar se llama onItemLongClick(), que debe devolver un booleano indicando si ha consumido o no el evento.

1. Haz una copia de la práctica anterior.
2. En el onCreate() añade código al final del todo para registrarte de los eventos de OnItemClick y OnItemLongClick de la lista.
3. En ambos casos, haz que se muestre un toast con la posición e identificador del elemento seleccionado.

```
[...]
lv.setAdapter(aa);
lv.setOnItemClickListener(
new AdapterView.OnItemClickListener() {
public void onItemClick(AdapterView<?> parent, View view,
int position, long id) {
Toast.makeText(getApplicationContext(),
"Pulsado " + position + ", " + id,
Toast.LENGTH_SHORT).show();
}
});
lv.setOnItemLongClickListener(
new AdapterView.OnItemLongClickListener() {
public boolean onItemLongClick(AdapterView<?> parent,
View view,
int position, long id) {
Toast.makeText(getApplicationContext(),
"Pulsado largo " + position + ", " + id,
Toast.LENGTH_SHORT).show();
return true;
}
});
[...]
```

¿Ves algo mejorable en este código?

Práctica 8.10: Borrado de elementos

A lo largo de su uso, la lista puede sufrir modificaciones, con elementos que se añaden, se borran, o se modifican. En esta práctica vamos a ver un pequeño ejemplo de borrado. En concreto, borraremos un elemento si hacemos un long click sobre él.

1. Haz una copia de la práctica anterior.
2. Modifica el evento de la pulsación simple para que el toast indique que si se mantiene pulsado se borrará el elemento.

3. Modifica el evento de pulsación larga para que se elimine. Para eso, utiliza el método remove(int) de la lista de libros. Como estamos utilizando una clase anónima, podrás acceder a la variable local del método onCreate(), aunque tendrás que declararla final[3].

```
[...]
lv.setAdapter(aa);
lv.setOnItemClickListener(
new AdapterView.OnItemClickListener() {
public void onItemClick(AdapterView<?> parent, View view,
int position, long id) {
Toast.makeText(getApplicationContext(),
"Mantén pulsado para borrar",
Toast.LENGTH_SHORT).show();
}
});
lv.setOnItemLongClickListener(
new AdapterView.OnItemLongClickListener() {
public boolean onItemLongClick(AdapterView<?> parent,
View view,
int position, long id) {
Libro l = aa.getItem(position);
Toast.makeText(getApplicationContext(),
"Borrado '" + l.titulo +
"' (" + position + ", " + id + ")",
Toast.LENGTH_SHORT).show();
libros.remove(position);
return true;
}
});
[...]
```

4. Ejecuta la aplicación y comprueba que no funciona. ¿Qué crees que está ocurriendo?

5. La ListView no es consciente del cambio en los datos, y por tanto no se actualiza. Tras la eliminación, fuerza la propagación del aviso del cambio desde el adaptador, que tendrás que hacer también final:

```
aa.notifyDataSetChanged();
```

6. Lanza la aplicación y comprueba que ahora sí funciona.
7. Tener que notificar el cambio es propenso a olvidos. Para evitarlo, ArrayAdapter proporciona métodos para manipular la lista de datos subyacente. Elimina las dos últimas líneas y pide al adaptador el borrado:

```
aa.remove(l);
```

Práctica 8.11: Identificadores estables de los elementos

[3] Si se utilizara Java 8, esto último no sería necesario.

En la práctica anterior hemos visto que ante un cambio en el modelo es necesario notificar a la vista que algo ha cambiado. Esa notificación es global: no se especifica qué ha pasado. Por tanto, ListView debe recrear todo su contenido.

Por desgracia, las vistas las obtiene del adaptador a partir de la posición de cada item, posiciones que podrían haber cambiado entre actualizaciones. Por tanto, ListView no tiene, a priori, una forma de hacer un seguimiento de las vistas que ya tiene y de sus items para evitar recrear muchas vistas.

Para conseguirlo, el adaptador tiene un método en el que se pide el identificador de un item. Además, el adaptador puede informar de que los identificadores son estables ante cambios, es decir se devolverá el mismo identificador para un elemento independientemente de su posición. En ese caso, la ListView podría hacer un seguimiento de los identificadores de los elementos y sus vistas, y evitar reconstruir las vistas incluso aunque se hayan producido cambios drásticos.

1. Haz una copia de la práctica anterior.
2. Implementa en el adaptador el método hasStableIds() que devuelva siempre true.
3. Implementa también método getItemId(int position) que devuelva getItem(position).titulo.hashCode();

```
@Override
public boolean hasStableIds() {
   return true;
}
@Override
public long getItemId(int position) {
   Libro l = getItem(position);
   return l.titulo.hashCode();
}
```

4. Ejecuta la aplicación y comprueba que sigue funcionando.

Práctica 8.12: Ordenación de los elementos

Dado que ArrayAdapter guarda los elementos que se muestran, podemos utilizarlo para que nos los ordene por algún criterio, y luego notifique a la vista el cambio. Para eso, se utiliza su método sort() que recibe un java.util.Comparable que compara dos elementos de la lista (Libro en nuestros ejemplos).

1. Haz una copia de la práctica anterior.
2. Modifica el layout para añadir dos botones en la parte de abajo, con el texto “Por título” y “Por autor”, situado uno junto al otro. Define las cadenas en el fichero de recursos, y establece como eventos los métodos onPorTitulo y onPorAutor.

```
<RelativeLayout
...>
<ListView
android:id="@+id/listView"
android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
android:layout_alignParentTop="true"
android:layout_above="@+id/botones"
/>
<LinearLayout
android:id="@+id/botones"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true">
<Button
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:text="@string/porTitulo"
android:onClick="onPorTitulo"
/>

<Button
android:layout_width="0dp"
android:layout_height="wrap_content"
android:layout_weight="1"
android:text="@string/porAutor"
android:onClick="onPorAutor"
/>
</LinearLayout>
</RelativeLayout>
```

3. Desde los métodos de los eventos necesitaremos acceder al adaptador, que ahora es una variable local (y final) del método onCreate(). Convierte la variable en un atributo.
4. Implementa el código del evento onPorTitulo(). Tendrás que llamar al método sort() del adaptador. Espera recibir un Comparador que reciba dos libros y diga cuál va antes. Devuelve un valor para que se ordene por título y, en caso de empate, por autor.
5. Implementa el código del evento onPorAutor() de una manera equivalente a la anterior.

```
public void onPorTitulo(View v) {
aa.sort(new java.util.Comparator<Libro>() {
@Override
public int compare(Libro lhs, Libro rhs) {
int resultTitulo;
resultTitulo = lhs.titulo.compareToIgnoreCase((rhs.titulo));
if (resultTitulo != 0)
return resultTitulo;
else
// Por simplicidad, asumo que no hay dos libros
// anónimos con el mismo autor.
return lhs.autor.compareToIgnoreCase(rhs.autor);
}
});
} // onPorTitulo
public void onPorAutor(View v) {
aa.sort(new java.util.Comparator<Libro>() {
```

```
@Override
public int compare(Libro lhs, Libro rhs) {
if (lhs.autor == null)
if (rhs.autor == null)
return lhs.titulo.compareTo(rhs.titulo);
else
return -1;
else if (rhs.autor == null)
return 1;
int resultAutor;
resultAutor = lhs.autor.compareToIgnoreCase((rhs.autor));
if (resultAutor != 0)
return resultAutor;
else
return lhs.titulo.compareToIgnoreCase(rhs.titulo);
}
});
} // onPorAutor
```

Práctica 8.13: Vista específica para la lista vacía

Si ejecutas la aplicación y borras todos los elementos, la lista quedará completamente vacía, algo que puede resultar confuso para el usuario. ListView permite que se le establezca una vista para ser mostrada cuando la lista quede vacía. La vista debe ser hermana del ListView en el layout. ListView se encargará de hacerse visible a sí misma o a la vista alternativa en función de si tiene o no elementos.

1. Haz una copia de la práctica anterior.
2. Aunque no es necesario, para que te resulte más cómodo el siguiente paso pon android:visibility="gone" en la ListView de la actividad.
3. Añade los componentes gráficos necesarios para conseguir el resultado de la figura 1.1.

El icono de “aviso” es el recurso @android:drawable/ic_dialog_alert.

4. Quita, si quieres, el android:visibility="gone" de la lista.
5. En el onCreate(), configura la lista para establecer como vista especial para el estado “vacío” la vista que acabas de crear:

```
lv.setEmptyView(findViewById(R.id.<id>));
```

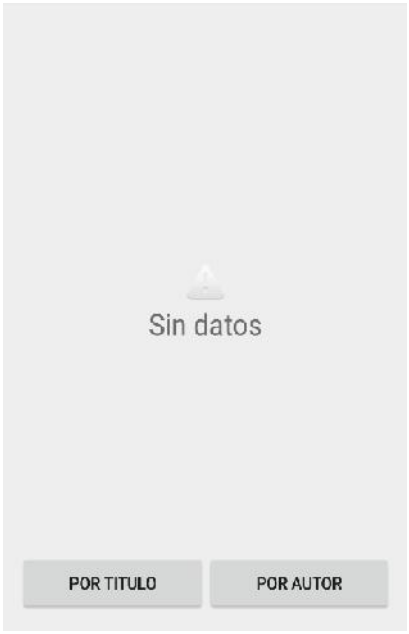

Figura 1.1: Ejemplo de vista vacía

6. Prueba la aplicación. Elimina todos los libros y observa que la vista cambia.

Como referencia, el layout podría ser algo así:

```
<RelativeLayout
android:id="@+id/emptyListView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_alignParentTop="true"
android:layout_above="@+id/botones">
<ImageView
android:id="@+id/alertIcon"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_centerInParent="true"
android:src="@android:drawable/ic_dialog_alert"/>
<TextView android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_below="@+id/alertIcon"
android:layout_centerHorizontal="true"
android:text="@string/listaVacia"
android:textSize="24sp"
/>
</RelativeLayout>
```

Práctica 8.14: ViewStub: optimizando las vistas

En la práctica anterior, el layout contiene una vista que es posible que no llegue a mostrarse nunca. Si esa vista resultara ser costosa de construir estaríamos desperdiciando recursos.

Cuando ocurre esto, un modo muy sencillo de mejorar el rendimiento de nuestras vistas es utilizar ViewStub's. Es un contenedor “dummy” que no consume apenas recursos y que en el momento de ser “inflado” (o ser hecho visible) ocasiona la carga de un layout secundario y se sustituye a sí mismo por él.

- Haz una copia de la práctica anterior.
- Crea un nuevo layout y llámalo emptylistlayout.xml. Copia en él el contenido del layout usado cuando la lista estaba vacía en la práctica anterior.
- Elimina del layout principal de la actividad el de la vista vacía y sustitúyelo por un ViewStub. Es importante que pongas su visibilidad en gone desde el principio, para que no se resuelva durante la inicialización de la actividad, antes de que la ListView entre en juego. Además, tendrás que poner el atributo android:layout con el identificador del layout secundario que se cargará.

```
<ViewStub
android:id="@+id/emptyListView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
< otros android:layout_* que pudieras tener >
android:layout="@layout/emptylistlayout"/>
```

1. Ejecuta la aplicación. No deberías observar ninguna diferencia.
2. Para comprobar que, efectivamente, estamos retrasando la carga del layout secundario, modifícalo para que sea incorrecto y la aplicación falle durante su “inflado”. Elimina, por ejemplo, el atributo layout_width.
3. Lanza de nuevo la aplicación. Comprueba a priori que todo funciona correctamente.
4. Elimina los items de la lista. Nota que en el momento en el que se debe mostrar la vista vacía la aplicación falla.

Práctica 8.15: ListActivity: ahorrándonos trabajo

1. Haz una copia de la práctica anterior.
2. Haz que la clase MainActivity herede de ListActivity.

```
public class ListViewConDosEtiquetas extends ListActivity
```

3. Modifica el layout para que el identificador de la lista sea el que espera la ListActivity, @android:id/list. Haz lo mismo con el de la vista para la lista vacía @android:id/empty.

```
<ListView android:id="@android:id/list"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
<ViewStub
   android:id="@android:id/empty"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_alignParentTop="true"
   android:layout_above="@+id/botones"
```

```
    android:layout="@layout/emptylistlayout"
    />
```

4. Elimina de onCreate() la línea en la que establecíamos la vista vacía (llamando a setEmptyView() de la ListView).

```
//lv.setEmptyView(findViewById(R.id.emptyListView));
```

5. Implementa el método protegido onListItemClick(...) con los mismos parámetros y cuerpo que el del listener que ya tienes que mostraba un toast indicando que se mantuviera pulsado para borrar. Elimina de onCreate() el establecimiento del listener.

```
        @Override
protected void onListItemClick(ListView l, View v, int position, long id) {
    Toast.makeText(getApplicationContext(),
        "Mantén pulsado para borrar",
        Toast.LENGTH_SHORT).show();
}

/* lv.setOnItemClickListener(
      new AdapterView.OnItemClickListener() {
        public void onItemClick(AdapterView<?> parent, View view,
                        int position, long id) {
          Toast.makeText(getApplicationContext(),
                "Mantén pulsado para borrar",
                Toast.LENGTH_SHORT).show();
        }
      });*/
```

6. Por desgracia, ListActivity no proporciona un método equivalente para las pulsaciones largas y tendremos que mantener el listener enganchado de manera artesanal. No obstante, en el código de onCreate(), en lugar de buscar la ListView manualmente, utiliza el nuevo método getListView(). Fíjate que ya no necesitarás la conversión de tipos.

```
//ListView lv = (ListView) findViewById(R.id.listView);
//lv.setAdapter(aa);
setListAdapter(aa);
ListView lv = getListView();
```

7. Para establecer el adaptador, utiliza el método setListAdapter() de la clase padre, en lugar de hacerlo a través de la vista que has obtenido para establecer el listener.

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

```
java.util.ArrayList<Libro> libros;
libros = new java.util.ArrayList<Libro>();
// [ .... ]
aa = new MiArrayAdapter(this, libros);
setListAdapter(aa);

ListView lv = getListView();
lv.setOnItemLongClickListener(
new AdapterView.OnItemLongClickListener() {
public boolean onItemLongClick(AdapterView<?> parent,
View view,
int position, long id) {
Libro l = aa.getItem(position);
Toast.makeText(getApplicationContext(),
"Borrado '" + l.titulo +
"' (" + position + ", " + id + ")",
Toast.LENGTH_SHORT).show();
libros.remove(position);
return true;
}
});
} // onCreate
@Override
protected void onListItemClick(ListView l, View v,
int position, long id) {
Toast.makeText(getApplicationContext(),
"Mantén pulsado para borrar",
Toast.LENGTH_SHORT).show();
}
```

Notas bibliográficas

http://developer.android.com/guide/topics/ui/notifiers/toasts.html
http://developer.android.com/guide/topics/ui/declaring-layout.html#AdapterViews
http://developer.android.com/guide/topics/ui/layout/listview.html
http://developer.android.com/reference/android/view/ViewStub.html
http://developer.android.com/reference/android/app/ListActivity.html