La Clase `List` y `ListView`

CONCEPTOS

PARTE I.

En la mayoría de las aplicaciones móviles, a los usuarios usualmente se les muestra una lista de opciones. La mayoría de las veces, es una matriz de cadenas o datos pueden ser almacenados en otro sitio.

Un adaptador es la forma en que Android traduce los datos desde alguna fuente de datos, a una vista. Normalmente se utilizan los adaptadores de `ListView` y `GridView`, y Android ofrece otros adaptadores muy útiles.

El adaptador va de la mano con una vista correspondiente, en este caso es un `ListView`. Juntos traen un código más limpio, rendimiento optimizado y reusabilidad al proyecto.

Separación limpia de lo importante
Cuando se agrega un objeto a un adaptador, la acción también agrega la vista representativa de ese objeto a las listas enlazadas al adaptador. El código que agrega datos en la lista, a través del adaptador, no tiene que saber cómo se construye la lista para cada elemento. Del mismo modo, el código, que sabe cómo construir vistas para cada elemento de la lista no tiene que saber de dónde vienen los datos.

Esto hace una separación limpia de lo importante, que normalmente se superpone en un punto. Cuando se escribe código, para rellenar elementos de interfaz de usuario con diversos datos, se puede entender cómo el proyecto se enreda cuando el código que extrae los datos también es el código que los muestra.

Optimizaciones: Reciclado de `View`
Otra ventaja de utilizar un `ListView/adaptador` es la ganancia de rendimiento. Si se construye un `ScrollView` con una lista de `Views` hijos, el rendimiento se vuelve lento rápidamente. El par `ListView/adaptador` soluciona esto con la técnica de reciclado de `View`.

El reciclado es una técnica en la que se vuelven a utilizar objetos `View`, una vez que sus ubicaciones actuales ya no son visibles. Esto tiene una gran ventaja; como la lista de elementos crece, el número total de `Views` sigue siendo el mismo, lo que corresponde con el número distinto de `Views` dibujados en una sola pantalla completa. Incluso mejor aún, el proceso se maneja detrás de las escenas, por lo que el adaptador sólo debe ser consciente de que los `Views` se vuelven a utilizar, no necesariamente en la forma en que se vuelven a utilizar.

La reutilización de código
Por último, el código que se escriba para un adaptador, la separación obligada de los datos, la vista, y las cuestiones de adaptación, se traducirá en código mucho más reutilizable. Donde se haya escrito anteriormente todo el código directamente en alguna `Activity` para mostrar una lista de objetos complejos, ahora se tiene una clase separada que se puede reutilizar para otras listas. De hecho, incluso se puede modificar la plantilla que utiliza el adaptador sin cambiar nada sobre el propio adaptador.

DESARROLLO

EJEMPLO 1.
Paso 1. Crear un nuevo proyecto Listas1. En el archivo Java predeterminado MainActivity.java, capturar el siguiente código:

```
import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ListAdapter;
```

```
public class MainActivity extends ListActivity {
    @Override
    public void onCreate(Bundle b) {
        super.onCreate(b);
        ListAdapter la = crearAdapter();
        setListAdapter(la);
    }
    protected ListAdapter crearAdapter(){
      return null;
    }
}
```

Paso 2. Los adaptadores básicamente agregan una colección de objetos a una actividad, proporcionando una asignación o las instrucciones sobre la forma de representar cada objeto en la lista de la actividad. En este caso, solamente es una simple lista de cadenas, por lo que este adaptador es sencillo y se implanta con `ArrayAdapter`.

Modificar el método de `crearAdapter` ( ) con el siguiente código:

```
protected ListAdapter crearAdapter(){
    String[] s = new String[] {
        "Elemento1",
        "Elemento2",
        "Elemento3"
    };
    ListAdapter la2 = new ArrayAdapter<String> (this, android.R.layout.simple_list_item_1, s);
      return la2;
}
```

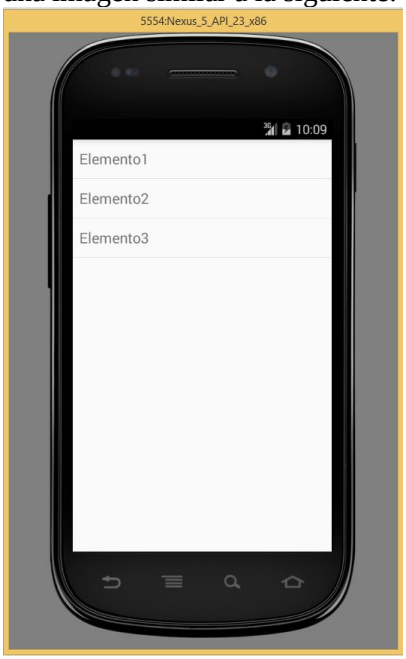
Paso 3. Lo anterior encapsula todo lo necesario para tener un adaptador funcional. Normalmente, aquí se puede hacer una llamada a los datos, como puede ser una consulta SQLite o una solicitud RESTful GET, pero por ahora sólo se utilizan cadenas simples. Se crea el `ArrayAdapter` y se especifican tres elementos: `Context`, `Layout` y `Data`.

- El contexto es la procedencia de la lista. En esta aplicación es muy sencillo, pero a veces es complejo cuando se pasan intentos de ida y vuelta a través de actividades, por lo que el contexto se utiliza para mantener una referencia a la actividad propietaria en todo momento.
- La plantilla muestra los datos. En este caso, se utiliza una plantilla preconstruida `simple_list_item_1`.
- Los datos por mostrar consisten en una simple lista de cadenas.

El resultado final debe ser similar al siguiente código:

```
import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
public class MainActivity extends ListActivity {
    public void onCreate(Bundle b) {
        super.onCreate(b);
        ListAdapter la = crearAdapter();
        setListAdapter(la);
    }
    protected ListAdapter crearAdapter(){
        String[] s = new String[] {
                "Elemento1",
                "Elemento2",
                "Elemento3"
        };
        ListAdapter la2 = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, s);
        return la2;
    }
}
```

Paso 4. Al ejecutar la aplicación se muestra una imagen similar a la siguiente:



PARTE II.

Para crear una nueva clase que herede de `ArrayAdapter<NuevasEntradas>`, se necesita proveer un constructor, en este caso con los parámetros (`Context`, `int`). Además, se sobrescribe el método `getView()` para configurar la construcción de cada `View`.

Primero, se revisan los detalles para la configuración de la clase NuevaEntradaAdapter, que son los siguientes:

El método `getView()`

Sobrescribiendo el método `getView()` de la clase `ArrayAdapter`:

```
public View getView(final int i, final View v, final ViewGroup vg) {
    final View v2 = getWorkingView(v);
    final ViewHolder vh = getViewHolder(v2);
    final NuevaEntrada ne = getItem(i);
    vh.tituloView.setText(ne.getTitulo());
    final String s = String.format("Por %s on %s", ne.getAutor(),
            DateFormat.getDateInstance(DateFormat.SHORT).format(ne.getFecha())
    );
    vh.subTituloView.setText(s);
    vh.imagenView.setImageResource(ne.getIcono());
    return v2;
}
```

La funcionalidad principal del adaptador comienza con el `getView()`, el cual es el que se invoca cada vez que el `ListView` necesita mostrar algo. Los argumentos de `getView()` se refieren a `convertView`, `workingView` y `viewHolder`.

El argumento de position `i` indica cuál elemento del arreglo de objetos que se agregan, es el que se manejará, lo cual es muy útil conocer.

El argumento `convertView` es el `View`, posiblemente reciclado. Si es `non-null`, entonces es un elemento que ya no será visible y se debe utilizar para conservar los objetos `View`. Si es `null`, entonces se crea un nuevo `View`. El elemento resultante (ya sea reutilizado o de nueva creación) es lo que se llama **`working view`**, o `View` de trabajo.

El argumento padre es una referencia al elemento que contiene el `ListView`. En este caso, no es necesario.

Las siguientes líneas determinan el `working view` y extraen un `ViewHolder`, más el objeto correspondiente `NuevaEntrada`:

```
final View v2 = getWorkingView(v);
final ViewHolder vh = getViewHolder(v2);
final NuevaEntrada ne = getItem(i);
```

El método `getItem`() solamente regresa el `NuevaEntrada`, que actualmente se intenta enviar para que el `ListView` lo muestre. Las líneas restantes realizan la decoración real de la vista:

```
vh.tituloView.setText(ne.getTitulo());
final String s = String.format("Por %s on %s", ne.getAutor(),
        DateFormat.getDateInstance(DateFormat.SHORT).format(ne.getFecha())
);
vh.subTituloView.setText(s);
vh.imagenView.setImageResource(ne.getIcono());
return v2;
```

El título y la imagen se asignan al título de `NuevasEntradas`. El subtítulo requiere formato de fechas.
Finalmente, se vuelve la vista al final de `getView`(), lo que hará que se pueda reciclar más adelante, cuando sea posible.

El método `getWorkingView()`

Aquí, se define el método particular `getWorkingView`():

```
private View getWorkingView(final View v3) {
    View workingView = null;
    if(null == v3) {
        final Context c2 = getContext();
        final LayoutInflater inflater = (LayoutInflater)c2.getSystemService
                (Context.LAYOUT_INFLATER_SERVICE);
        workingView = inflater.inflate(entradaLayoutRecurso, null);
    } else {
        workingView = v3;
    }
    return workingView;
}
```

El `workingView` es el resultado de tener un `View`, fue reciclado o se forzó la creación de uno nuevo.

El método `getViewHolder()`

Aquí, se define el método particular `getViewHolder`():

```
private ViewHolder getViewHolder(final View workingView) {
    final Object tag = workingView.getTag();
    ViewHolder vh = null;
    if(null == tag || !(tag instanceof ViewHolder)) {
        vh = new ViewHolder();
        vh.tituloView = (TextView) workingView.findViewById(R.id.xtvtitulo);
        vh.subTituloView = (TextView) workingView.findViewById(R.id.xtvsubtitulo);
        vh.imagenView = (ImageView) workingView.findViewById(R.id.xivicono);
        workingView.setTag(vh);
    } else {
        vh = (ViewHolder) tag;
    }
    return vh;
```

```
}
private static class ViewHolder {
    public TextView tituloView;
    public TextView subTituloView;
    public ImageView imagenView;
}
```

El `ViewHolder` es una clase personalizada definida por conveniencia, eficiencia y seguridad de tipos. El principal beneficio de `ViewHolder` se basa en el hecho de que, cuando se almacena como etiqueta de una vista, se recicla junto con la propia vista. Un `View` puede almacenar cualquier objeto, conocido como su etiqueta, para una referenciación adecuada. En este caso, se almacena un objeto que ya tiene referencias directas a los subvistas dentro de la plantilla. Esto evita realizar las operaciones de búsqueda de cada subvista (por medio de `findViewById`()) cada vez que se redibuja. Como se incrementa el número de llamadas a `getView`(), es igual el número de veces que se invoca a `findViewById`() en un momento determinado. También, se le puede convertir a nuevos tipos específicos de vistas (`TextView`, `ImageView`) y ya no tener que convertirlos de nuevo.

EJEMPLO 2

Paso 1. Crear un nuevo proyecto Listas2. En la carpeta java/com.example.mipaquete, abrir el archivo Java predeterminado MainActivity.java y capturar el siguiente código:

```
import android.app.Activity;
import android.os.Bundle;
public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle b) {
        super.onCreate(b);
        setContentView(R.layout.main);
    }
}
```

Paso 2. En la carpeta res/layout, abrir el archivo XML predeterminado activity_main.xml y capturar el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>
```

Paso 3. Un `ListView` es un `View` que agrupa a sus hijos en una lista. Para ello, modificar el archivo anterior activity_main.xml con el siguiente código. El objetivo es tener un `Listview` dentro de una plantilla estándar y a partir de allí construir plantillas más complejas.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/lista" />
```

```
    <ListView
        android:id="@+id/xlv1"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Paso 4. Lo siguiente es vincular objetos comunes utilizando un adaptador común. Con frecuencia se tiene un modelo de objetos que describe los tipos de datos que se trata de mostrar; en este caso, se tiene una clase inmutable NuevasEntradas.java que encapsula la información de nuevas entradas; se incluye título, autor, fecha y una imagen.

En la carpeta java/com.example.mipaquete, crear el archivo Java NuevaEntrada.java y capturar el siguiente código:

```
import java.util.Date;
public final class NuevaEntrada {
    private final String    titulo;
    private final String    autor;
    private final Date      fecha;
    private final int       icono;
    public NuevaEntrada(final String t, final String a, final Date f, final int i) {
        this.titulo = t;
        this.autor  = a;
        this.fecha  = f;
        this.icono  = i;
    }
    public String getTitulo() {
        return titulo;
    }
    public String getAutor() {
        return autor;
    }
    public Date getFecha() {
        return fecha;
    }
    public int getIcono() {
        return icono;
    }
}
```

Se ha utilizado un `ArrayAdapter` que tomó cadenas simples y las ligó a una plantilla `TextView` simple por cada cadena en la lista. Puesto que se utilizan objetos más complejos, tales como NuevaEntrada, no se puede depender necesariamente de `toString`() para satisfacer un `View` complejo. Android proporciona una clase `SimpleAdapter` que puede ser útil, pero en este caso solamente se hereda y sobrescribe el `ArrayAdapter`.

Paso 5. Enseguida, se mencionan los pasos finales. En la carpeta res/drawable, colocar las dos siguientes imágenes incluidas icon_1.png y icon_2.png. Además, en la carpeta res/layout, crear el archivo nueva_entrada_lista.xml y modificarlo con el siguiente código:

```
<?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">
    <ImageView
        android:id="@+id/xivicono"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:padding="3dp" />
    <TextView
        android:id="@+id/xtvtitulo"
        android:layout_width="fill_parent"
```

```
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/xivicono"
        android:layout_alignTop="@id/xivicono"
        android:layout_margin="5dp"
        android:textSize="14sp"
        android:textStyle="bold" />
    <TextView
        android:id="@+id/xtvsubtitulo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@id/xtvtitulo"
        android:layout_below="@id/xtvtitulo"
        android:textSize="12sp" />
</RelativeLayout>
```

Como se puede ver, se utilizó un `RelativeLayout` para colocar el título, el icono y los `Views` de subtítulos. Se especifica la altura y ancho de los elementos de interfaz de usuario; el valor se puede expresar con las unidades de dimensión soportados por Android (`px`, `dp`, `sp`, `pt`, `in`, `mm`).

Paso 6. En la carpteta java/com.example.mipaquete, crear y modificar el archivo NuevaEntradaAdapter.java con el siguiente código:

```
import java.text.DateFormat;
import android.content.Context;
import android.view.*;
import android.widget.*;
public final class NuevaEntradaAdapter extends ArrayAdapter<NuevaEntrada> {
    private final int entradaLayoutRecurso;
    public NuevaEntradaAdapter(final Context c, final int entLayRec) {
        super(c, 0);
        this.entradaLayoutRecurso = entLayRec;
    }
    @Override
    public View getView(final int i, final View v, final ViewGroup vg) {
        final View v2 = getWorkingView(v);
        final ViewHolder vh = getViewHolder(v2);
        final NuevaEntrada ne = getItem(i);
        vh.tituloView.setText(ne.getTitulo());
        final String s = String.format("Por %s on %s", ne.getAutor(),
                DateFormat.getDateInstance(DateFormat.SHORT).format(ne.getFecha())
        );
        vh.subTituloView.setText(s);
        vh.imagenView.setImageResource(ne.getIcono());
        return v2;
    }
    private View getWorkingView(final View v3) {
        View workingView = null;
        if(null == v3) {
            final Context c2 = getContext();
            final LayoutInflater inflater = (LayoutInflater)c2.getSystemService
                    (Context.LAYOUT_INFLATER_SERVICE);
            workingView = inflater.inflate(entradaLayoutRecurso, null);
        } else {
            workingView = v3;
        }
        return workingView;
    }
    private ViewHolder getViewHolder(final View workingView) {
        final Object tag = workingView.getTag();
        ViewHolder vh = null;
        if(null == tag || !(tag instanceof ViewHolder)) {
            vh = new ViewHolder();
            vh.tituloView = (TextView) workingView.findViewById(R.id.xtvtitulo);
```

```
            vh.subTituloView = (TextView) workingView.findViewById(R.id.xtvsubtitulo);
            vh.imagenView = (ImageView) workingView.findViewById(R.id.xivicono);
            workingView.setTag(vh);
        } else {
            vh = (ViewHolder) tag;
        }
        return vh;
    }
    private static class ViewHolder {
        public TextView tituloView;
        public TextView subTituloView;
        public ImageView imagenView;
    }
}
```

El constructor que se especifica enseguida, recibe un `Context` y un `int`:

```
public NuevaEntradaAdapter(final Context c, final int entLayRec) {
    super(c, 0);
    this.entradaLayoutRecurso = entLayRec;
}
```

Se debe especificar el contexto de la clase padre `ArrayAdapter`, y el `entradaLayoutRecurso` indica cuál diseño se debe utilizar para cada noticia. Ya se ha creado uno en res/layout/nueva_entrada_lista.xml, por lo que el recurso que se pasará en este constructor será `R.layout.nueva_entrada_lista`.

Reuniendo todo lo anterior: La lista con los objetos y el adaptador.
Ahora que se ha terminado con las plantillas, objetos y el adaptador, se utiliza la nueva herramienta creada.

Para hacer uso del nuevo adaptador, se revisa la `Activity` original. Se necesita realizar lo siguiente:

- Referenciar el `ListView` en el `MainActivity`.
- Ligar el `ListView` al `NuevaEntradaAdapter`.
- Llenar el `NuevaEntradaAdapter` con los objetos `NuevaEntrada`.

Paso 7. El código modificado debe ser similar al siguiente:

```
import android.app.Activity;
import android.os.Bundle;
import java.util.*;
import android.widget.ListView;
public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle b) {
        super.onCreate(b);
        setContentView(R.layout.activity_main);
        final ListView lv = (ListView) findViewById(R.id.xlv1);
        final NuevaEntradaAdapter nea = new NuevaEntradaAdapter(this,
R.layout.nueva_entrada_lista);
        lv.setAdapter(nea);
        for(final NuevaEntrada i : getEntradas()) {
            nea.add(i);
        }
    }
    private List<NuevaEntrada> getEntradas() {
        final List<NuevaEntrada> datos = new ArrayList<NuevaEntrada>();
        for(int i = 1; i < 31; i++) {
            datos.add(
                    new NuevaEntrada(
                            "Datos de Entrada " + i, "Alejandro ESCOM " + i,
                            new GregorianCalendar(2016, 12, i).getTime(),
                            i % 2 == 0 ? R.drawable.icon_1 : R.drawable.icon_2
```

```
                    )
                );
            }
            return datos;
        }
}
```

Paso 8. Por último, al ejecutar la aplicación se debe mostrar una imagen similar a la siguiente con el código terminado. El usuario puede desplazar la lista para mostrar más elementos: