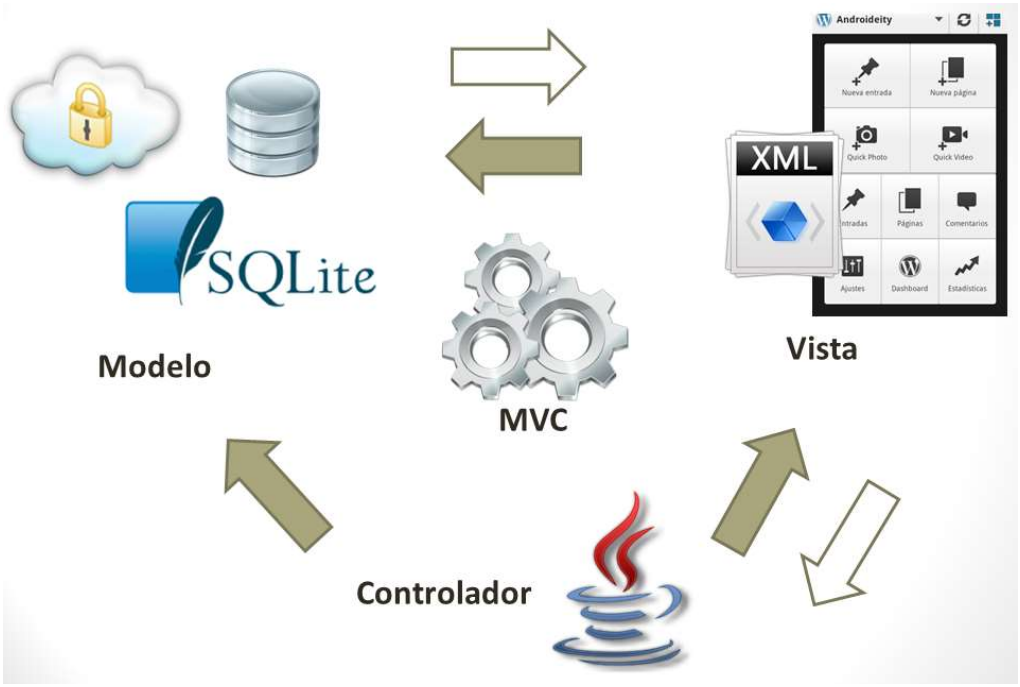
Interfaces de usuario. Layouts

1. Modelo vista controlador

En Android utilizamos el patrón de arquitectura llamado **Modelo Vista Controlador (MVC)**, que separa los datos de una aplicación, la interfaz de usuario y la lógica de negocios en tres componentes distintos que se relacionarán para al final tener como resultado una aplicación.

Los componentes de este modelo:



- **Modelo.** Nos referimos con modelo a las representaciones que construiremos basadas en la información con la que operará nuestra aplicación. También hay que decidir cómo se guardará la información: ¿Base de datos? ¿Web services? El modelo elegido depende obviamente de las necesidades de información de la aplicación.

- **Vista.** La vista no es más que la interfaz con la que va a interactuar el usuario. En Android, las interfaces las construimos en XML. Construimos nuestro esqueleto en XML que equivale al HTML de un sitio. Posteriormente, con ayuda de estilos, que también los escribimos en XML, podemos empezar a darle formato de colores, posiciones, formato, etc. a nuestro esqueleto.

- **Controlador.** Finalmente nos topamos con el controlador que son todas esas clases que nos ayudarán a darle vida a esas interfaces que ya construimos y nos permitirán desplegar y consumir información de/para el usuario. Estos controladores se programan en lenguaje Java y son el *core* de la aplicación.

2. Interfaz de usuario

2.1. Creación de una interfaz de usuario con código

Vamos a ver un primer ejemplo de cómo crear una interfaz de usuario utilizando exclusivamente código. Esta no es la forma recomendada de trabajar con Android pero nos será de ayuda para entender algunos conceptos.

Por ejemplo, si necesitamos introducir un campo de texto en una aplicación el código sería algo parecido a:

```
...
public class MainActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      ...
      TextView texto = new TextView(this);
      texto.setText("Hola");
      setContentView(texto);
      ...
   }
   ...
}
```

Sin embargo, esta forma de crear las interfaces de usuario no es la correcta ya que el mantenimiento de la misma y los posibles cambios serían muy costosos.

Un principio importante en el diseño de software es que conviene separar todo lo posible el diseño, los datos y la lógica de la aplicación, y en este caso la capa de presentación se genera directamente mediante programación.

2.2. Creación de una interfaz de usuario usando XML

Android proporciona una alternativa para el diseño de interfaces de usuario. Se trata de los ficheros de diseño basados en XML. Vamos a ver uno de estos ficheros:

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

   <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/hello_world" />
</RelativeLayout>
```

Resulta sencillo interpretar su significado. Se introduce un elemento de tipo *RelativeLayout*. Su función es contener otros elementos de tipo *View* (o controles). Este *RelativeLayout* tiene varios atributos.

El primero, *xmlns:android*, es una declaración de un espacio de nombres de XML que utilizaremos en Android (este parámetro solo es necesario especificarlo en el primer elemento). Los atributos *android:layout_width* y *android:layout_height* permiten definir el ancho y alto de la vista. En el ejemplo se ocupará todo el espacio disponible

en la pantalla puesto que hemos utilizado los valores *match_parent*. También tenemos el valor *wrap_content* para que el tamaño dependa del valor que haya dentro del elemento.

Dentro del *RelativeLayout* solo tenemos un elemento de tipo *TextView*. Este dispone de varios atributos. Los dos primeros definen el ancho y alto, que dependen del texto que contenga el elemento *TextView*. El atributo *android:text="@string/hello_word"* indica el texto a mostrar. No se ha indicado un texto en concreto sino una referencia, "@string/*hello_word*". Esta referencia ha de estar definida en el fichero *res/values/strings.xml*:

```
<resources>
    <string name="app_name">U001Hola</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

Esta es la práctica recomendada en Android para la inserción de textos, dado que facilita su localización a la hora de realizar la traducción a otros idiomas.

En resumen, los ficheros *layout* se utilizan para introducir el diseño de los interfaces y el fichero *strings* para introducir los textos utilizados en los distintos idiomas.

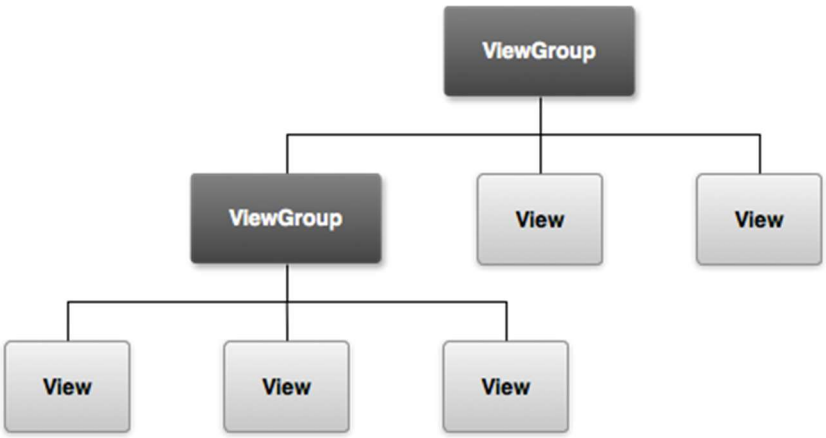
Android permite usar uno de estos métodos o ambos a la hora de crear una interfaz de usuario. Por ejemplo, se puede declarar un TextView en XML y luego modificar sus propiedades en tiempo de ejecución.

2.2.3. Activity, ViewGroup y View

Las interfaces de usuario se construyen utilizando las siguientes clases:

- View. Representa un componente visual. Todos los componentes derivan de *View*. También se les llama *widgets* o controles. Por ejemplo, un botón.
- ViewGroup. Extensiones de *Wiew* que agrupan varias *Wiews*. Por ejemplo, *Layout*.
- Activity. Pantalla de la aplicación. Donde se van a añadir los *Views* y *ViewGroups*.

La interfaz se construye usando una jerarquía de objetos *View* y *ViewGroup*:



El diseño de la interfaz de usuario en Android sigue una filosofía muy diferente a la de otras plataformas. En Android la interfaz de usuario no se diseña exclusivamente con código Java, sino utilizando XML o incluso utilizando una interfaz gráfica.

2.2.4. Clase R.Java

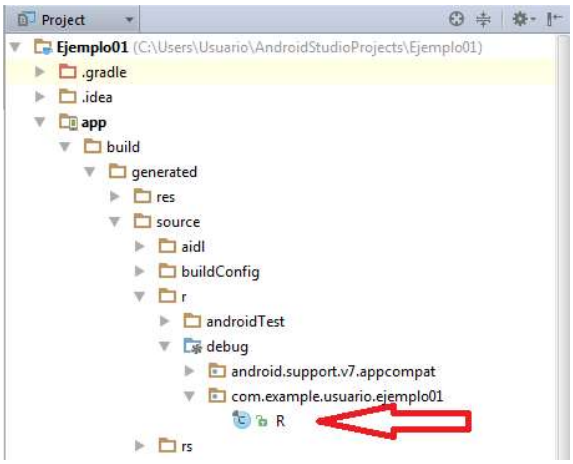
Si creamos una primera aplicación con el asistente, dentro del fichero *MainActivity.java* aparece el siguiente codigo:

```
public class MainActivity extends AppCompatActivity {

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

Aquí, *R.layout.activity_main* corresponde a un objeto *View* que será creado en tiempo de ejecución a partir del recurso *activity_main.xml*. O sea, en *activity_main.xml* definimos mediante código XML como queremos que sea la interfaz de la pantalla principal, y R.*layout.activity_main* es la referencia a dicha interfaz. Trabajar de esta forma, en comparación con el diseño basado en código, no quita velocidad y requiere menos espacio.

El archivo R.java es un archivo que se autogenera dentro de la carpeta *build*, y que contiene una referencia a todos los elementos de la carpeta *res*.



Este fichero se genera automáticamente. Nunca debe editarse de forma manual. Referencia los recursos del proyecto: imágines, vistas, ..

Para referenciar un recurso desde código:

- R.color.blue
- R.id.boton_aceptar
- R.layout.layout_ppal
- R.menu.menu_inicial

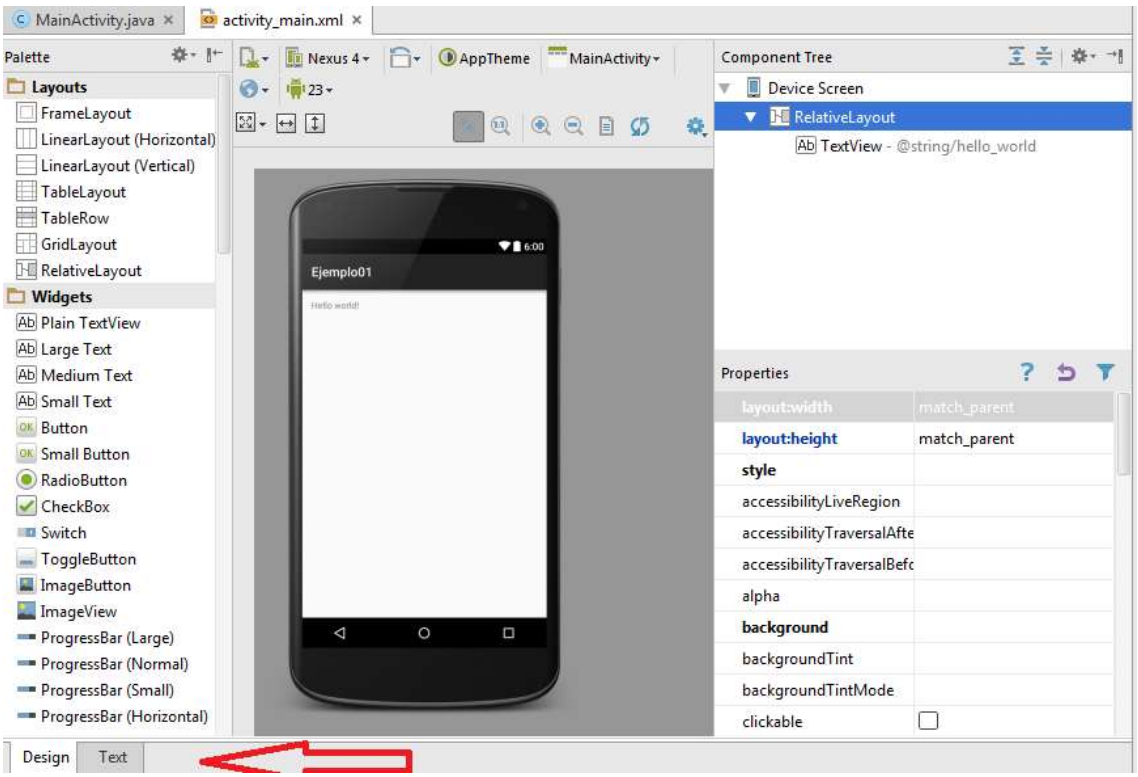
Desde otros XML:

- android:text="@string/str_cancelar"
- android:layout_toRightOf="@id/boton_aceptar"

3. Edición visual de las vistas

Los *layouts* o ficheros de diseño en XML los podemos editar de forma visual. Si abrimos el fichero *res/layout/activity_main.xml* podemos apreciar que en la parte inferior de la ventana aparecen dos pestañas: *Design* y *Text*.

Android Studio permite dos tipos de diseño: editar directamente el código XML (pestaña *Text*) o realizar el diseño de forma visual (pestaña *Design*).



En el marco derecho se visualiza una lista con todos los elementos de la vista. En este momento solo hay dos: un *RelativeLayout* que contiene un *TextView*. En el marco central aparece una representación de cómo se verá el resultado. En la parte superior aparecen varios controles para representar esta vista en diferentes configuraciones.

Cuando diseñamos una vista en Android, hay que tener en cuenta que desconocemos el dispositivo final donde será visualizada y por lo tanto debemos intentar que se visualice bien en cualquier configuración.

Para editar un elemento lo seleccionamos en el marco de la derecha o pinchamos sobre él en el marco central pudiendo de esta forma modificar algunas de sus propiedades. El marco de la izquierda permite insertar de forma rápida nuevos elementos a la vista simplemente arrastrándolos.

4. Layouts

4.1. Tipos

Si queremos combinar varios elementos de tipo vista tendremos que utilizar un objeto de tipo *Layout*.

Un *Layout* es un contenedor de una o más vistas y controla su comportamiento y posición. Hay que destacar que un *Layout* puede contener a otro *Layout* y que es un descendiente de la clase *View*.

Algunos de los distintos *Layouts* con los que podemos trabajar son:

- **LinearLayout**: Es el *Layout* más utilizado en la práctica. Distribuye los elementos uno detrás de otro, bien de forma horizontal o vertical.

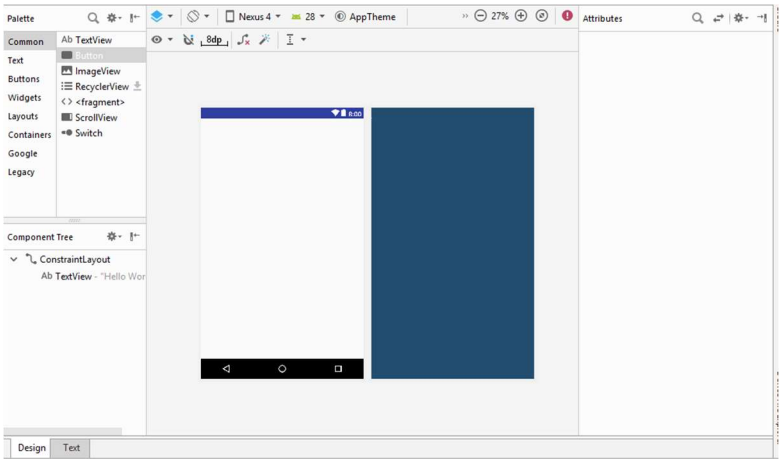
- **TableLayout**: Distribuye los elementos de forma tabular. Se utiliza la etiqueta *TableRow* cada vez que queremos insertar una nueva línea.

- **RelativeLayout**: Permite comenzar a situar los elementos en cualquiera de los cuatro lados del contenedor o ir añadiendo nuevos elementos pegados a estos.

- **AbsoluteLayout**: Permite indicar las coordenadas (x,y) donde queremos que se visualice cada elemento.

- **FrameLayout**: Posiciona todos los elementos usando todo el contenedor, sin distribuirlos espacialmente. Este *Layout* suele utilizarse cuando queremos que varios elementos ocupen un mismo lugar, pero solo uno será visible. Para modificar la visibilidad de un elemento utilizamos la propiedad *visibility*.

- **ConstraintLayout:** Es una versión mejorada de RelativeLayout que permite realizar interfaces complejas simplemente arrastrando y soltando sin necesidad de modificar el fichero XML.



También podemos utilizar otras clases de *Layouts*:

- **ScrollView**: Visualiza una columna de elementos, cuando estos no caben en pantalla se permite un deslizamiento vertical mediante un *scroll*.

- **ListView**: Visualiza una lista deslizable verticalmente de varios elementos.

- **GridView**: Visualiza una cuadrícula deslizable de varias filas y varias columnas.

- **TabHost**: Proporciona una lista de ventanas seleccionables por medio de etiquetas que pueden ser pulsadas por el usuario para seleccionar la ventana que desea visualizar.

- **ViewFlipper**: Permite visualizar una lista de elementos de forma que se visualice uno cada vez. Puede ser utilizado para intercambiar los elementos cada cierto intervalo de tiempo.

4.2. LinearLayout

Distribuye los elementos en una columna o en una fila. Por defecto, la orientación es horizontal.

Algunos atributos XML que podemos usar son:

android:id	Proporciona un identificador
android:gravity	Determina como se alinean los elementos
android:layout_height	Establece la altura
android:layout_width	Establece la anchura
android:padding	Establece el padding o relleno para los cuatro márgenes
android:paddingBottom	Establece el padding o relleno para el margen inferior
android:paddingLeft	Establece el padding o relleno para el margen izquierdo
android:paddingRight	Establece el padding o relleno para el margen derecho
android:paddingTop	Establece el padding o relleno para el margen superior
android:orientation	Define la orientación: horizontal o vertical
android:weight	Define la proporción del espacio disponible que nuestro View va a ocupar

Especial mención hay que hacer al atributo *id*. Por ejemplo:

```
android:id="@+id/tvMiTexto"
```

@+id indica que se agregue un *id* de un recurso dentro de *R.java* cuando se compila la aplicación. Si no lo usamos no podremos hacer referencia al recurso.

El atributo *gravity* define cómo se va a alinear un objeto tanto vertical como horizontalmente. Existen dos propiedades: *gravity* y *layout gravity***.** La primera, *gravity*, determina cómo se alinea el contenido del *View*, mientras que la segunda, *layout_gravity*, determina cómo se alinea el *View* dentro de su contenedor padre.

A todos los elementos que coloquemos en un *LinarLayout* hay que asignarles una anchura y una altura para lo cual usamos los atributos *android:layout_width* y *android:layout_height*. Para ello tenemos varias posibilidades:

- Asignar una altura y anchura fijas. Las unidades de medidas válidas son:

    - **dp** (Density independent Pixels): Unidad abstracta basada en la densidad de la pantalla. Esta unidad es equivalente a un píxel en una pantalla con una densidad de 160 *dpi (dots per inch)*. Por lo tanto, en pantallas de mayor densidad, se aumenta el número de píxeles utilizados para dibujar 1*dp* según los *dpi* de la pantalla y en pantallas de menor densidad el número de píxeles utilizados para 1*dp* se reducirán.
    - **sp** (Scale independent Pixels): Parecida a *dp*, pero se escala en función del tamaño de la fuente.
    - **pt** (Points): 1/72 de pulgada.
    - **px** (Pixels): Se corresponde con los píxel de la pantalla.
    - **mm** (Millimeters)
    - **in** (Inches)

- Utilizar el valor *wrap_content* (el tamaño depende del valor que hay dentro del elemento) o *match_parent* (ocupa todo el espacio disponible).

4.3. Relative Layout

Los *widgets* o elementos se posicionan en relación con los otros elementos.

Entre los atributos que nos sirven para posicionar los elementos con respecto a otros están:

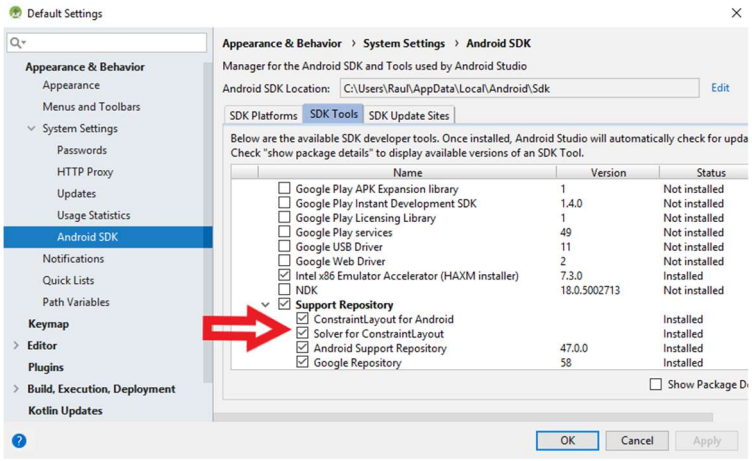
android:layout_above	El elemento se posiciona encima de otro indicado
android:layout_below	El elemento se posiciona debajo de otro indicado
android:layout_alignParentBottom	Añade el elemento en la parte inferior del *Layout*
android:layout_alignParentLeft	Añade el elemento en la parte izquierda del *Layout*
android:layout_alignParentRight	Añade el elemento en la parte derecha del *Layout*
android:layout_alignParentTop	Añade el elemento en la parte superior del *Layout*
android:layout_centerHorizontal	Centra el elemento horizontalmente
android:layout_centerVertical	Centra el elemento verticalmente
android:layout_toLeftOf	El elemento se posiciona a la izquierda de otro indicado
android:layout_toRightOf	El elemento se posiciona a la derecha de otro indicado

4.4. Constraint Layout

Básicamente supone una nueva forma de diseñar layouts. Los layouts se pueden construir arrastrando y soltando sin necesidad de editar el fichero XML y, lo que es más importante, la interfaz de usuario creada es válida para todos los dispositivos existentes en el mercado.

En este layout la posición de una vista es relativa a la de las otras o a la del propio layout, de forma que se ajustan independientemente del tamaño de la pantalla del dispositivo.

Para poder usar este tipo de layout, en el SDK manager, las casillas “ConstraintLayout for Android” y “Solver por ConstraintLayOut” deben estar seleccionadas, para que se puedan incluir las librerías en nuestros proyectos:



Esto quedará recogido en el fichero *build.grade* de nuestra aplicación:

```
implementation 'com.android.support.constraint:constraint-layout:1.1.3'
```

Una descripción muy detallada de cómo funciona se puede encontrar en https://developer.android.com/training/constraint-layout/?hl=es-419.

5. Adaptadores en Android (adapters)

Los adaptadores en Android sirven para proporcionar una interfaz común al modelo de datos que existe por detrás de todos los controles de selección, como pueden ser las listas desplegables. 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.

- **ActivityAdapter** y **ActivityIconAdapter** nos proporciona los nombres o iconos de las actividades que pueden ser invocadas desde el Intent que estemos utilizando.

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:

```
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);
```

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:

- El contexto, que normalmente será simplemente una referencia a la actividad donde se crea el adaptador.
- El ID del layout que define el aspecto visual de la lista (en este caso hacemos uso de un layout predefinido en Android: *android.R.layout.simple_list_item_1*)
- 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.setAdapter (adaptador)
```

6. ListView

En Android, utilizamos el control llamado *ListView* para desplegar una lista de opciones.

En el momento en el que agregamos este control de selección en nuestro *layout*, necesitamos invocar al método *setAdapter()* para suministrar los datos y las vistas de cada ítem que contendrá la lista. Así como un *listener* a través del método *setOnItemSelectedListener()* para saber cuándo es que se ha seleccionado una opción. Con estos elementos, tendremos una lista de opciones funcional que podemos utilizar en nuestras aplicaciones.

En el caso de que necesitemos que nuestra actividad principal despliegue solamente una lista de opciones, sería una buena idea indicar que esta sea una subclase de *ListActivity*, en lugar de la clase *Activity* como lo hacemos regularmente.

En el siguiente ejemplo práctico tenemos una lista de opciones y podemos visualizar la opción seleccionada por el usuario:

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

      <TextView
            android:id="@+id/seleccionado"
            android:text="@string/opcion"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textColor="#A4C639"
            android:textSize="20dip" />

      <ListView
            android:id="@+id/list"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:drawSelectorOnTop="false"/>

      ……
</LinearLayout>
```

El código Java sería:

```
...
public class ListDemoActivity extends Activity {
   ListView lv;
   TextView seleccionado;
   String[] datos = {"dato1", "dato2", "dato3", "dato4", "dato5"}

   @Override
   public void onCreate (Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      lv = (ListView) findViewById (R.id.list);
      seleccionado = (TextView) findViewById (R.id.seleccionado);
      lv.setAdapter (new ArrayAdapter <String> (this, android.R.layout.simple_list_item_1, datos));

      lv.setOnItemClickListener (new OnItemClickListener() {
            public void onItemClick (AdapterView <?> parent, View view, int position, long id) {
                  seleccionado.setText ("Has seleccionado: " + datos[posicion]);
            }
      });
   }
}
```

Lo primero que hacemos es crear tres variables, una de tipo *ListView* que nos servirá para manipular la lista de opciones, la segunda variable será de tipo *TextView* para asignarle a este control las opciones que vayamos seleccionando cada vez y la tercera que corresponde a un *array* de *Strings* con las opciones que queremos que tenga la lista. Después, dentro del método *onCreate()* recuperamos el *ListView* y el *TextView* definidos en el XML en las variables declaradas líneas arriba.

A continuación llenamos la lista, para lo cual llamamos al método *setAdapter()* que recibe como parámetros el contexto en el cuál se está utilizando la lista (*this*), el nombre

del recurso que define el aspecto visual de la lista (en este caso hacemos uso de un *layout* predefinido por Android: *android.R.layout.simple_list_item_1*) y por último, el arreglo de datos que en este caso definimos en la variable *datos*.

Posteriormente, lo único que hacemos es asignarle a la variable *lv* un *listener* que nos permitirá definir una acción que responda al evento de seleccionar un ítem de la lista. En este caso, lo que hacemos es indicar que se nos muestre el texto de la opción en el *TextView*.

7. GridView

Nos permite crear una cuadrícula bidimensional de elementos entre los que podemos elegir.

Cuando trabajamos con *GridView* podemos definir el número y el tamaño de las columnas; el número de filas por el contrario se define dinámicamente en función del número de ítems que el adaptador que estemos utilizando indique como disponibles para la vista a desplegar en la pantalla.

Existen una serie de propiedades que cuando se combinan, nos permiten determinar el número de columnas y sus respectivos tamaños:

- **android:numColumns** indica el número de columnas que tendrá la cuadrícula. O en el caso de que indiquemos el valor *auto_fit* en este atributo, Android calculará el número de columnas basado en el espacio disponible y en las propiedades que veremos a continuación.

- **android:verticalSpacing** y **android:horizontalSpacing** indican cuántos pixeles de espacio en blanco deben existir entre cada ítem de la cuadrícula.

- **android:columnWidth** indica cuántos pixeles de ancho deberá tener cada columna.

- **android:stretchMode** define, para las cuadrículas cuyo valor para el atributo *android:numColumns* sea *auto_fit,* cómo debe aprovecharse cada espacio disponible que no esté siendo ocupado por el ancho de las columnas o por los espacios en blanco que se definan. De esta forma, utilizaremos el valor *android:stretchMode=columnWidth* para indicar que las columnas deberán expandirse y aprovechar el espacio en blanco disponible y el valor *android:stretchMode=spacingWidth* para hacer que el espacio en blanco entre las columnas se expanda.

Pongamos un ejemplo: Supongamos que tenemos una pantalla de 320px de ancho y para nuestro *GridView* hemos definido los atributos android:columnWidth=100 y *android:horizontalSpacing=5*. Si se definen tres columnas se tendrá un espacio ocupado de 310px (que corresponden a 3 columnas de 100px y 2 espacios en blanco de 5px cada uno). Si definimos el atributo *android:stretchMode=columnWidth,* cada una de las tres columnas incrementará su ancho en unos 3-4px para así aprovechar los 10px disponibles en la pantalla. En cambio, si definimos el atributo *android:stretchMode=spacingWidth* lo que ocurrirá es que los espacios en blanco aumentarán a 10px cada uno para así aprovechar el espacio disponible en la pantalla.

Por otro lado, hay que decir que *GridView* funciona de manera muy similar a los controles de selección vistos anteriormente; es decir, utiliza un adaptador para proveer al control de las opciones que se desplegarán y que se define a partir del método *setAdapter();* de la misma forma utilizamos un *listener* para poder recuperar la opción que el usuario selecciona en la cuadrícula.