Diseño del Interfaz

Hemos visto que las apps de Android separan las 'piezas' que forman el programa separando el código fuente de todo lo demás. En este apartado vamos a centrarnos en las interfaces: los **layouts**, **widgets**, las **vistas**, los botones, etc.

Tienes que tener en cuenta que, aunque al final estés trabajando con xml, todos los elementos que conforman la interfaz de usuario de una app son en el fondo **clases**, y por tanto se compilan y se ejecutan normalmente. Obviamente, esto es transparente para nosotros, que nos limitamos a usar el IDE y simplemente colocamos botones en una ventana y a correr.

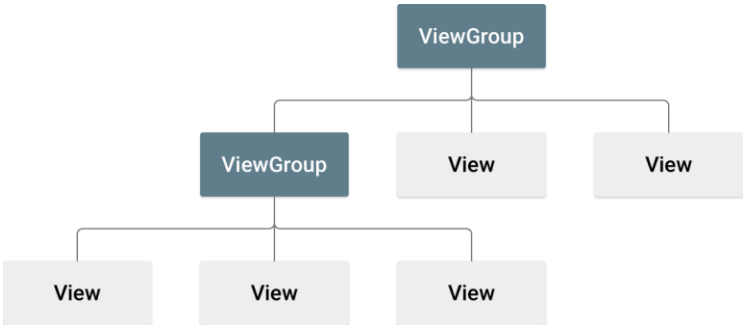
Vistas, Contenedores y Layouts

En Android, la interfaz de usuario (UI) es una jerarquía de *diseños* y *widgets*.

- Los *widgets* o vistas son subclases de **View**. Son los componentes de la UI del tipo botones, etiquetas, tablas y cuadros de texto. Ya has utilizado dos Widget en las prácticas anteriores: el **TextView** y el **Button**. Todos ellos tienen sus eventos, sus propiedades, etc. Si ya has creado interfaces gráficas en Java, todo esto te sonará-

  Si esto fuera una página HTML normal, te diría que los *widgets* son las etiquetas <button>, <label>, <input>, etc.

- Los *contenedores* son subclases de **ViewGroup**. Su objetivo es contener otros widgets y por supuesto, a otros contenedores. En general se usan para posicionar los distintos elementos en la pantalla.



  Si esto fuera una página HTML normal, te diría que los *contenedores* son las etiquetas <body>, <section>, , etc.

- Un *layout* es en realidad un contenedor. Su propósito es organizarlo todo en una única pantalla visible para el usuario.

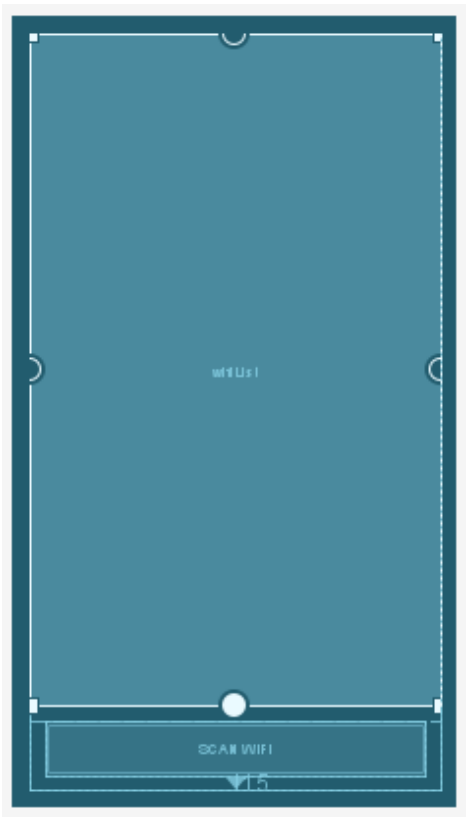
  Si esto fuera una página HTML, te diría que los *layout* son la etiqueta <HTML>

- Una actividad por tanto estará formada por una pantalla, que generalmente será un *layout* con todos sus *contendores* y *widgets*. Es posible que una activity trabaje con más de un *layout*, pero eso lo veremos más adelante.

Ejemplo: este fichero **layout.xml** define un interfaz formado por un listado (ViewGroup) y un botón (View) contenido en un RelativeLayout (ViewGroup).

```
<?xml version = "1.0" encoding = "utf-8"?>
<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"
    android:layout_margin = "16dp"
    android:orientation = "vertical"
    tools:context = ".MainActivity">
    <ListView
        android:id = "@+id/wifiList"
        android:layout_width = "match_parent"
        android:layout_height = "match_parent"
        android:layout_above = "@+id/scanBtn" />

    <Button
        android:id="@+id/scanBtn"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_alignParentBottom="true"
        android:layout_gravity="bottom"
        android:layout_marginStart="15dp"
        android:layout_marginTop="15dp"
        android:layout_marginEnd="15dp"
        android:layout_marginBottom="15dp"
        android:text="Scan WiFi" />
</RelativeLayout>
```



Lo habitual no es trabajar sobre el XML, sino empleando la herramienta visual del Android Studio. No obstante, puede ser necesario en ocasiones que tengamos que tocar el XML. Cada una de los *Atributos* que colocas en la herramienta visual para un *diseño* o un *widget* aparecerá en su lugar correspondiente del XML. Adicionalmente, los *diseños* o *widgets* que se colocan **dentro** de otro *diseño* aparecerán **dentro** de la etiqueta del *diseño* padre en el XML.

Las Vistas

Todos los componentes visuales en Android son subclases de View. Esto define básicamente un espacio rectangular en pantalla sobre el que podemos pintar cosas e interaccionar con el usuario. Cuando creamos un interfaz, generamos un fichero xml en el que definimos su layout, y después vamos creando los diferentes widgets o vistas que queremos utilizar. Una vez completada la interfaz, pasamos a definir la **activity** a la que está vinculada.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
```

La segunda línea del onCreate () se emplean para vincular la en layout a la activity. El método **setContentView ()** sirve precisamente para eso. Es posible incluso cambiar el layout de una activity dinámicamente en tiempo de ejecución, por ejemplo, al rotar el móvil.

Para referirnos a los diferentes layouts, widgets y contenedores (en realidad, a todo lo que hay en Resources) se emplea la **clase R**. En el caso que estamos tratando, estamos referenciando a un layout llamado activity_main.xml.

```
val textView : TextView = findViewById(R.id.textView)
val button : Button = findViewById(R.id.button)
```

Para poder usar cada uno de los diferentes widgets desde nuestra activity, es necesario referenciarlos. Para eso se emplea el método **findViewById ()**. En este caso, la referencia se hace mediante la id del widget. Por tanto, **siempre** hay que dar una id a cada uno de los widgets de nuestra interfaz, sin repetir el nombre.

El problema que tiene **findViewById ()** es que siempre retorna una View, por lo que será necesario hacer un casting al tipo de objeto que estamos esperando. Esto sucede tanto en Java como en Kotlin.

Finalmente, otro uso de la clase R podría ser este:

```
val textHola = getString(R.string.text_hello_world)
```

Las Vistas en Android

Android proporciona un conjunto de vistas o widgets por defecto. Y sí, es posible crear tus propios widgets, bajarte otros, extenderlos, etc. Los más comunes son:

- **TextView**: Una etiqueta de solo lectura. Permite multilínea, formateo, etc.
- **EditText**: Caja de texto editable. Permite multilínea, texto flotante, etc.
- **ListView**: Un contenedor que es agrupa widgets en forma de lista vertical.
- **Spinner**: Un combo box de toda la vida, desplegable al pulsarlo.
- **Button**: El botón de toda la vida.
- **Checkbo**x: Un botón de dos estados representado con un cuadradito.
- **RadioButton**: Como un checkbox, pero es un circulito. Se puede hace run **Radio-Group**, un contenedor de RadioButton en el que solamente se puede pulsar uno.
- **SeekBar**: Una barra de desplazamiento que puede tener cualquier valor.

Los Layouts

Existen diversos tipos de contenedores. Los más relevantes son:

- **AbsoluteLayout:** Básicamente es un layout que tiene un eje de coordenadas (X, Y). Para colocar cada widget en su interior le tienes que decir dónde ha de colocarlo. Es el menos flexible de todos.
- **Linear Layout:** Es un modelo de cajas, te permite alinear los Widgets en filas horizontales y/o verticales en una columna. Se puede ordenar una fila manipulado el atributo Gravedad.
- **RelativeLayout:** Define la posición de cada vista como *relativa* a otros elementos equivalentes o en posiciones relativas al RelativeLayout padre. Por ejemplo, puedes decir que un **Botón A** está a la izquierda de un **Botón B** y alineado al centro de su **RelativeLayout.**
- **ConstraintLayout:** Es una mejora del **RelativeLayout**, y permite una edición Visual. Define la posición de cada vista a partir de restricciones sobre el diseño principal y las vistas secundarias. Es decir, puedes colocar un **Botón A** y un **Botón B** dentro de un **ConstraintLayout** diciendo las distancias que hay entre ellos. Por ejemplo: Botón A está a 16 dp de la parte superior del ConstraintLayout, y a 16 dp de su parte izquierda. El Botón B se muestra a 16 dp a la derecha del Botón A, y se muestra alineada con la parte superior del Botón A.

- **ScrollView:** Si bien no es un layout per se, el ScrollView nos permite añadir un scroll a un layout para cuando éste resulta ser más grande que la pantalla del móvil. Usarlo en sencillo: añade el ScrollView, y anida dentro suyo el layout que quieras usar. Sólo admite uno.

Los Eventos

En Android, cuando realizamos cualquier clase de acción sobre un **Widget**, normalmente se 'dispara' un **Evento**. Estos **Eventos** tienen un método asociado que se ejecutará cuando se produzca ese **Evento**. Hasta ahora, estábamos relacionando el Evento **onClick** de un **Button** con un método que cambiaba el texto de un **EditText**, pero en realidad existen muchos más. Por ejemplo, onClick se lanza cuando se pulsa el Button; pero puedes asignarle en evento onTouch que se dispara cuando tocas el Button.

No vamos a repasar la totalidad de los Eventos de cada Widget porque nos podríamos pasar la vida aquí. En clase trabajaremos los más habituales, pero si quieres hacer algo *especialmente raro* te va a tocar buscar en Internet la forma de implementarlo.

Gestionando Eventos [JAVA]

Los elementos que toman parte en esto podríamos definirlos así de forma sencilla:

- **Evento**: acción del usuario sobre el dispositivo o sobre uno de los Widget.
- **Listener**: son componentes que se registran para recibir o tratar los eventos.

Los Widget hacen uso de estos Eventos mediante los escuchadores de eventos (**Event Listener**) y los manejadores de eventos (**Event Handler**). Los primeros son las Interfaces de la clase View que disponen de métodos callbak que será llamados por Android cuando se produzca la acción correspondiente. En concreto, nos estamos refiriendo a estas líneas de código:

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    private Button myButton = null;
    private TextView myTextView = null;

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

        myButton = (Button) findViewById( R.id.button );
        myButton.setOnClickListener(this);

        myTextView = (TextView) findViewById( R.id.textView );
    }
```

Los **Event Handler** son los métodos llamados cuando se genera un **Evento**. En concreto, nos estamos refiriendo a estas líneas de código:

```
public void onClick (View v){
    // el texto actual del TextView
    String textViewText = myTextView.getText().toString();
    // Texto 'Pulsa el boton' del strings.xml
    String textPulsa = getString(R.string.text_pulsa);
    // Texto 'Hola Mundo' del strings.xml
```

Gestionando Varios Eventos [JAVA]

El problema que tiene el código anterior es que, si asignamos el evento OnClick a varios widgets, **todos** ellos ejecutan el mismo Handler. Una forma de solucionarlo es la comentada en la práctica de la calculadora: diferenciar qué widget ha generado el evento con un if o un switch case.

Esto está bien cuando estás aprendiendo, pero es un error trabajar así. Si tenemos diferentes Widgets con diferentes eventos, acabaríamos con un activity con media docena de implements y con múltiples funciones para decidir qué widget ha generado qué. Y esto es un caos.

```
@Override
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.button1:
            func1();
            break;
        case R.id.button2:
            func2();
            break;
        case R.id.button3:
            func3();
            break;
        case R.id.button4:
            func4();
            break;
    }
}
```

Programáticamente hablando, cada **Widget** debería de tener su Evento, su **Listener** y su **Handler**. Punto. De esta forma, todo queda más limpio.

La forma correcta de programar un evento es mediante clases anónimas. Una **clase interna anónima** es una forma de clase que se declara y crea una instancia con una sola declaración (línea de texto). Como consecuencia, no guardamos referencia de esa clase, no hay un nombre para la clase que pueda usarse en otra parte del programa: es decir, es **anónimo**.

Explicado de otra forma, hacemos un new normal de la clase, pero no guardamos ese objeto en ninguna variable. Las clases anónimas se utilizan normalmente en situaciones en las que es necesario poder crear una clase pequeña que luego se va a pasar como parámetro. Dado que requiere mucho código, puedes ahorrártelo con una clase anónima. Esto normalmente se hace con una interfaz.

Por ejemplo:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return string1.toUpperCase().compareTo(string2.toUpperCase()
            }
        };
```

Todo este tocho de texto en realidad es **una sola línea de código**. De golpe y porrazo estamos diciendo que:

- Tenemos una variable pública y estática.
- La variable se llama CASE_INSENSITIVE.
- Su tipo de dato es Comparator<String>.
- Ese Comparator<String> es otra clase que, en vez de estar en un fichero por ahí perdido, te la defino en la misma línea.
- Esa **clase anónima** Comparator<String> tiene un método compare ().

Las clases internas anónimas también pueden basarse en clases. En este caso, la clase anónima extiende implícitamente la clase existente. Si la clase que se está extendiendo **es abstracta**, entonces la clase anónima **debe implementar** todos los métodos abstractos. También puede heredar y sobrecargar métodos no abstractos.

Una clase anónima no puede tener un **constructor explícito**. En su lugar, se define un constructor implícito que usa super(...) para pasar cualquier parámetro a un constructor en la clase que se está extendiendo. Por ejemplo:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

El constructor implícito para nuestra subclase anónima de SomeClass llamará a un constructor de SomeClass que coincida con la firma de llamada SomeClass (int, String). Si no hay ningún constructor disponible, obtendrá un error de compilación. Cualquier excepción lanzada por el constructor emparejado también es lanzada por el constructor implícito. Naturalmente, esto **no funciona** cuando se extiende una interfaz.

En Android se utilizan las clases anónimas para definir el comportamiento de cada uno de los Widgets. Es recomendable que te acostumbres a utilizarlas por muy pequeño que te parezca al App. En el ejemplo vemos cómo se asigna un evento individual a un botón concreto mediante clase anónima. Por supuesto, esto no impide añadir más eventos al mismo botón.

```
Button button= (Button) findViewById(R.id.button);
button.setOnClickListener( new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        // Codigo de lo que hace el boton

    }
} );
```

Práctica 7

Realiza la práctica propuesta en el documento **03 – Mejor Videojuego [Java]**. Debería de ser sencilla de realizar con lo que ya sabes.

Práctica 8

Realiza la práctica propuesta en el documento **03 – Mejor Videojuego [Kotlin]**. Debería de ser sencilla de realizar con lo que ya sabes.

Gestionando Eventos [KOTLIN]

Esto va a ser rápido. Haz lo mismo que con Java, asigna un evento individual mediante clase anónima al widget. Lo que pasa es que es más fácil todavía, dado que en el fondo **"Kotlin es Java resumido"**

```
findViewById<Button>( R.id.button ).setOnClickListener {

    // Codigo de lo que hace el boton

}
```

Práctica 9

Realiza la práctica propuesta en el documento **04 – Llamar Walter White [Java]**. Debería de ser sencilla de realizar con lo que ya sabes.

Práctica 10

Realiza la práctica propuesta en el documento **04 – Llamar Walter White [Kotlin]**. Debería de ser sencilla de realizar con lo que ya sabes.

Práctica 9

Realiza la práctica propuesta en el documento **05 – Llamar Walter Anónima [Java]**. Debería de ser sencilla de realizar con lo que ya sabes.

Práctica 10

Realiza la práctica propuesta en el documento **05 – Llamar Walter Anónima [Kotlin]**. Debería de ser sencilla de realizar con lo que ya sabes.

Práctica 11

[Kotlin] Crea una app que sea un login. El usuario deberá introducir el user & pass. Si coincide con el par "admin/admin" muestra un mensaje de bienvenida. Si no, de error. Usa clases anónimas.

Práctica 12

[Kotlin] Crea una app que permita obtener el nombre, apellido, DNI, sexo y fecha de nacimiento. El usuario deberá introducir los valores (usa widgets variados). Comprueba que:

- El nombre no sea inferior a 2 caracteres, mayor que 15 ni contenga números.
- El apellido no sea inferior a 2 caracteres, mayor que 15, no tenga espacios ni números.
- El DNI tiene que tener 8 números y una letra
- Al seleccionar el sexo, se muestra una imagen genérica de un hombre o una mujer.
- Si es menor de 20 años se muestra un mensaje de error.