

U.T.3 PROGRAMACIÓN PARA DISPOSITIVOS MÓVILES. INTERFAZ GRÁFICA DE USUARIO.



INTERFACES DE USUARIO. LAYOUT.

La interfaz de usuario es la ventana que permite al usuario interactuar con la aplicación, se define en los archivos XML del directorio `res/layout`.

Cada archivo xml describe un `layout` (una pantalla) y cada layout a su vez puede contener otros elementos, que permite especificar la disposición de los controles de un interfaz de usuario.

Cada layout tiene un `identificador` como recurso de la aplicación y podremos pedirle al sistema que nos construya el interfaz a partir de él.

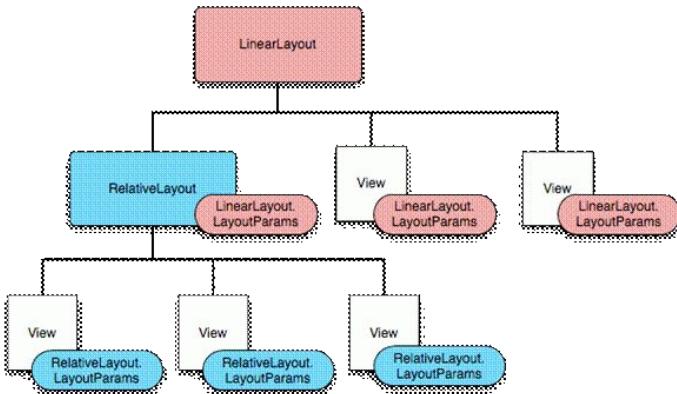
Por ejemplo en la IU principal de una aplicación, el layout se identifica como `R.layout.activity_main`, indica que el recurso está incluido en el archivo `R.java` y tiene asociado un identificador numérico. La sentencia utilizada para construir el interfaz desde el archivo java de la aplicación es:

```
setContentView(R.layout.activity_main);
```

Podremos tener diferentes versiones del mismo layout, por ejemplo, podríamos querer que el interfaz cambie si el dispositivo está en vertical (`portrait`) o en horizontal (`landscape`), o que en una ventana pidiendo datos personales pidiera el segundo apellido en España, pero no en Estados Unidos, donde debería pedir a cambio el middle name.

VISTAS. LAS CLASES VIEW Y VIEWGROUP

En la plataforma Android un interface de usuario se define usando un árbol de nodos **view** y **viewgroups**, como vemos en la imagen de abajo. El árbol puede ser tan simple o complejo como necesites hacerlo, y se puede desarrollar usando los widgets y layouts que Android proporciona o creando tus propios views.



Los objetos **View** son generalmente los elementos (widgets) de interfaz de usuario, tales como botones o campos de texto. Descienden de la clase **android.view.View**.

La clase **view** es útil como clase base para los **widgets**. La lista de widgets que puedes utilizar incluyen `TextView`, `EditText`, `InputMethod`, `Button`, `RadioButton`, `CheckBox`, y `ScrollView`.

Los objetos **ViewGroup** son contenedores invisibles que definen cómo se disponen los objetos que contienen. Descienden de la clase **android.view.ViewGroup**.

La clase **viewgroup** es la base de la clase **layouts**, que son subclases implementadas que proveen los tipos más comunes de los layouts de pantalla.

La clase Layout

Los **layout** heredan de la clase **ViewGroup** para gestionar la disposición de sus Views hijas o **widgets**. Se les conoce como **contenedores** y sirven para reorganizar los elementos de nuestra aplicación.

Existen una gran variedad de layouts, en función de su posicionamiento en la pantalla:

- ★ **LinearLayout**. Nos permiten alinear los elementos de nuestra aplicación en una única dirección, ya sea horizontal o vertical. La orientación predeterminada es horizontal.
- ★ **RelativeLayout**. En este caso todos los elementos van colocados en una posición relativa a otro. Aquí podemos jugar con las distancias entre elementos en la pantalla, la cual se expresa en pixels.
- ★ **TableLayout**. permite colocar los elementos en forma de tabla. Se utiliza el elemento **<TableRow>** para designar a una fila de la tabla. Cada fila puede tener uno o más puntos de vista. Cada vista se coloca dentro de una fila en forma de celda.
- ★ **FrameLayout**. Permite el cambio dinámico de los elementos que contiene.
- ★ **ConstraintLayout** permite posicionar y dimensionar widgets de una manera flexible.

ConstraintLayout

Con la llegada de **Android Studio 2.2**, Google nos presentó un nuevo layout que cambia radicalmente el modo empleado para diseñar interfaces gráficas. Está disponible como una biblioteca de soporte que puede usar en sistemas Android comenzando con API nivel 9 (Gingerbread)

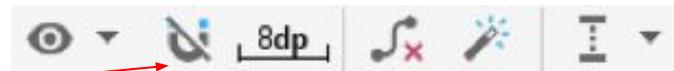
Uno de los elementos básicos para crear diseños en **ConstraintLayout** es el posicionamiento relativo de sus elementos entre si mismos, de manera que permite posicionar un widget dado en relación con otro, según los ejes horizontal y vertical:

- Eje horizontal: lados izquierdo, derecho, inicio y final
- Eje vertical: lados superior, inferior y línea de base de texto

El concepto general es restringir un lado dado de un widget a otro lado de cualquier otro widget.

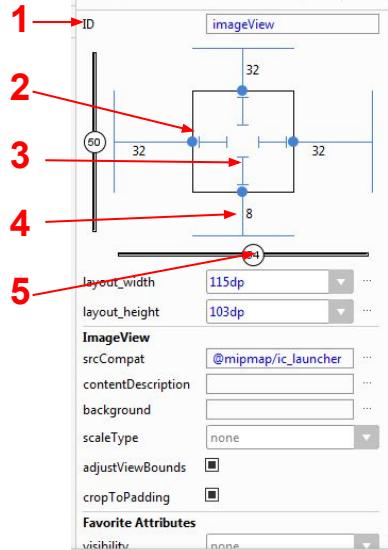
La manera de trabajar con este Layout es principalmente en la vista diseño, en lugar de utilizar el fichero XML. El siguiente enlace muestra un vídeo para crear una interfaz con ConstraintLayout.

<https://youtu.be/XamMbnzI5vE>



Incluye un modo llamado **Autoconnect**, el cual una vez activado, cada vez que añadamos una vista al layout permitirá crear dos o más restricciones (*constraint*) para cada vista, para asignarlo al espacio donde nosotros la hemos soltado.

Herramientas vista diseño ConstraintLayout



1 establece el tamaño de la vista en una proporción.

2 eliminar constraint: Se elimina la restricción para este punto de anclaje.

3 establecer alto/ancho: Para cambiar la forma en la que se calcula las dimensiones de la vista, pulsa en este elemento.

Existen tres posibilidades:

ajustar a contenido: equivale al valor warp_content.



ajustar a constraint: equivale a poner 0dp.



tamaño fijo: equivale a poner un valor concreto de dp.



4 establecer margen: Podemos cambiar los márgenes de la vista.

5 Sesgo del constraint: Ajustamos cómo se reparte la dimensión sobrante.



1 Ocultar constraint: Elimina las marcas que muestra las restricciones y los márgenes existentes.

2 Autoconectar: Al añadir una nueva vista se establecen unos constraint con elementos cercanos de forma automática.

3 Definir márgenes por defecto

Borrar todos los constraint: Se eliminan

4 todas las restricciones del layout.

5 Crear automáticamente constraint: Dada una vista seleccionada, se establecen unos constraint con elementos cercanos de forma automática.

Empaquetar / expandir:

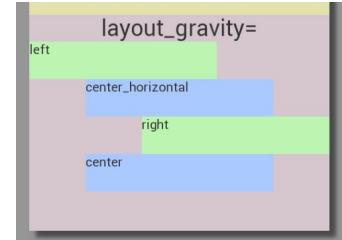
6 Alinear:

Añadir línea de guía:

Atributos de la vista (view) respecto del Layout

layout_width y layout_height permite ajustar el ancho y el alto de la vista.

- **match_parent** ajusta al tamaño máximo posible.
- **wrap_content** ajusta al tamaño del contenido de la vista.
- **200 px(pixels), 200dp (pixels/densidad)** tamaño concreto de la vista

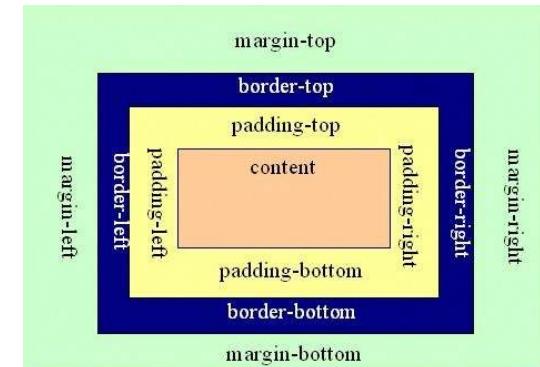


layout_gravity Centra o justifica la vista dentro del layout.

layout_weight. En un LinearLayout que dispone de espacio sin utilizar permite repartir este espacio proporcionalmente entre los componentes del layout, según el valor indicado en este atributo.

layout_margin para el global, y layout_margin_bottom, layout_margin_left, layout_margin_right, layout_margin_top, establecen un margen exterior a la vista.

padding, padding_bottom, padding_left, padding_right, padding_top establecen el margen interior de la vista



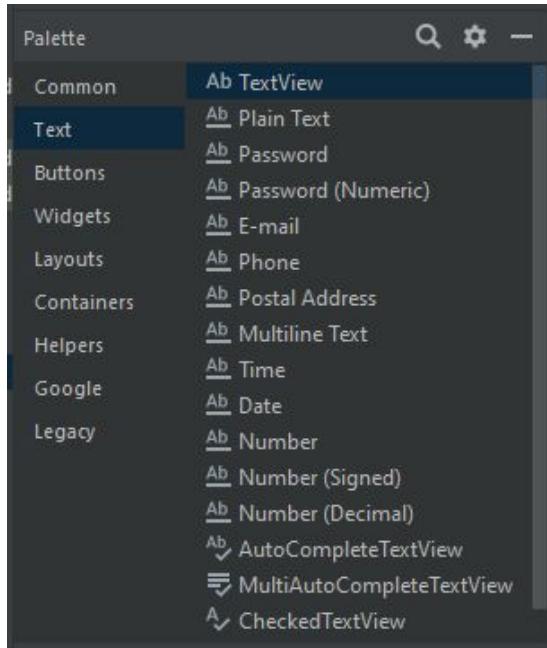
Componentes o widgets

Los componentes o widgets están disponibles en la **paleta del Graphical Layout**. Cada componente estará asociado a una clase Java y tiene sus propios atributos y eventos. Entre los más usados se encuentran los siguientes controles:

- ★ **Etiqueta de texto:** `android.widget.TextView`
- ★ **Botón:** `android.widget.Button`
- ★ **Caja de texto:** `android.widget.EditText`
- ★ **Calendario:** `android.widget.CalendarView`
- ★ **CheckBox:** `android.widget.CheckBox`
- ★ **Selección de fechas:** `android.widget.DatePicker`.
- ★ **Botón con imagen:** `android.widget.ImageButton`
- ★ **Imagen:** `android.widget.ImageView`
- ★ **Información de progreso:** `android.widget.ProgressBar`
- ★ **RadioButton:** `android.widget.RadioButton`
- ★ **Menú desplegable:** `android.widget.Spinner`

TextView

El control **TextView** o *etiquetas de texto*. Se utiliza para mostrar un determinado texto al usuario.



Algunas de sus propiedades son:

android:id: indica el identificador del textView. Permite acceder al componente desde java.

android:text establece el texto del control

android:background color de fondo

android:textColor color del texto

android:textSize tamaño de la fuente

android:textStyle estilo del texto: negrita, cursiva, ...

android:typeface permite cambiar el tipo de fuente: normal, sans, serif,...

Acceso a TextView desde java

Algunos de los métodos Java usados con TextView son:

`findViewById` : enlaza un recurso de la interfaz de usuario de una aplicación, con una variable en nuestro código. Esto nos permite acceder a ese recurso para cambiar alguna de sus propiedades.

`getText()` retorna el texto contenido en la etiqueta.

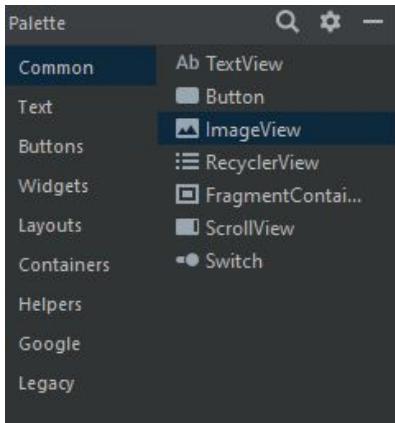
`setText(String t)` modifica el texto de la etiqueta.

Por ejemplo: Para cambiar el texto de una etiqueta de la actividad en tiempo de ejecución.

```
//asocia el recurso textView1 con la variable etiqueta  
TextView etiqueta= (TextView)findById(R.id.textView1);  
  
//almacena en la variable texto, el texto que contiene la etiqueta.  
String texto = etiqueta.getText().toString();  
  
texto += "123";  
  
//modifica el texto mostrado en la etiqueta  
etiqueta.setText(texto);
```

ImageView

El control **ImageView**. Se utiliza para mostrar una imagen en la interfaz de usuario. Se puede asignar una imagen a un ImageView desde un archivo o desde los recursos del proyecto.



Algunas de sus propiedades son:

android:id: indica el identificador del ImageView. Permite acceder al componente desde java.

android:visibility puede tomar 3 valores: visible, invisible, gone.

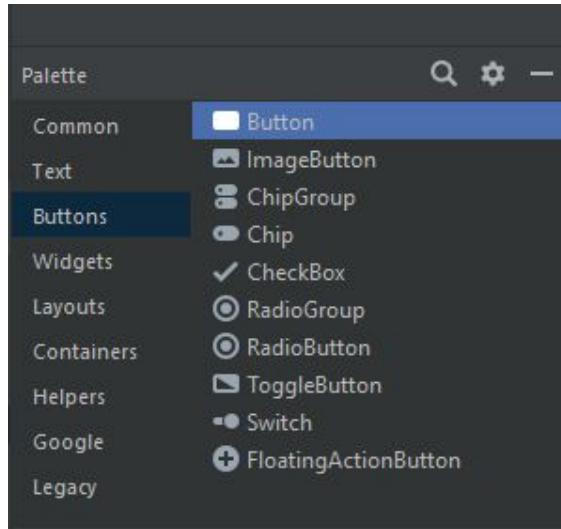
app:srcCompat indica la imagen a visualizar como recurso drawable o mipmap

```
<ImageView  
    android:id="@+id/imageView3"  
    android:layout_width="0dp"  
    android:layout_height="106dp"  
    android:layout_marginStart="84dp"  
    android:layout_marginTop="277dp"  
    android:layout_marginEnd="41dp"  
    android:visibility="visible"  
    app:layout_constraintEnd_toStartOf="@+id/button"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    app:srcCompat="@mipmap/ic_launcher" />
```

Pick a Resource. muestra todos los recursos de imagen disponibles para tu app. Drawable y Mip map.

Button

El control **Button** es un elemento de la IU en el que el usuario puede pulsar o hacer clic para realizar una acción.



Otras propiedades:

android:id: indica el identificador del botón. Se puede representar como:

@id/<identificador> Si no se pone el +, tendremos que definir nosotros el identificador de alguna forma en el fichero R.java.

@+id/<identificador> Si se indica el + (lo más habitual), le pedimos a **aapt** (la herramienta que procesa los recursos) que nos añada el identificador automáticamente a la clase **R**.

android:onClick Llamada a un método que permite realizar una acción al pulsar el botón
android:backgroundTint color de fondo.

android:drawableLeft para añadir un ícono a la izquierda del texto de un botón

Evento onClick de Button con XML

Es posible especificar un método sobre el evento `onClick` directamente en el recurso, y que Android haga la asociación automáticamente.

```
    android:onClick="botonPulsado"
```

al pulsar el botón se ejecutará el método `botonPulsado`

Para que esto funcione, el método de la actividad establecido en el atributo `android:onClick` del XML debe cumplir:

- Ser público
- No devolver ningún valor (`void`)
- Recibir un único parámetro de tipo `View`, que contendrá el elemento pulsado.

```
<Button  
    android:id="@+id/button"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:backgroundTint="#00BCD4"  
    android:text="¡¡PULSAME!!"  
    android:onClick="botonPulsado"  
    android:textColor="#FFC107"  
    android:textSize="20sp"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />
```

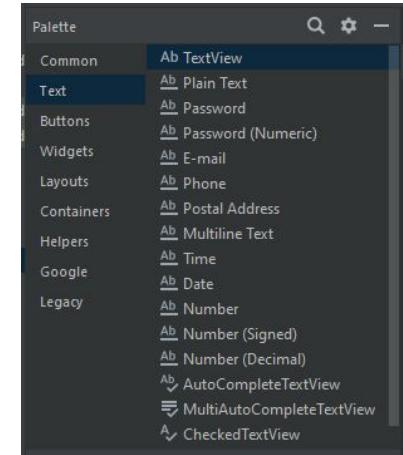
```
public void botonPulsado(View v) {  
    //v es el botón que se ha pulsado  
    Button b = (Button) v;  
    b.setText(numPulsados);  
}
```

EditText

Las cajas de texto o Plain Text son controles donde el usuario puede introducir texto en tiempo de ejecución.

La propiedad más interesante a establecer, además de su posición, tamaño y formato, es el texto a mostrar, atributo `android:text`. y la propiedad `android:inputType`. Esta propiedad indica el tipo de contenido que se va a introducir en el cuadro de texto como:

- una dirección de correo electrónico (`textEmailAddress`),
- una contraseña (`textPassword`)
- un número genérico (`number`),
- un número de teléfono (`phone`),
- una dirección web (`textUri`),
- un texto genérico (`text`).



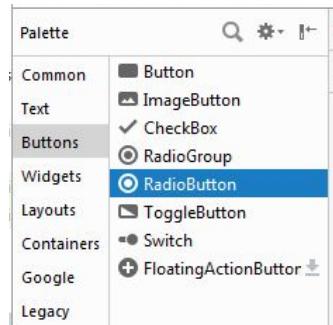
```
<EditText
```

```
    android:id="@+id/editText"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:inputType="text"  
    android:text="@string/escribeAlgo" />
```

Evento para leer un valor de un EditText desde java

```
//asocia el recurso cajaTexto con la variable et  
EditText et = (EditText) findViewById(R.id.cajaTexto);  
//evento que se lanza al pulsar enter en la caja de texto  
et.setOnKeyListener(new android.view.View.OnKeyListener() {  
    public boolean onKey(View v, int keyCode, android.view.KeyEvent event) {  
        // Han pulsado (o soltado) una tecla... y ha sido una pulsación de enter  
        if ((event.getAction() == android.view.KeyEvent.ACTION_DOWN) &&  
            (keyCode == android.view.KeyEvent.KEYCODE_ENTER)) {  
            //almaceno el valor de la caja de texto en la variable numero  
            int numero=Integer.parseInt(et.getText().toString());  
            ... return true;  
        }else return false;}  
});
```

RadioButton



Los botones de radio ([RadioButton](#)) se agrupan en [RadioGroup](#), que se encarga de garantizar que sólo uno de los botones de radio esté activo en un determinado momento.

La clase [RadioGroup](#) hereda de [LinearLayout](#), por lo que podremos orientarlo tanto horizontal como verticalmente. Se espera que en el interior de un [RadioGroup](#) sólo haya objetos de la clase [RadioButton](#), aunque no es obligatorio.

Por su parte, el atributo más destacado de los [RadioButton](#) es [checked](#), que nos indica si está o no seleccionado.

```
<RadioGroup
```

```
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="38dp"  
    android:id="@+id/radiogroup">
```

```
<RadioButton
```

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/Android"  
    android:id="@+id/radio1"  
    android:textSize="20sp" />
```

```
</RadioGroup>
```

Acceso a RadioButton desde java

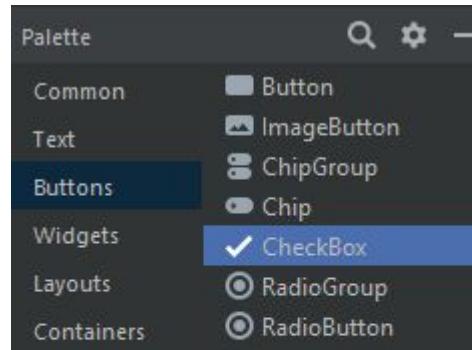
```
//asocia el recurso radioGroup1 con la variable rg  
  
RadioGroup rg = (RadioGroup)findViewById(R.id.radioGroup1);  
  
//recoge en el switch el id del boton de radio activado  
  
switch (rg.getCheckedRadioButtonId())  
{  
    case R.id.radio0:  
        ...  
        break;  
    case R.id.radio1:  
        ...  
        break;  
    ...  
}  
  
//también podemos llamar directamente al evento  
rg.setOnCheckedChangeListener(new  
    RadioGroup.OnCheckedChangeListener()  
{  
    public void onCheckedChanged(RadioGroup group,  
        int checkedId) {  
        // Hacer algo...  
    }  
});
```

CheckBox

Una casilla de verificación es un control tipo botón especial que tiene dos posibles estados, marcado y desmarcado que podremos inicializar con la propiedad `android:checked` marcado (`true`) o desmarcado (`false`).

En el código de la aplicación podremos hacer uso de los métodos `isChecked()` para conocer el estado del control, y `setChecked(estado)` para establecer un estado concreto para el control.

En cuanto a los posibles eventos que puede lanzar este control, el más interesante, para detectar los cambios de estado del checkBox, es `onCheckedChanged`.

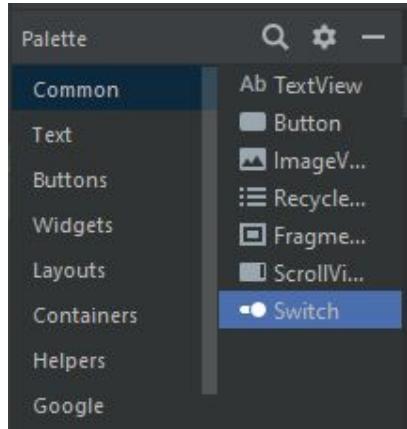


```
<CheckBox  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/camara"  
    android:id="@+id/checkBox"  
    android:layout_below="@+id/radiogroup"  
    android:layout_alignParentLeft="true"  
    android:layout_alignParentStart="true"  
    android:layout_marginTop="42dp" />
```

Acceso a CheckBox desde java

```
//asocia el recurso diaSemana con la variable cb  
CheckBox cb = (CheckBox) findViewById(R.id.diaSemana);  
//evento que se lanza al activar una casilla de verificación  
cb.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener()  
{  
    public void onCheckedChanged(CompoundButton buttonView,boolean isChecked) {  
        // Hacer algo! Sabemos quién ha sido pulsado y su estado  
    }  
});  
//también podemos saber si un checkBox está activado  
if (cb.isChecked())...
```

Switch



Un Switch es un widget de alternancia de dos estados. Los usuarios pueden arrastrar el "pulgar" del interruptor hacia atrás y adelante para seleccionar una de las dos opciones o simplemente toque el interruptor para alternar entre opciones.

Al igual que checkBox podemos utilizar tres métodos de la interfaz Checkable: `isChecked()` para determinar si está activo, `setChecked()` para cambiar el estado y `toggle()` para invertir el valor del estado actual.

Algunas de sus propiedades son:

`android:text`: indica el texto del switch

`android:textOn` texto que se muestra cuando el Switch está marcado

`android:textOff` texto que se muestra cuando el Switch no está marcado.

```
<Switch  
    android:id="@+id/switch1"  
    android:layout_width="232dp"  
    android:layout_height="36dp"  
    android:text="Activar Bluetooth"  
    android:textOff="Desactivado"  
    android:textOn="Activado"  
    app:layout_constraintBottom_toTopOf="@+id/  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="pare  
    app:layout_constraintTop_toTopOf="parent"
```

EVENTOS DE USUARIO

Cuando el usuario interacciona con la interfaz, y por ejemplo pulsa un botón, se generan eventos. La aplicación debe especificar qué acciones se deben llevar a cabo cuando se producen determinados eventos.

A los objetos que están pendientes de los eventos que se producen se los llama *Listeners*.

Android captura los eventos de usuario y se los pasa a la clase encargada de recogerlos. Por lo general va a ser un objeto tipo *View* el que recogerá estos eventos por medio de dos técnicas alternativas:

Los escuchadores de eventos (*Event Listener*)

Los manejadores de eventos (*Event Handler*).

Escuchador de eventos o Event Listener

Un Escuchador de eventos o **Event Listener** es una interface de la clase [View](#) que contiene un método **callback** que ha de ser registrado. Cada Escuchador de Eventos tiene solo un método callback, que será llamado por Android cuando se produzca la acción correspondiente. Tenemos los siguientes escuchadores de eventos:

onClick() Método de la interfaz [View.OnClickListener](#). Se llama cuando el usuario pulsa un elemento.

onLongClick() Método de la interfaz [View.OnLongClickListener](#). Se llama cuando el usuario pulsa un elemento durante más de un segundo.

onFocusChange() Método de la interfaz [View.OnFocusChangeListener](#). Se llama cuando el usuario navega dentro o fuera de un elemento.

onKey() Método de la interfaz [View.OnKeyListener](#). Se llama cuando se pulsa o se suelta una tecla del dispositivo.

onTouch() Método de la interfaz [View.OnTouchListener](#). Se llama cuando se pulsa o se suelta o se desplaza en la pantalla táctil.

¿Cómo crear un escuchador de eventos?

Existen 2 alternativas:

1. Crear un escuchador de eventos con la clase `setOnClickListener()` y gestionarlo mediante un objeto anónimo de la clase `OnClickListener()` que implementará dentro el método `onclick()`

```
protected void onCreate(Bundle savedInstanceState) {  
    ...  
    Button boton = (Button)findViewById(R.id.boton);  
    boton.setOnClickListener( new OnClickListener() {  
        public void onClick(View v) {  
            // Acciones a realizar  
        }  
    });  
    ...  
}
```

Para buscar un control por el identificador de su recurso, se utiliza `findViewById(id)`, que devuelve un objeto de la clase `android.view.View`, padre de todos los controles.

¿Cómo crear un escuchador de eventos?

2. Implementar la interfaz `OnClickListener` como parte de tu clase y recoger los eventos en el método `onClick()`. Esta alternativa es la recomendada por Android, al tener menos gasto de memoria.
A continuación se muestra un ejemplo:

```
public class Ejemplo extends Activity implements OnClickListener{  
    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        Button boton = (Button)findViewById(R.id.boton);  
        boton.setOnClickListener(this);  
    }  
    public void onClick(View v) {  
        int id=v.getId();  
        if(id==R.id.boton){  
            // Acciones a realizar  
        }else if(id==R.id.boton2){  
            ...  
        }  
    }  
}
```

En el caso de que hubiera varios componentes de la interfaz: otros botones, etiquetas, imágenes, etc. que quieran gestionar este evento `onClick` en el método tendremos que obtener el id del recurso del componente pulsado y establecer la acción a realizar para cada caso.

Manejadores de eventos. Event Handlers

Sólo puede definirse dentro de una clase `View`, su uso es más sencillo ya que no necesita implementar un interface, ni registrar el método `callback`. Podrás utilizar varios métodos (`callback`) directamente como manejadores de eventos por defecto (`Event Handlers`).

En esta lista se incluye:

`onKeyDown (int KeyCode, KeyEvent e)` → Llamado cuando una tecla es pulsada.

`onKeyUp (int KeyCode, KeyEvent e)` → Cuando una tecla deja de ser pulsada.

`onTrackballEvent(MotionEvent me)` → Llamado cuando se mueve el trackball.

`onTouchEvent(MotionEvent me)` → Cuando se pulsa en la pantalla táctil.

`onFocusChanged(boolean obtengoFoco, int direccion, Rect prevRectanguloFoco)` →
Llamado cuando cambia el foco.

Estos métodos pueden ser sobrescritos para poder personalizar nuestra aplicación.

onKeyDown

`onKeyDown(int KeyCode, KeyEvent e)` → Llamado cuando una tecla es pulsada.

El primer parámetro es un entero que identifica el código de la tecla pulsada.

Algunos de los códigos de los botones en un móvil:

Botón de Encendido - `KEYCODE_POWER`

Botón de Retorno - `KEYCODE_BACK`

Botón de Menu - `KEYCODE_MENU`

Botón de Hogar - `KEYCODE_HOME`

Botón de Búsqueda - `KEYCODE_SEARCH`

Botón de Cámara - `KEYCODE_CAMERA` ...

El segundo parámetro `KeyEvent e` permite obtener información sobre el evento, como por ejemplo, cuando se produjo.

```
public boolean onKeyDown (int codigoTecla, KeyEvent evento) {  
    super.onKeyDown(codigoTecla, evento);  
    //al pulsar la tecla  
    if (codigoTecla) == KeyEvent.KEYCODE_DPAD_UP  
        { //acciones a realizar  
        return true;  
        }  
    return false;  
}
```

El método ha de devolver un valor booleano, verdadero, si consideramos que la pulsación ha sido procesada por nuestro código, y falso, si queremos que otro manejador de evento reciba la pulsación.

Fragment

Los Fragments nacen para solucionar el problema de **adaptar una interfaz de usuario programada para un móvil en otros dispositivos de mayor tamaño, como tablets.**

Un Fragment como su propio nombre indica representa **una porción, un fragmento, de espacio de la interfaz de usuario** que puede añadirse o eliminarse de una interfaz de forma independiente al resto de elementos de la actividad, y que por supuesto puede reutilizarse en otras actividades. Esto va a permitir que podamos dividir una interfaz en varias porciones con la finalidad de diseñar diversas configuraciones de pantalla, dependiendo de su tamaño y orientación, sin tener que duplicar código en ningún momento.

La utilización de los fragment aparece a partir de la **versión 3.0 de Android.**

Tipos de Fragment

Se pueden considerar dos tipos de fragment en función de cuando los añadimos a nuestra aplicación:

Fragment Estáticos: son los declarados en layout mediante el widget `<FragmentContainerView>`, como si fueran cualquier control y que no se definen en el `AndroidManifest.xml`. Se utilizan principalmente para reutilizar código en actividades.

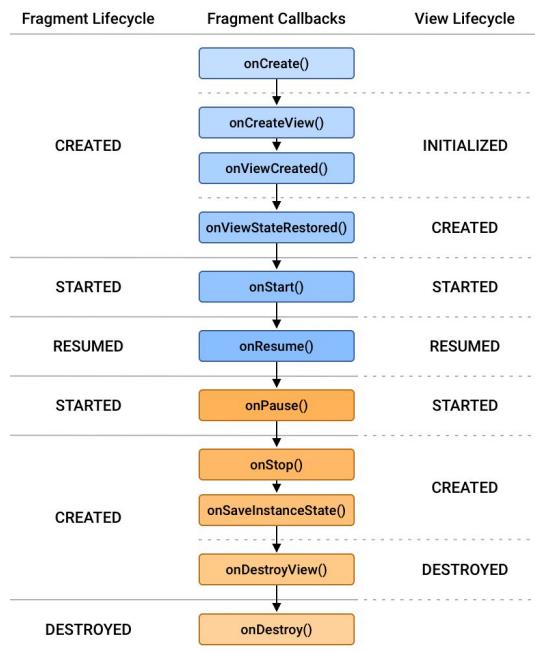
Fragment Dinámicos: que pueden insertarse o eliminarse de la interfaz en tiempo de ejecución, es decir mientras la activity está activa, adaptándose al tipo de pantalla u orientación. Son los más interesantes.

Android dispone de algunos tipos de Fragment específicos según su uso, que son: `DialogFragment`, `ListFragment`, `WebViewFragment` y `PreferenceFragment`

Ciclo de vida de un Fragment

Un fragment tienen su propio ciclo de vida integrado en el de la actividad a la que se conectan. Por lo que debe tener asociado, además del layout, su propia clase java, que debe extender de la clase **Fragment** y sobreescibir generalmente alguno de los métodos del ciclo de vida:

El método más importante del ciclo de vida de un fragment es **onCreateView()** que equivale al **onCreate()** de la actividad, es donde mostramos la vista asociada al fragment.



```
public override View OnCreateView (LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    return inflater.Inflate(Resource.Layout.Ejemplo_Fragment, container, false); }
```

LayoutInflater : permite construir una vista a partir de un Layout.

ViewGroup : contenedor donde incluirán el fragment.

Bundle : información "persistente" para reconstruir el contenido.

Métodos del ciclo de vida de un Fragment

- `onAttach (Actividad)` llamada una vez que el fragmento se asocia con su actividad.
- `onCreate (Bundle)` llamado para hacer la creación inicial del fragmento.
- `onCreateView (LayoutInflater, ViewGroup, Bundle)` crea y devuelve el layout asociado con el fragmento.
- `onViewStateRestored (Bundle)` el estado guardado de la jerarquía de vistas ha sido restaurado.
- `onStart ()` se hace visible el fragmento para el usuario
- `onResume ()` el fragmento permite la interacción con el usuario
- `onPause ()` fragmento ya no está interactuando con el usuario,
- `OnStop ()` fragmento ya no es visible para el usuario
- `onDestroyView ()` limpieza de las vistas del fragment
- `OnDestroy ()` limpieza del estado del fragment
- `onDetach ()` justo antes de que el fragment deje de estar asociado con su actividad.