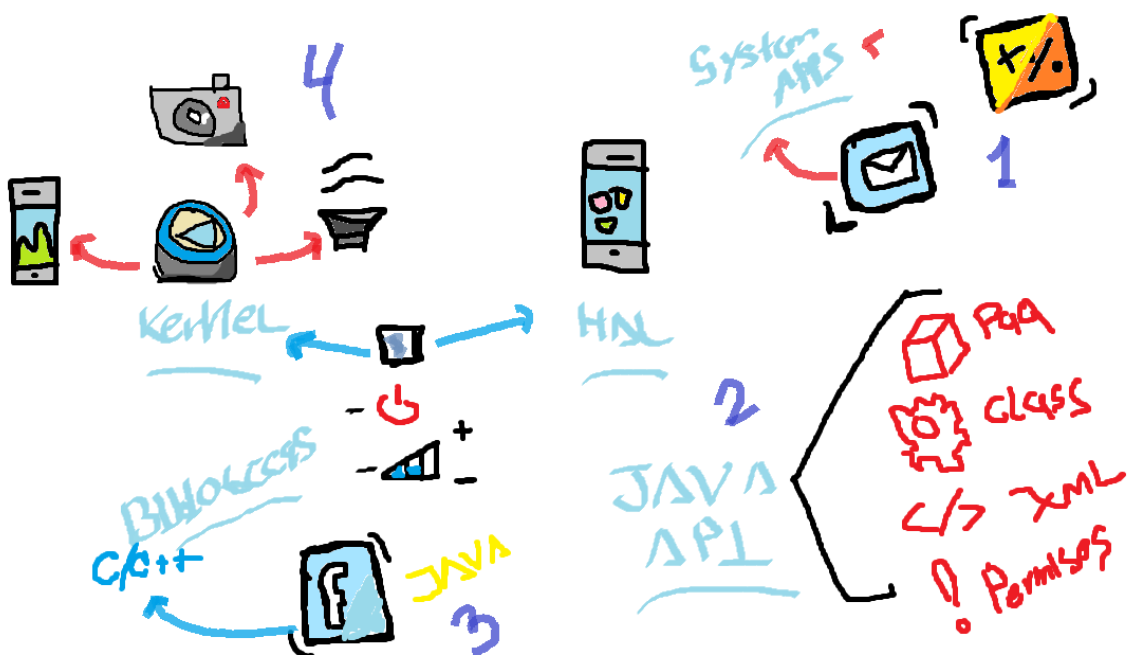


# Arquitectura de Android.

1. **Kernel** : Basado en el núcleo de Linux, es el sector que gobierna el hardware, es donde están los drivers de la cámara, sonido, video, etc.
2. **HAL (Hardware Abstraccion Layer)** : Es un puente entre el hardware y el software por lo que permite al sistema operativo Android para poder hacer llamadas a la capa de kernel para poder utilizar los drivers y poder utilizar las bibliotecas.
  - Encender.
  - Apagar.
  - Tomar fotos.
  - Subir volumen.
3. **Bibliotecas C/C++ Nativas y ART** : Aplicación escrita en JAVA debe ser interpretada a lenguaje maquina mediante .
  - Android Runtime (api >= 21).
  - Dalvik (api < 21).

Componentes y servicios que requieren de bibliotecas nativas escritas en C/C++, no necesitan ser interpretadas.
4. **JAVA API Framework** : Es un conjunto de;
  - Paquetes
  - Clases
  - XML
  - Intents
  - Permisos

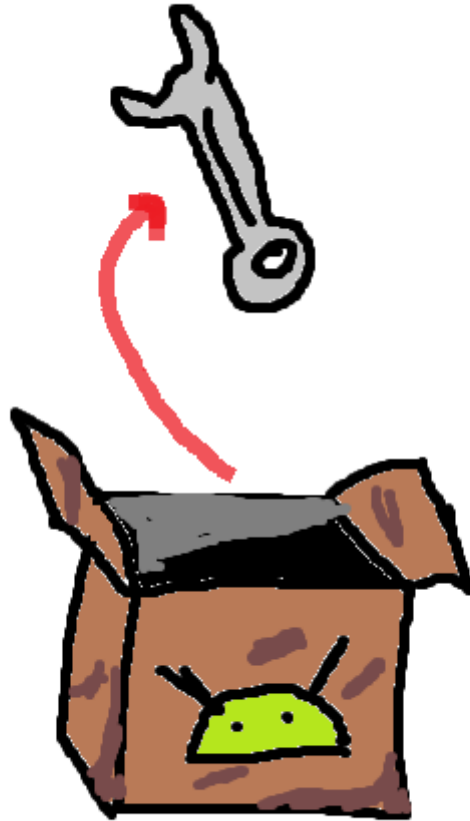
Cada versión de Android tiene un nivel de api único.
5. **System Apps** : Son aplicaciones que ya están incorporadas de fabrica como la aplicación del correo o la calculadora.



## Android SDK.

Es un conjunto de herramientas de desarrollo. Comprende un depurador de código, biblioteca, un simulador de teléfono basado en QEMU.

Cada vez que Google lanza un nuevo SDK que los desarrolladores deben descargar e instalar. Estas herramientas permiten que el proceso de desarrollo fluya sin problemas, desde el desarrollo y la depuración de programas hasta el empaquetado.



## AVD Manager.

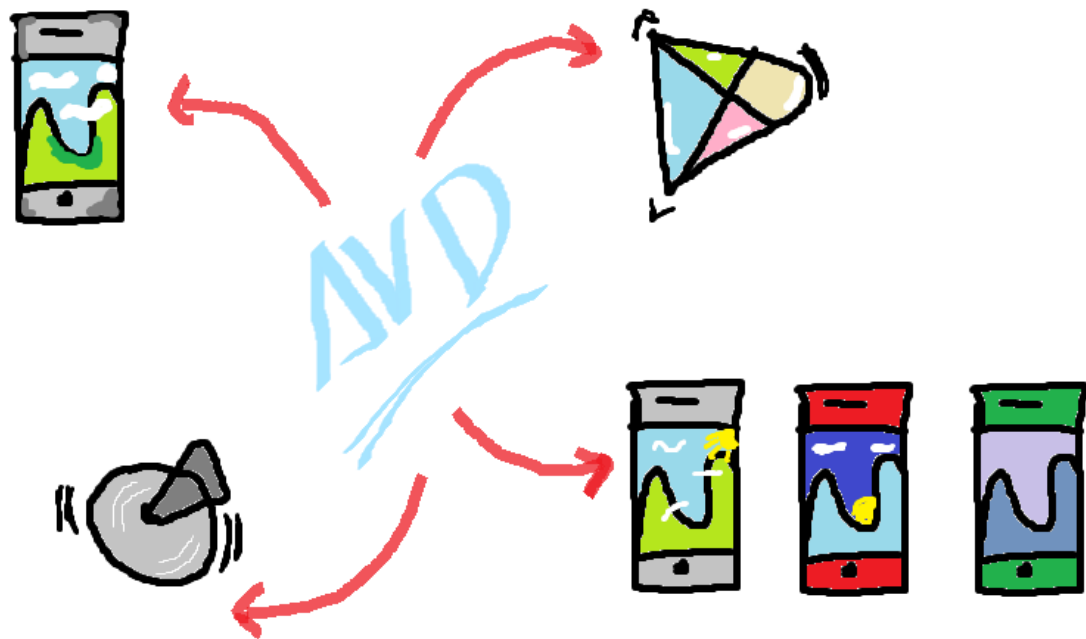
---

Configuración que define las características de un teléfono o una Tablet Android, o un dispositivo Wear OS, Android TV o Automotive OS, que se va a emular. El administrador permite crear y administrar los AVD.

Tools > AVD Manager.

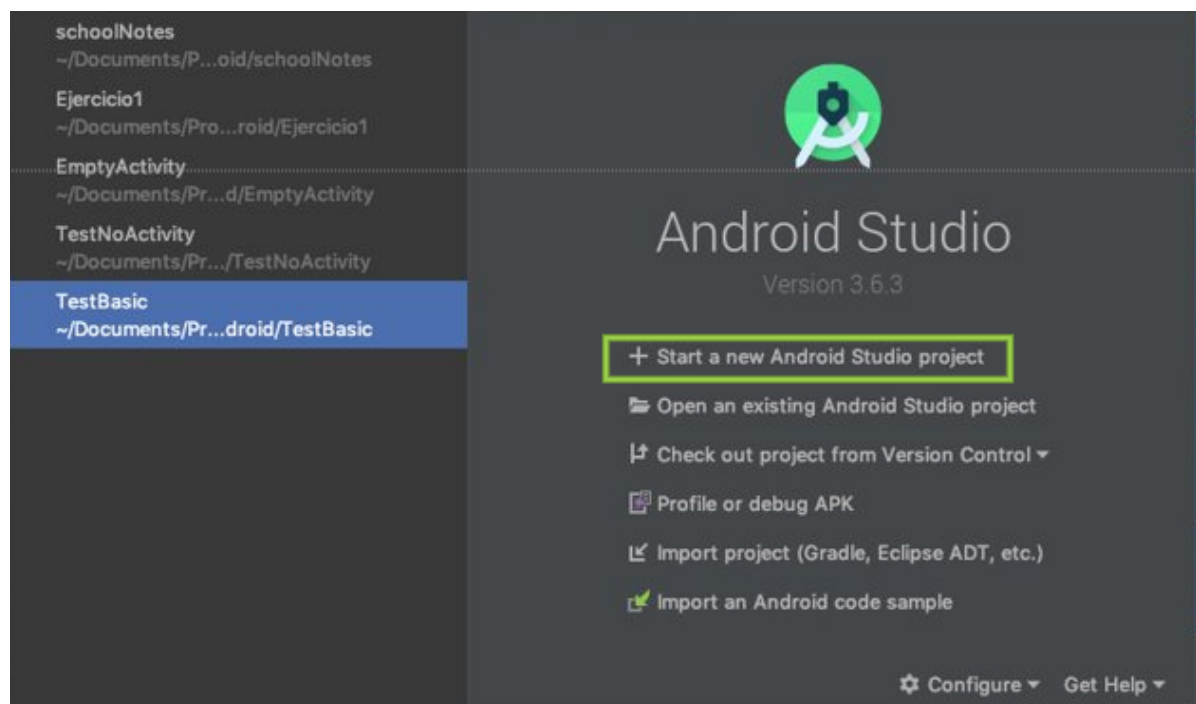
Un AVD cuatro cosas:

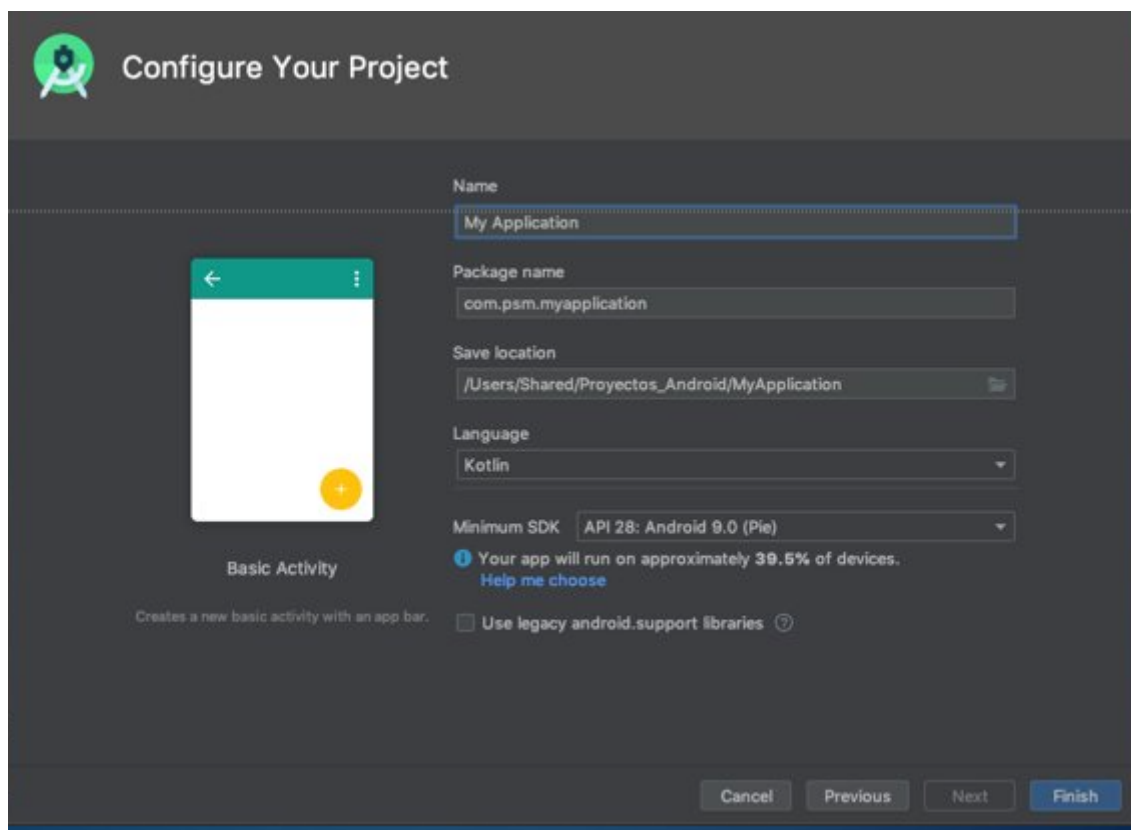
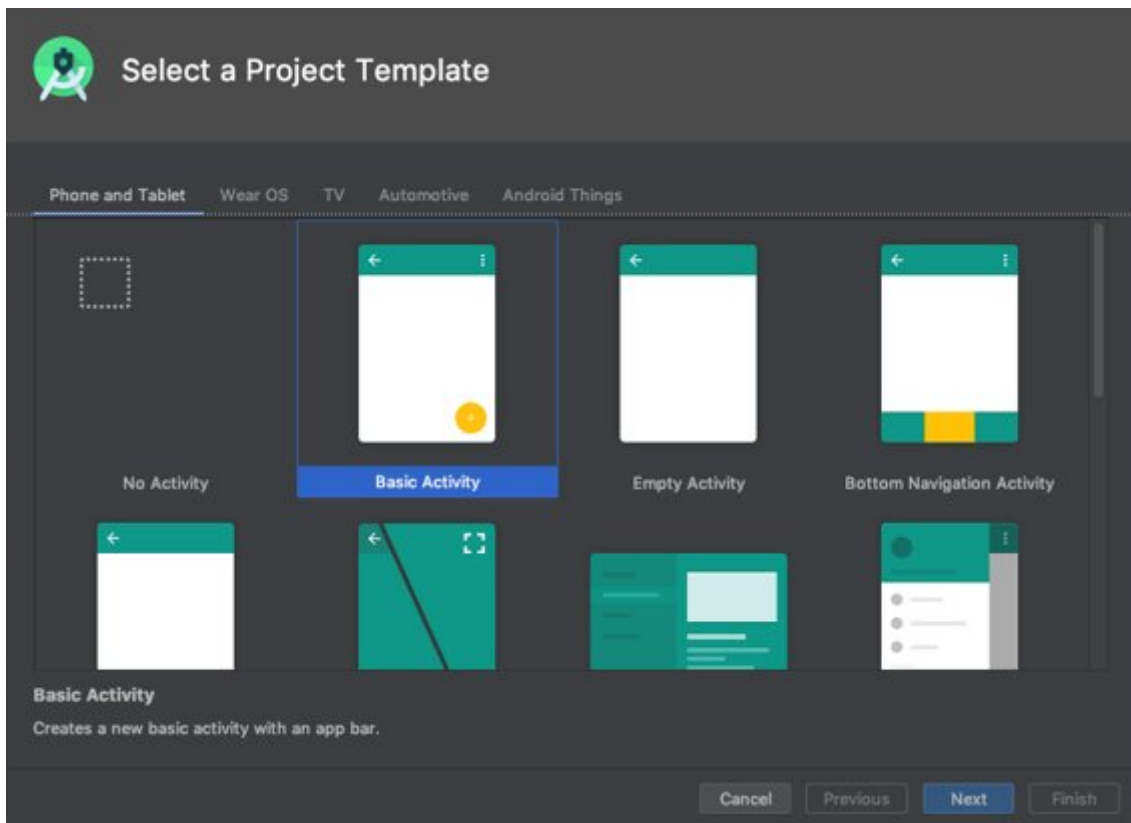
- **Perfil de hardware.** Define características de un dispositivo que se envía desde fábrica (Puede contener o no la *Play store*).
- **Imagen del sistema.** Hay varias imágenes, las *API de Google* incluyen acceso a *Google play*, Una imagen etiquetada con el logotipo de *Google play* en la columna *Play store* incluye un app de *Google play* acceso a los servicios de *Google play* del dialogo *Extended controls*.
- **Área de almacenamiento.** Tiene un área específica al almacenamiento en la máquina de desarrollo.
- **Máscara.** Especifica la apariencia del dispositivo.



## Interfaz de Android .

### Creación del proyecto.



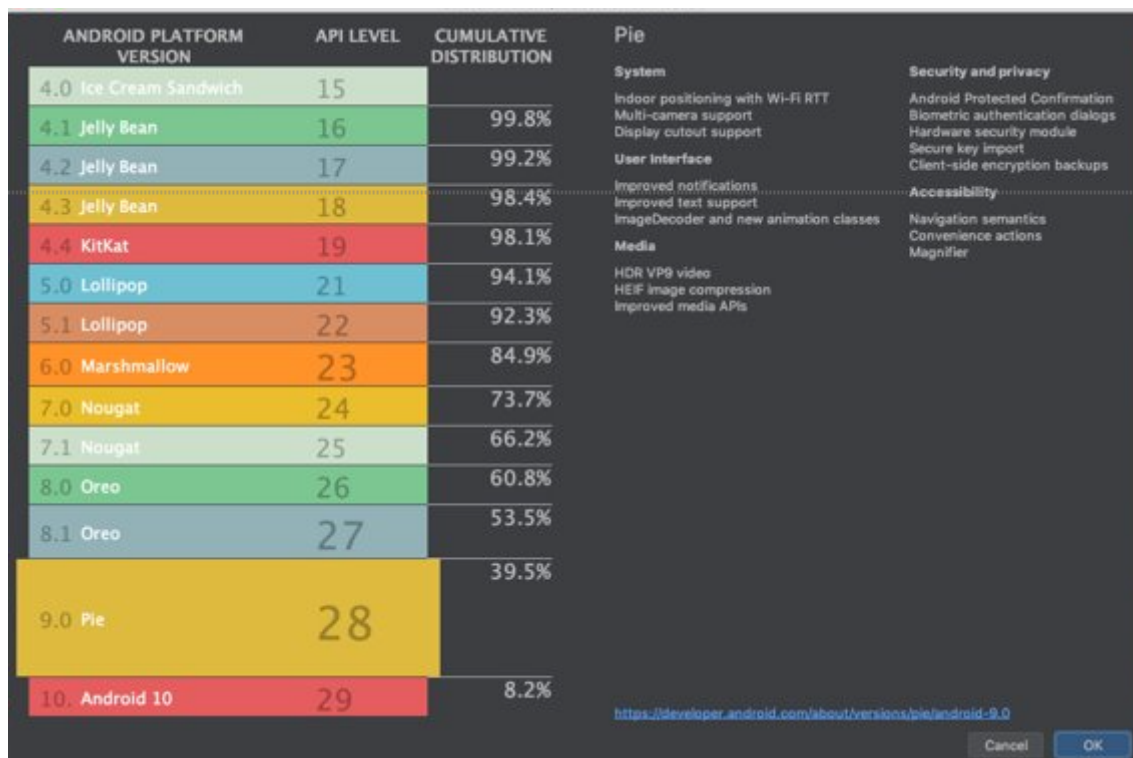


## Minimum API level.

Este valor especifica el mínimo nivel de API que requiere la aplicación no podrá ser instalada en dispositivos de versión inferior.

Procurar escoger valores pequeños para que la aplicación pueda instalarse en la mayoría de los dispositivos.

Esto tiene la desventaja de no poder incorporar características mas nuevas.



## Primer proyecto en Android.

Un proyecto en Android Studio puede contener varios módulos en un mismo proyecto. Cada módulo está formado por un descriptor de la aplicación (*Manifest*), el código fuente en java, una serie de ficheros en recursos (*Res*) y ficheros para construir el módulo *Gradle Scripts*.

## Android Manifest.

Este fichero describe la aplicación de Android

- Define Nombre, paquete, icono, estilos, etc. (*Visual*).
- Indica las actividades, interacciones, servicios y proveedores de contenido de la aplicación. (*Dinamismo*).
- Declara los permisos que requerirá la aplicación.

## JAVA y RES.

La carpeta java contiene el código fuente de la aplicación.

La carpeta RES contiene los recursos usados en la aplicación.

- **Drawable.** Ficheros de imágenes.
- **Mipmap.** Icono de la aplicación.
- **Layout.** Ficheros XML con vistas de la aplicación. Las vistas nos permitirán configurar las diferentes pantallas que compondrá la interfaz de usuario.
- **Values.** Ficheros XML para indicar valores usados en la aplicación.

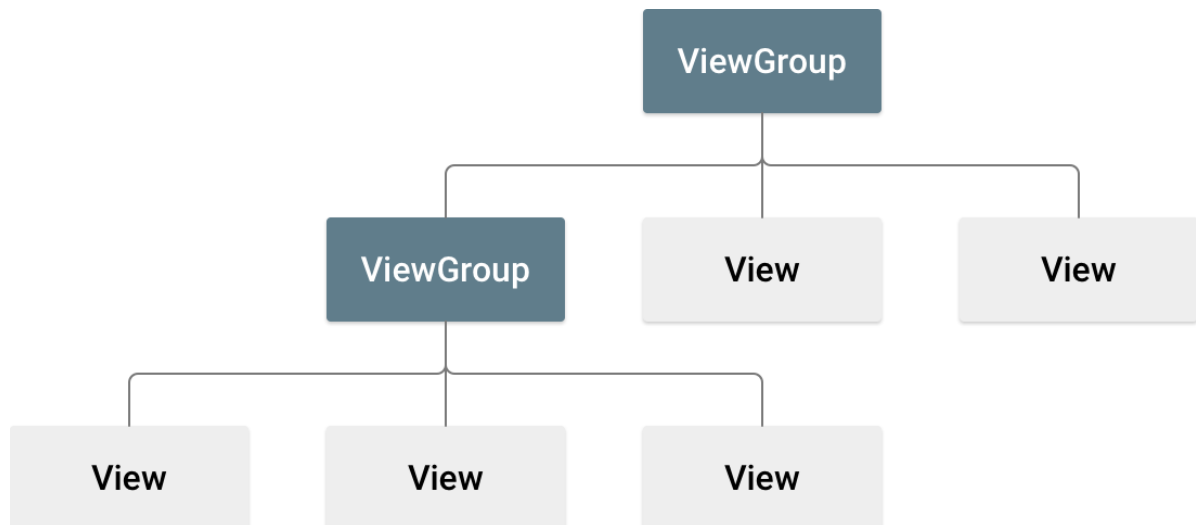
## Gradle Script.

Se almacena una serie de ficheros Gradle que permiten compilar y construir la aplicación.

El más importante es **build.gradle** que es donde se configuran las opciones de compilación del módulo: SDK Version, versionCode y versionName.

## Layouts.

Un diseño define la estructura en una interfaz de usuario en tu aplicación, por ejemplo, en una actividad. Todo los elementos del diseño se crean usando una jerarquía de objetos *view* y *viewGroup*. Una *view* suele mostrar un elemento que el usuario puede ver y con el que puede interactuar. En cambio, un *viewGroup* es un contenedor invisible que define la estructura de diseño de *View* y otros objetos *viewGroup*.



Los objetos *view* se denominan "widgets" y pueden ser un a de muchas subclases, como *Button* o *TextView*. Los objetos *viewGroup* se denominan "diseños" y pueden ser uno de los muchos tipos que proporcionan una estructura de diseño diferente, como *LinearLayout* o *ConstraintLayout*.

## Se pueden declarar layout de dos maneras.

- **Declarar elementos de la UI.** Android proporciona un vocabulario XML simple que coincide con las clases y subclases de vistas, como las que se usan en widgets o layouts.

Se puede usar la función editor de diseño de Android para crear un diseño XML mediante una interfaz de arrastrar y soltar.

- **Crear una instancia de elementos de diseño durante el tiempo de ejecución,** La aplicación puede crear objetos *View* y *viewGroup* (y manipular sus propiedades) de una forma programática.

Declarar la UI en XML permite separar la presentación de la app del código que controla su comportamiento. El uso de archivos XML también facilita la creación de distintos diseños para diferentes tamaños de pantalla y orientaciones.

## Escribe XML.

Se pueden crear rápidamente diseños de UI y de los elementos de pantalla que contienen, de la misma manera que creas paginas web en HTML, con una serie de elementos anidados.

Cada archivo de diseño debe contener un elemento raíz, que debe ser un objeto *View* o *ViewGroup*. Una vez que hayas definido el elemento raíz, se pueden agregar *widgets* u objetos de diseño adicionales como elementos secundarios para crear gradualmente una jerarquía de vistas que defina el diseño.

```
<?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:id="@+id/text"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
<Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

## Carga el recurso XML.

Cuando compilas la aplicación, cada archivo XML de diseño se compila en un recurso *View*. Debes cargar el recurso de diseño desde el código de la aplicación, en la implementación de devolución de la llamada *Activity.onCreate()*. Para eso, se llama a *setContentView()* pasando de la referencia a tu recurso de diseño en forma de *R.layout.layout\_file\_name*. Por ejemplo, si el diseño XML se guarda como *main\_layout.xml*, los cargás para la actividad de la siguiente manera:

```
fun onCreate(savedInstanceState: Bundle){
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_layout)
}
```

## Atributos.

Cada objeto *View* y *ViewGroup* admite su propia variedad de atributos XML. Algunos son específicos de un objeto *View*, estos atributos también son heredados por cualquier otro objeto *View* que pueda extender esta clase. Algunos son comunes para todos los objetos *View*, por que se heredan de la clase raíz *View*. Además otros atributos se consideran "parámetros de diseño", que son atributos que describen ciertas orientaciones de diseño de *View*, tal como lo define el objeto *ViewGroup* superior de ese objeto.

### ID.

Cualquier objeto *View* puede tener un ID entero asociado para identificarse de forma única dentro del árbol. Cuando se compila la aplicación, se hace referencia a este ID. Pero normalmente se asigna el ID en el archivo XML común para todos los objetos *View* y se utilizara muy a menudo.

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



Con el espacio de nombres de paquete android establecido, se hace referencia a un ID de la clase de recursos *android.R*, en lugar de la clase de recursos local.

Para crear vistas y hacer referencia a ellas desde la aplicación, puedes seguir el siguiente patrón común:

1. Definir una vista o un *widget* en el archivo de diseño y asignarle un ID único:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

2. Luego, crear una instancia del objeto *View* y capturarla desde el diseño ( generalmente en el método *onCreate()* ):

```
val myButton: Button = findViewById(R.id.my_button)
```

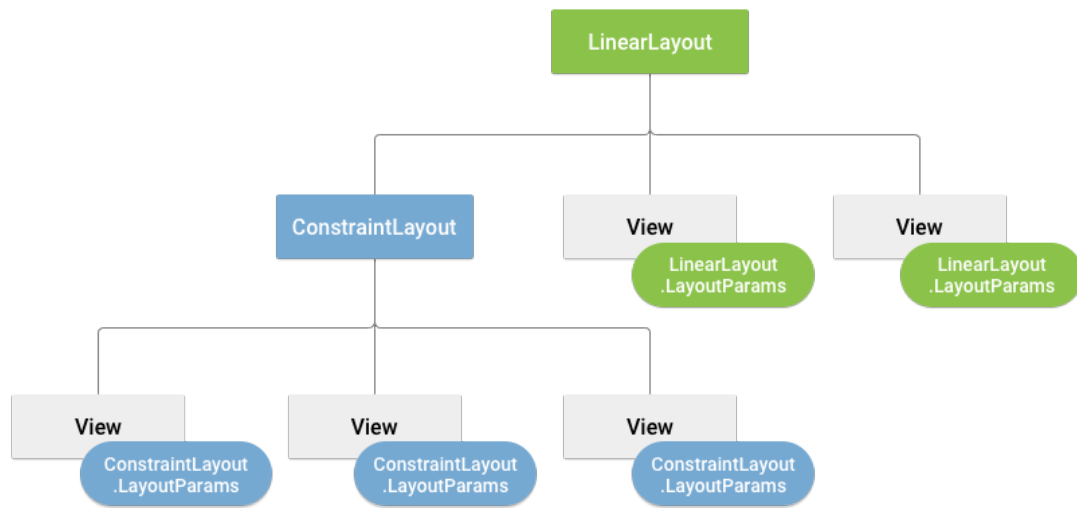
Definir ID para los objetos *View* es importante cuando se crea un *RelativeLayout*, las vistas del mismo nivel pueden definir su diseño en función de otra vista del mismo nivel.

## Parámetros de diseño.

Los atributos de diseño XML denominados *layout\_something* definen parámetros de diseño para el objeto *View* que son adecuados para el objeto *ViewGroup* en el que reside.

Cada clase *ViewGroup* implementa una clase anidada que extiende *ViewGroup.LayoutParams*. Esta subclase contiene tipos de propiedad que definen el tamaño y la posición de cada vista secundaria.





Cada subclase *LayoutParams* tiene su propia sintaxis para configurar valores, aun que también puede definir diferentes *LayoutParams* para sus propios elementos secundarios.

Todos los grupos de vistas incluyen un ancho y una altura (*LayoutWidth* y *LayoutHeight*), y cada vista debe definirlos. Muchos *LayoutsParams* también incluyen márgenes y bordes opcionales.

Generalmente se usa una de estas constantes para establecer el ancho y la altura:

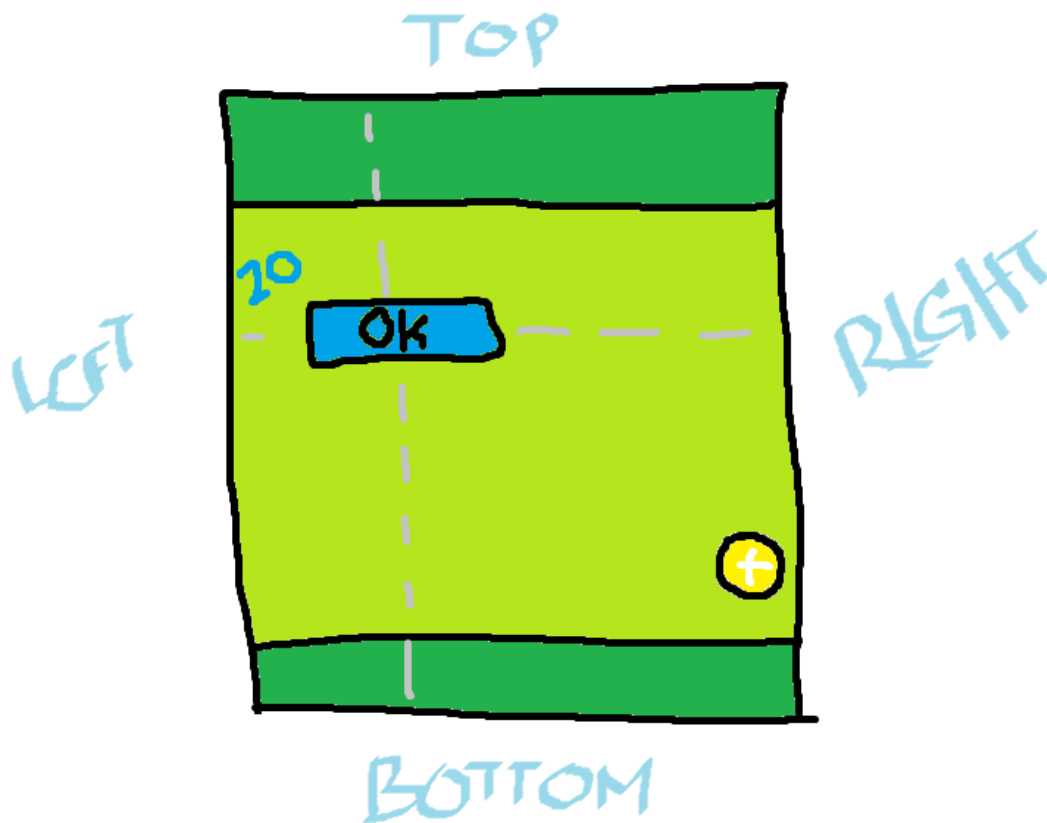
- **wrap\_content**. Le indica a tu vista que modifique su tamaño conforme a las dimensiones que requiere su contenido.
- **match\_parent**. Le indica a tu vista que se agrande tanto como lo permita su grupo de vistas superior.

El uso de medidas relativas, como unidades de píxeles independientes de la densidad (**dp**), **wrap\_content** o **match\_content**, es un mejor enfoque, ya que ayuda a garantizar que la app se visualice correctamente en distintos tamaños de pantalla de dispositivos.

## Posición del diseño

La geometría de una vista es la de un rectángulo. Una vista tiene una ubicación, expresada como un par de coordenadas. Ambos métodos muestran la ubicación de la vista respecto al elemento superior, cuando *getLeft()* muestra 20 significa que la vista se encuentra a 20 píxeles a la derecha del borde izquierdo de su elemento superior directo.

Se ofrecen varios métodos convenientes para evitar cálculos innecesarios: *getRight()* y *getBottom()*. Llamar a *getRight()* es similar al siguiente cálculo: *getLeft()* + *getWidth()*.



## Tamaño relleno y márgenes.

---

Una vista tiene dos valores de ancho y altura.

El primer par se conoce como *ancho medio* y *altura media*. Estas dimensiones definen cuán grande quiere ser una vista dentro de su elemento superior. Se pueden obtener llamando `getMeasureWidth()` y a `getMeasureHeight()`.

El segundo par se conoce como *ancho* y *altura*, o ,algunas veces, *ancho de dibujo* y *altura de dibujo*. Estas dimensiones definen el tamaño real de la vista en la pantalla en el momento de dibujarlas y después del diseño. Se pueden obtener llamando a `getWidth()` y `getHeight()*`.

Para medir estas dimensiones, una vista considera su relleno. El relleno se expresa en píxeles para las partes izquierda, superior, derecha e inferior. El relleno se puede usar para desplazar contenido de una vista determinada cantidad de píxeles. Un relleno izquierdo de 2 empuja el contenido de la vista 2 píxeles hacia la derecha del borde izquierdo. El relleno se puede ajustar usando el método `setPadding(int, int, int, int)` y se puede consultar llamando as `getPaddingLeft()`, `getPaddingTop()`, `getPaddingRight()` y `getPaddingBottom()`.

Una vista puede definir un relleno, no proporciona ningún tipo de compatibilidad para márgenes.

## Diseños comunes.

---

Cada subclase de la clase `ViewGroup` proporciona una manera única de mostrar las vistas que anidas en ella.

### Diseño lineal



Un diseño que organiza sus elementos secundarios en una sola fila horizontal o vertical. Si la longitud de la ventana supera la de la pantalla, crea una barra de desplazamiento.

### Objeto RelativeLayout



Te permite especificar la ubicación de los objetos secundarios en función de ellos mismos (el objeto secundario A a la izquierda del objeto secundario B) o en función del elemento primario (alineado con la parte superior del elemento primario).

### Vista web



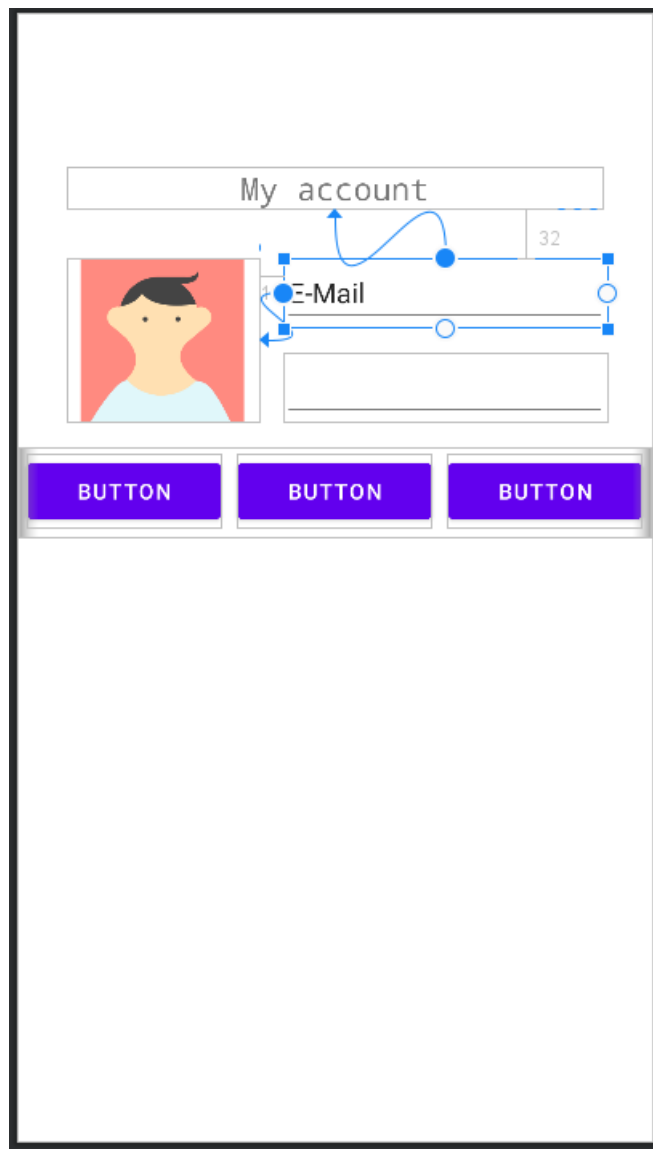
Muestra páginas web.

hay dos formas de visualizar una activity, portrait( parado) y landscape(acostado). Entonces que pasa si volteas el celular y no caben todos los views de portrait, se crea una variante de la activity, con create a variation activity.

en la carpeta res, hay otra carpeta que contiene los valores de colores, strings y en este caso tambien las dimensiones de la aplicación, son restricciones del tema para la aplicacion.

## Constraints.

Son restricciones hacia la forma visual de la aplicación Android, estas restricciones pueden ser de "strings", "dime" y "color", son archivos en xml. También hay restricciones para los activities.



## Strings.

```
<resources>
  <string name="app_name">practice</string>
</resources>
```

Después de agregan las restricciones de strings en el script xml correspondiente.

```
<resources>
  <string name="app_name">practice</string>
  <string name="Tittle_View">My account</string>
  <string name="email">E-Mail</string>
  <string name="password">Password</string>
  <string name="button_01">Button One</string>
  <string name="button_02">Button Two</string>
  <string name="button_03">Button Three</string>
</resources>
```

asi como se pueden crear variantes del activity, tambien se crean variante del idioma

string.xml + click der > Open translation editor

de esta manera se abre el editor de traducciones. Entonces se crearan dos archivos strings correspondiente a cada idioma

## Colors.

---

Así como se pueden cambiar las restricciones del idioma, de igual manera se cambian los colores que restringen al estilo de la aplicación, pero mas adelante se vera mas a fondo.

es importante cambiar los ids de los views para tener mayor control sobre ellos.

## Kotlin.

---

### ¿Por que Kotlin?.

---

- Eficiencia de la codificación.
- Se escribe menos código que otros entornos, tiene sintaxis CONCISA.
- Minimiza la necesidad de lidiar con muchas tareas comunes de limpieza, permitiéndote enfocar en escribir código que sea fundamental para la funcionalidad de la aplicación.

### Reducción de errores.

---

- Seguridad nula incorporada, requiere que seamos muy inencionados respecto a la anulabilidad de los valores y pone reglas para trabajar con datos nuleables.
- Expresar nuestras intenciones con un valor y cómo esperamos que se use ese valor. Esto a su vez permite al compilar identificar cuándo se usa el valor de una manera inconsistente.

#### Ejemplo

```
var a: string = "abc"
a = null // Error

var b:string? = "abc"
b = null // Correcto

//El signo de interrogación se especifica sun nuleabilidad.
```

## Compatibilidad.

---

- Excelente compatibilidad con java, permitiendo usar librerías escritas en ese lenfuaje.
- Excelente compatibilidad con Android, al compilar producen el mismo formato de archivo de distribución que las aplicaciones que las aplicaciones desarrolladas en java.

## Declarar variables.

---

En Kotlin se puede definir variables de dos formas:

- **Var** : Define variables que son mutables (puede cambiar su valor durante el tiempo de ejecución).
- **Val** : Define variables de solo lectura (su valor no cambia durante la ejecución). Marcara error si tratas de cambiar el valor al compilar.

## Tipos de datos.

---

## Enteros.

Tipo	Tamaño	Valor máximo
Byte	8 bit	128
Short	16 bit	32767
Int	32 bit	2,147,483,647(2 <sup>31</sup> -1)
Long	64 bit	9,223,372,036,854,775,807(2 <sup>63</sup> -1)

## Flotantes.

Tipo	Tamaño
Float	8 bit
Double	16 bit

## Boleanos.

Tipo	Descripción
Boolean	Verdadero o falso
Char	Un caracter
String	Secuencia de caracteres

## Declarar variables.

Kotlin no necesita terminar en punto y coma, son opcionales. Tampoco requiere especificar el tipo de dato por que se infiere en el contexto. Ejemplo:

```
var name = "Leo"; //Infiere el tipo string.  
  
name = 2; // Error por que name es de tipo string.  
  
var name:String = "Leo"; //En este ejemplo se especifica el tipo de dato.
```

## NULL.

Kotlin es estrictamente null safety forces, tenemos que indicar explícitamente que variable puede utilizar valores nulos.

Agrega un signo de interrogación (?) al tipo de dato, para indicar que la variable es nuleable.

## Concatenación de strings.

No se necesita usar el carácter "+" para unir strings, es fácil incorporar una variable o una expresión en un string ejemplo:

```
val name = "Leo";
println("Mi nombre es $name");
println("Mi nombre es: ${name.toUpperCase()}");
```

## Estructuras de control.

- **If:**

Es una expresión que podría retornar un valor, no existe un operador ternario por que **If** hace bien ese rol.

*Ejemplo.*

```
var max = a
if(a < b) max = b

//Otra manera
var max:Int
if( a > b){
    max = b
}else{
    max = b
}

//Otra manera
val max = if(a > b) a else b
//De esta manera if se usa como una expresion en lugar de una declaración
(Devolver o asignar valor a una variable), la expresión debe tener un else.
```

- **When :**

Se divide en ramas, evalúa cada rama hasta que una de estas cumpla la condición de la rama con su argumento. Se puede usar como *expresión* o *declaración*.

### **Expresión.**

El valor de la rama que cumpla se convierte en el valor de la expresión general.

### **Declaración.**

Los valores de las ramas individuales se ignoran. *(Al igual que el if, cada rama puede ser un bloque y su valor es el valor de la última expresión del bloque).*

*Ejemplo.*

```
//La rama else se evalúa si no cumple ninguna de las otras condiciones de la rama. Si when se usa como expresión, la rama else es obligatoria.

when(x){
    1->print("x == 1")
    2->print("x == 2")
    else->{
        print("x is neither 1 or 2")
    }
}
```

- **For :**

Itera en cualquier cosa que provee un iterador. Es equivalente al *foreach* de lenguajes como C#.

```
for(item in collection){
    print(item)
}

//Para iterar un rango de números use:
for(i in 1..3){
    print(i)
}
```

## Funciones.

La manera de hacer funciones en *Kotlin* es muy parecido a otros lenguajes, se especifica su accesibilidad, luego de la palabra *fun*, seguido de los parámetros de la función, al final se indica si esa función retornara un valor especificando el tipo.

```
private fun message(msg:String):String{
    return "This is a message for you: $msg"
}
```

## Prámetros en funciones

Los parámetros de una función se definen usando *Pascal notation (nombre:Tipo)*, están separadas por comas. Cada parámetro debe especificar su tipo de dato.

```
fun powerOf(number:Int, exponent:Int)
//Los parametros pueden ser inicializados.

fun powerOf(number:Int, exponent:Int = 0)
//de esta manera el parámetro exponent se vuelve opcional.
```

## Clases.

Las clases en *Kotlin* se declaran con la palabra clave *Class*. La declaración de la clase consta de:

- Nombre de la clase.
- Encabezado de la clase.
  - Parámetros.
  - Constructor principal.
- Cuerpo de la clase.

Podemos indicar los parámetros que son utilizados para definir un constructor por defecto. Si indicamos *var* o *val*, en estos parámetros pasarán a formar parte de las propiedades de la clase.

Una clase extiende de *Any*, que es similar a *Object* en *java*. Las clases en *Kotlin* por defecto son públicas.

*Ejemplo.*



```
class Persona(nombre:String, var edad:Int){
    val clavePersona = nombre.toUpperCase()
}

//Nombre de la clase "Persona".
//El encabezado consiste de los parámetros nombre y edad. Edad al
//tener la palabra var, es considerada miembro de la clase.
//El cuerpo de la clase se define por "{}".
```

## Constructores.

---

Una clase en *Kotlin* puede tener un constructor principal y uno o más secundarios. El constructor principal es parte del encabezado de la clase, después de el nombre y los parámetros.

El constructor principal no puede contener ningún código. El código de inicialización se puede colocar en bloques de inicialización, que tienen como prefijo la *init*.

## Constructores secundarios.

Tienen el prefijo del constructor. Si la clase tiene un constructor primario, cada constructor secundario necesita delegar el constructor primario. La delegación a otro constructor de la misma clase se realiza utilizando la palabra clave *this*.

## Creando instancia de objetos.

---

Para crear una instancia de una clase, llamamos al constructor como si fuera función regular. (*Kotlin no tiene New*).

## Miembros de la clase.

---

- Constructores y bloques inicializadores.
- Funciones.
- Propiedades.
- Clases anidadas e internas.
- Declaraciones de objeto.

## GETTER y SETTER.

No es necesarios definirlos. Las propiedades se leen y escriben indicando su nombre. Si requieres puedes escribirlos.

## Herencia.

---

Todas las clases tienen una super clase común *Any*, es la superclase predeterminada para una clase sin super tipos declarados.

*Any* tiene tres métodos:

- **equals()**.
- **hashCode()**.
- **toString()**.

*Por tanto se definen todas las clases de Kotlin.*

Las clases son finales, no se pueden heredar. No soporta herencia múltiple. Una clase además de extender de su padre puedan implementar varios interfaces.

# Colección.

---

Una colección generalmente contiene varios objetos del mismo tipo. Los objetos de una colección se denominan elementos.

## Tipos de colección:

- **listOf()**

Colección ordenada con acceso a elementos por índices, números enteros que reflejan su posición. Los elementos pueden aparecer más de una vez en una lista.

Permite crear una lista inmutable. Podremos tener todos los elementos de un mismo tipo o de diferentes tipos de datos.

**Formas de declarar una lista:**

- **emptyList()**

Permite crear una lista vacía.

- **listOfNull()**

Crea una lista inmutable con solo elementos no nulos.

- **arrayListOf()**

Crea una lista inmutable de tipo ArrayList.

- **Set()**

Colección de elementos únicos. Grupo de objetos sin repeticiones.

- **Map().**

Conjunto de pare clave-valor. Las claves son únicas y cada una de ellas se asigna exactamente a un valor. Los valores pueden estar duplicados. Son útiles para almacenar conexiones lógicas entre objetos.

Un par de interfaces representan cada tipo de colección:

**Read-only.** Proporciona operaciones para acceder a los elementos de la colección.

**Mutable.** Extiende la correspondiente interfaz de solo lectura con operaciones de escritura: agregar, eliminar y actualizar sus elementos.

*(Alterar una colección mutable no requiere que sea var: operaciones de escritura modifican el mismo objeto de la colección mutable, por lo que la referencia no cambia. Sin embargo, si intenta reasignar una val colección. obtendrá un error de compilación).*

```
val numbers = mutableListOf("One", "Two", "Three", "Four")
numbers.add("Five") //Correcto.
numbers = mutableListOf("Six", "Seven") //Error de compilación.
```

## Eventos de entrada.

---

En Android, existe más de una forma de interceptar los eventos desde una interacción del usuario con tu aplicación. Al considerar los eventos dentro de tu interfaz de usuario, el enfoque consiste en capturar los eventos desde el objeto vista específico con el que interactúa el usuario. La clase *View* proporciona los medios para hacerlo.

Dentro de las clases *View* es posible que notes varios métodos de devolución de llamada públicos que puedan ser útiles para eventos de IU. El *framework* de Android llama a estos métodos cuando la acción respectiva ocurre en ese objeto. Ejemplo, se toca una vista (como botón), se llama al método *onTouchEvent()* en ese objeto. Para interceptar esto, debes extender la clase y anular el método. Extender todos los objetos *View* para controlar ese tipo de evento no sería práctico. La clase *View* contiene una colección de interfaces anidadas con devoluciones de llamada que puedes definir más fácilmente. Estas interfaces, llamadas **objetos de escucha de eventos**, te permiten capturar la interacción del usuario con tu IU.

Si se desea extender la clase *View* para crear un componente más personalizado. Quizá extender la clase *Button* para crear algo más elaborado. Se podrán definir los comportamientos de eventos predeterminados para tu clase utilizando los **controladores de eventos de la clase**.

## Objetos de escucha de eventos.

Es una interfaz de la clase *View* que contiene un solo método de devolución de llamada. El *framework* de Android llamará a estos métodos cuando la vista con la que se haya registrado el objeto de escucha se active por la interacción del usuario con el elemento IU.

**Se incluyen los siguientes métodos de devolución de llamada.**

- **.onClick()**

Se puede implementar el *OnClickListener()* como parte de la actividad para evitar la carga extra de la clase y la asignación de objetos.

```
//Importamos la clase View
import android.view.View

//Añadimos el evento OnClickListener a la actividad
class MainActivity : AppCompatActivity(), View.OnClickListener {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

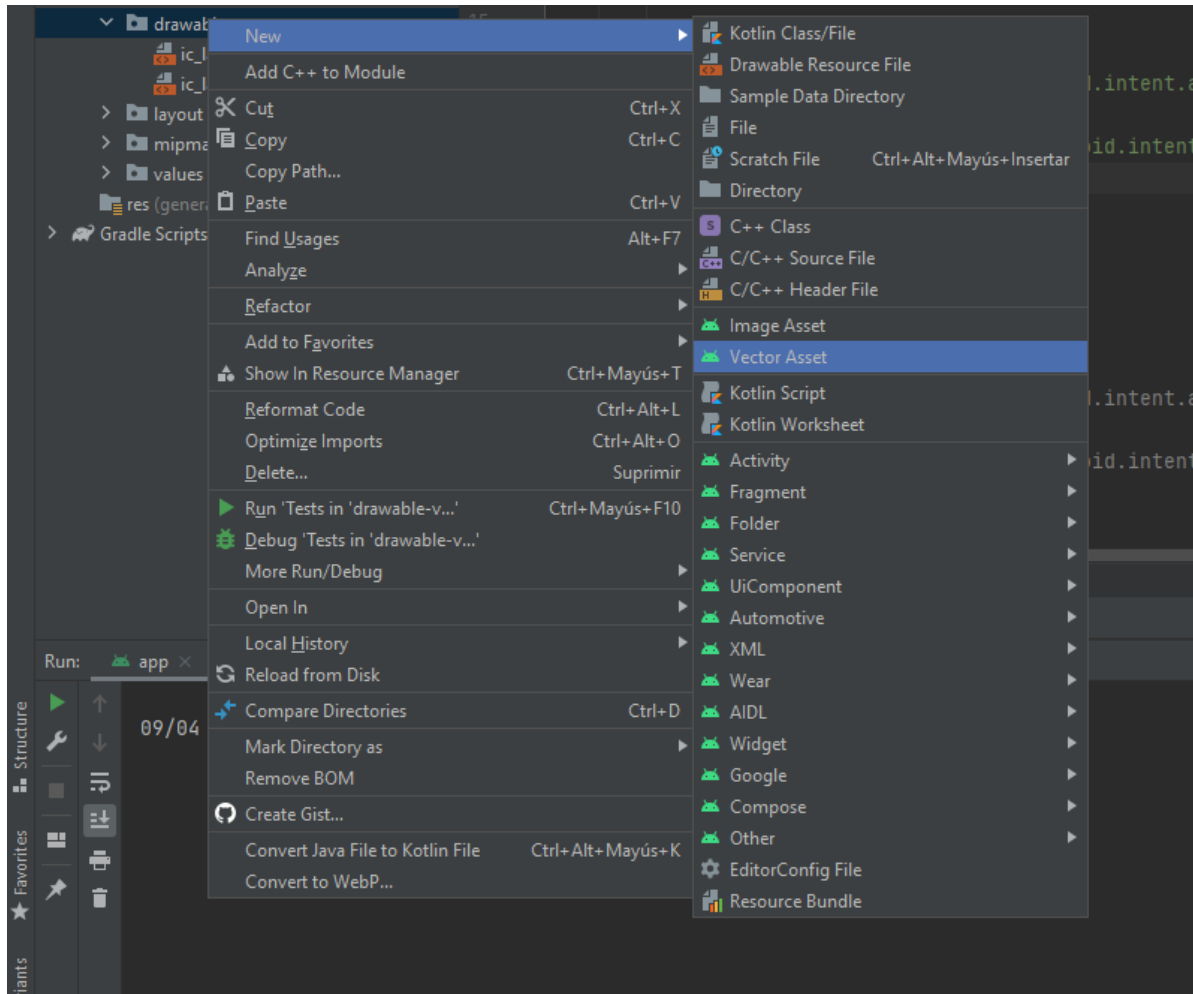
        //Añadimos el metodo listener de nuestro boton
        //Y especificamos en que contexto va a estar escuchando
        //En este caso sera el mismo main activity

        this.okButton.setOnClickListener(this) //el contexto es This
    }

    override fun onClick(v: View?) {
        val name:String = this.txt_name.text.toString()
        val age:String = this.txt_age.text.toString()
        val person:String = "My name is $name, and i $age years old"
        this.person.setText(person)
    }
}
```

## Vector resources.

Para utilizar los iconos que tiene predefinidos Android studio, iremos a la carpeta *drawable* (Vista Android).



## Vectores.

Android studio incluye *asset studio*, una herramienta que permite agregar iconos de material e importar archivos de gráficos vectoriales escalables (SVG) y de *adobe photoshop document* (PSD) a tu proyecto como recursos de elementos de diseño vectoriales.

El uso de elementos de diseño vectoriales en lugar de mapas de bits reduce el tamaño de tu APK por que se puede cambiar el tamaño del mismo archivo según las diferentes densidades de pantalla sin perder la calidad de la imagen.

## Styles.

Es una forma de especificar las características que queremos aplicar a una vista sin tener que estableces los atributos en cada una de las vistas.

puedes aplicar un estilo a una o mas vistas, la vista toma los valores del atributo.

cuando el estilo contiene atributos que no son soportados por la vista, son ignorados.

### Como definir un estilo

```
<resources>
  <style name="NoteTitle"
parent="@android:style/TextAppearance.Material.Small">
    <item name="android:textColor">@color/newColor</item>
  </style>
</resources>
```

Con el atributo "parent" defino que voy a heredar un estilo y el valor del atributo es de quien voy a heredar.

## Herencia en estilos.

- Estilos soportan herencia de otros estilos o *frameworks*.
- Reciben todos los atributos de ese estilo principal, además puede agregar atributos adicionales o sobre escribir.

## Temas.

En Android se maneja todo el estilo por temas que rigen el aspecto visual de la aplicación, tiene solo recursos XML, uno para modo light y otro para el modo *night*, si se crea otro tema, se debe dar de alta en Android *manifest* dentro de la etiqueta aplicación, si se da de alta en una etiqueta *activity*, el tema solo se aplicará a ese *activity*.

## Como aplicar un estilo como un tema.

Se pueden crear los temas de la misma manera en que se crean los estilos, la diferencia esta en como se aplican: en vez de aplicar un estilo con el atributo *Style* en una vista, aplica un tema con el atributo *android:theme* en la etiqueta *application* o en una etiqueta *activity*, en *AndroidManifest*.

```
<manifest ... >
  <application android:theme="@style/Theme.AppCompat" ... >
  </application>
</manifest>
```

Para poder aplicar el tema a una sola actividad, en este caso el tema claro:

```
<manifest ... >
  <application ... >
    <activity android:theme="@style/Theme.AppCompat.Light" ... >
    </activity>
  </application>
</manifest>
```

Los ejemplos anteriores muestran cómo aplicar un tema como *Theme.AppCompat* que proporciona la biblioteca de compatibilidad de Android. La mejor manera de personalizar el tema es extender estos estilos de la biblioteca de compatibilidad y anular algunos de los atributos.

## Estilos vs Temas.

Los temas y los estilos tienen muchas similitudes, pero tienen diferentes propósitos. Ambos tienen la misma estructura *clave-valor* que asigna atributos a recursos.

Un estilo especifica como se vera un tipo de vista determinado. Un estilo puede especificar los atributos de un botón. Cada atributo que especificas en un estilo es un atributo que puedes configurar en al archivo de diseño. Resulta más fácil de usar y mantener en varios *widgets*.

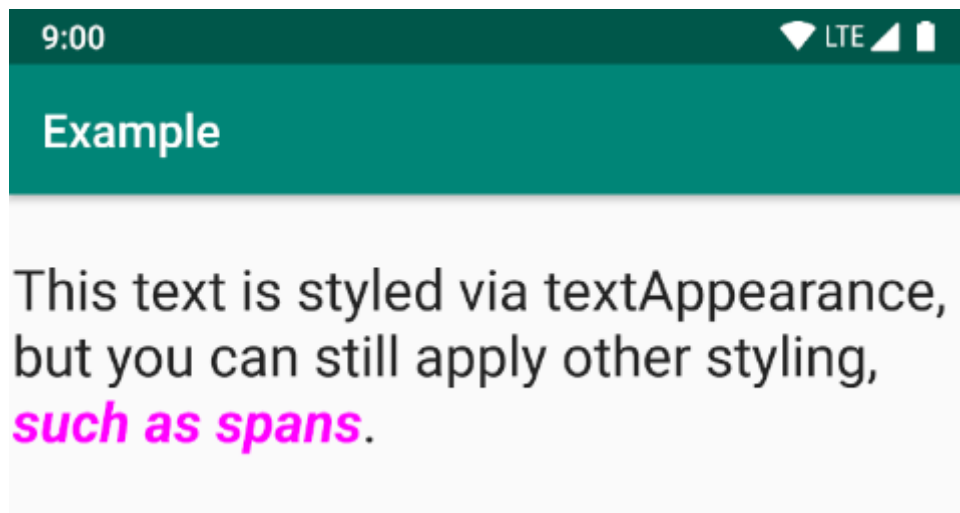
Los estilos y los temas se diseñaron para trabajar en conjunto. Puedes tener un estilo que una parte del botón debe ser *ColorPrimary* y otra deber ser *ColorSecondary*. Las definiciones reales de los colores se proporcionan en el tema. Cuando el dispositivo cambia al modo nocturno de esta manera cambiar los valores de todos esos nombres de recursos. No es necesario cambiar los estilos, estos usan los nombres semanticos y no las definiciones de color especificas.

## Jerarquía de estilos.

---

Debes usar temas y estilos para mantener la coherencia. Si especificaste los mismos atributos en varios lugares, la siguiente lista determina qué atributos se aplican finalmente. La lista esta ordenada de mayor a menor prioridad.

1. Aplicación de estilo a nivel caracter o parrafo a través de intervalos de texto a clases derivdas de *TextView*.
2. Aplicación de atributos de forma programatica.
3. Aplicación de atributos individuales directamente a una vista.
4. Aplicación de un estilo de vista.
5. Estilo predeterminado.
6. Aplicación de un tema a una colección de vistas, una actividad o toda la app.
7. Aplicación de un estilo determinado específico de la vista, como la configuración de una *TextAppearance* en una *TextView*.



## TextAppearance.

---

Una limitación es que puedes aplicar un solo estilo a un *View*. Sin embargo, en un *TextView*, también puedes especificar un atributo *TextAppearance* que funciona de una amañera similar.

```
<TextView
    android:textAppearance="@android:style/TextAppearance.Material.Headline"
    android:text="this text is styled via textAppearance!" />
```

*TextAppearance* te permite definir un estilo específico del texto y dejar el estilo de una `view` disponible para otros usos. Sin embargo, ten en cuenta que, si defines cualquier atributo de texto directamente en `view` o en un estilo, esos valores anularán los de `TextAppearance`.

# Ciclo de vida de una *Activity*.

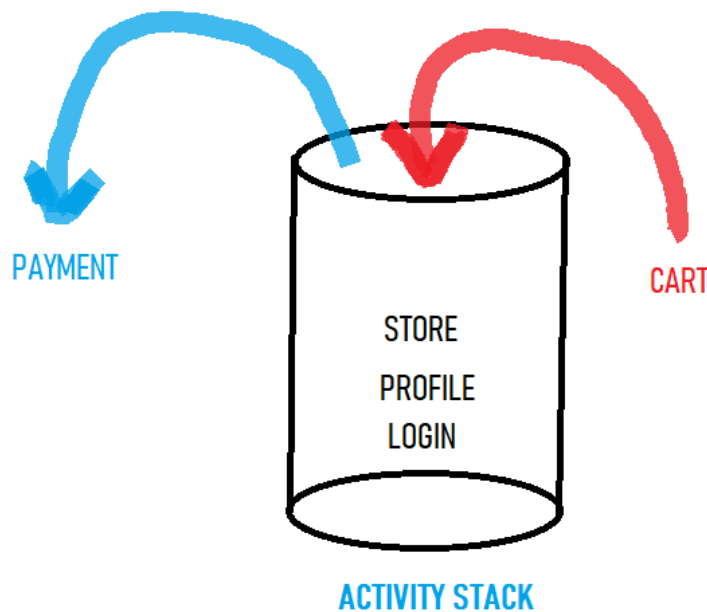
---

Una aplicación en Android esta formada por un conjunto de elementos básicos de interacción con el usuario conocidos como *Activity*.

Son las actividades las que controla el ciclo de vida, dado que el susuario no cambia de aplicación, si no de *Activity*.

EL sistema mantiene la pila con las *Activity* previamente visualizadas, de forma que el usuario pueda regresar a la actividad anterior pulsando "retorno".

La aplicación navega através de las *Activity*, no cambia de entorno, todo se almacena en la pila de *Activity*, de esta manera se guarda el estado en que se quedo esa *Activity*, pero esto no quiere decir que vivirá para siempre, el sistema puede decidir si será destruida o no, y se puede jugar mucho con el ciclo de vida de las *Activity*



Una aplicación de Android corre dentro de su propio proceso Linux. En este proceso se crea con la aplicación y continuará vivo hasta que ya no sea requerido y el sistema reclame su memoria para asignársela a otra aplicación.

La destrucción de un proceso no es contralada directamente por la aplicación, sino por el sistema (Memoria disponible, importancia de las parte de la app).

Tras eliminar el proceso de la aplicación, el usuario vuelve a ella, se crea de nuevo el proceso, pero se habrá perdido el estado que tenía esa aplicación.

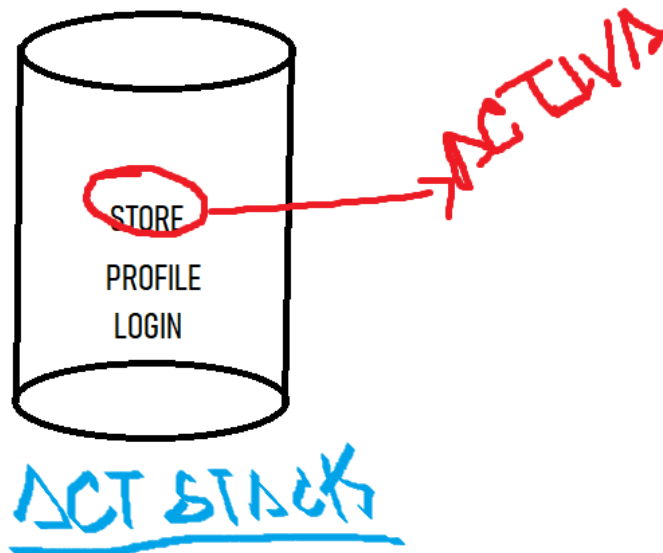
Sera responsabilidad del programador almacenar el estado de las actividades, si queremos que cuando sea reiniciadas conserve su estado.

## 4 Estados de una actividad.

---

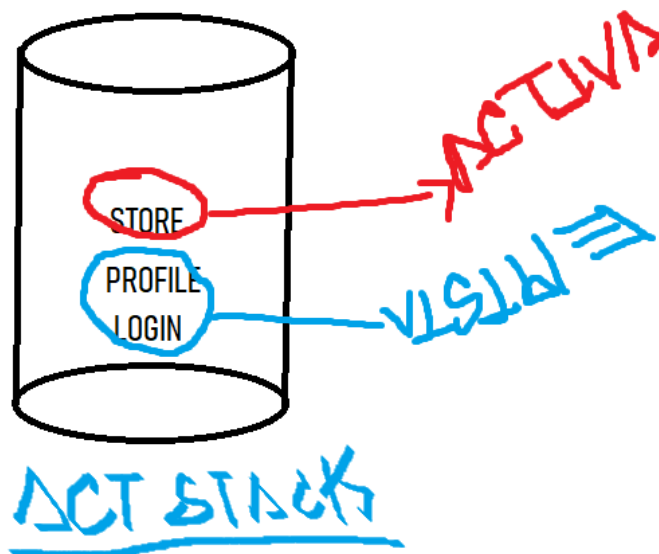
- **Activa (Running).**

La actividad está encima de la pila, lo que quiere decir que es visible y tiene foco.



- **Visible (Paused).**

La actividad es visible pero no tiene el foco. Se alcanza este estado cuando otra actividad para activa, lo cual esta transparente o no ocupa toda la pantalla.



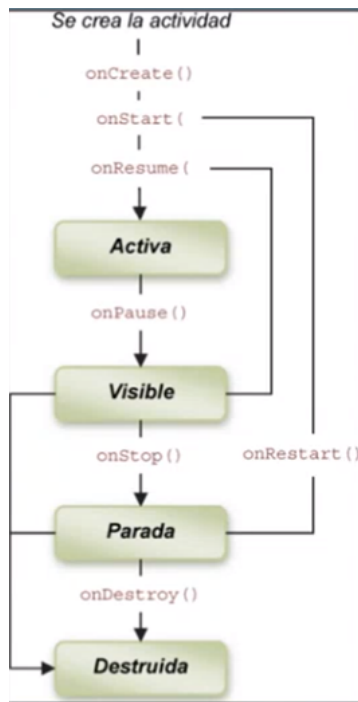
- **Parada (Stopped).**

Cuando la *Activity* no es visible. El programador debe guardar el estado de la interfaz de un usuario, las preferencias, etc.

- **Destruida (Destroyed).**

Cuando la *Activity* termina, al invocarse el método *finish()* matada por el sistema.





Como se observa entre cada estado de las *Activity* se ejecutan funciones. Cada vez que una *Activity* cambia de estado se generan eventos que podrán ser capturados por ciertos métodos de la *Activity*.

## onCreate(Bundle).

Se llama en la creación de la actividad. Se utiliza para realizar todo tipo de inicializaciones, como la creación de la interfaz de usuario o la inicialización de estructuras de datos.

```
override fun onCreate(savedInstanceState: Bundle?) {}
```

### saveInstanceState.

Se puede guardar o almacenar información antes de que se destruya la *Activity*, se utiliza para almacenar y controlar estado de la *Activity*.

## onResume().

Se llama cuando la actividad va a comenzar a interactuar con el usuario. Es un buen lugar para lanzar las animaciones y la música.

## onPause().

Indica que la *Activity* está a punto de ser lanzada a segundo plano, normalmente por que se lanza otra *Activity*. Este lugar adecuado para detener animaciones, música o almacenar los datos que estaban en la edición.

## onStop().

La actividad ya no será visible para el usuario, si hay muy poca memoria es posible que la *Activity* se destruya sin llamar a este método.

## onRestart().

Indica que la *Activity* volverá a ser representada después de haber pasado por **onStop()**.

## onDestroy().

Se llama antes de que la actividad sea totalmente destruida. Ejemplo: cuando el usuario pulsa el botón de volver o cuando se llama al método **finish()**.

## Intents.

Cuando se quiere llamar una *Activity* desde otra, a esto se le llama *intent* con la función **Intent(packageContext:, ShowActivity)**. Pero el intent se puede usar para interactuar con el hardware del dispositivo también como abrir la cámara.

- **PackageContext.** Es el contexto donde actualmente te encuentras, en el ejemplo se usa el *this*.
- **ShowActivity.** Que *Activity* quieres mostrar, en este ejemplo tenemos un *Activity* llamado *ShowInfoActivity* y se especifica que tipo es (class.java).

### Ejemplo.

```
val intent = Intent(packageContext: this, ShowInfoActivity::class.java)
startActivity(intent)
```

La variable *intent* contiene la información de lo que se quiere hacer, pero sin embargo no se activa, se usa la función *startActivity()* para poder comenzar el *Activity*.

```
startActivity(intent)
```

Mediante los *intent* se puede mandar información de un *Activity* a otro mediante un método de *intent* llamado *putExtra()*. Al *putExtra* se le define como clave valor.

```
val intent = Intent(packageContext: this, ShowInfoActivity::class.java)
intent.putExtra(name: "edad", value: 22)
intent.putExtra(name: "nombre", value: "Miguel")
startActivity(intent)
```

De esta manera se envía la información a otro *Activity* pero ¿Cómo la podemos recibir?. Debemos tener acceso al *intent* que recibe todos los valores extra con *intent.extras*, este objeto ya tiene la información almacenada, se puede ver como si fuera un objeto JSON.

```
val extras = intent.extras
val nombre = extras?.getString(key: "nombre")?: "No tiene dato"
val edad = extras?.getInt(key: "edad")?: "No tiene dato"
print("Mi nombre es $nombre y tengo $edad")

// En este ejemplo se usa el if ternario (?:) en kotlin es llamado operador elvis.
```

## Ejemplo de un intent para visitar una pagina web.

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse(uriString:
"https://www.pluralsight.com"))
startIntent(intent)
```

## Activity de inicio.

R es una clase que se genera que contiene todos los recursos que contiene la carpeta res.

para que la activity sea parte de nuestra aplicación cuando se vaya a probar en el dispositivo o emular, se debe dar de alta en el manifest del proyecto.

```
<activity android:name=".activity_main">

</activity>
```

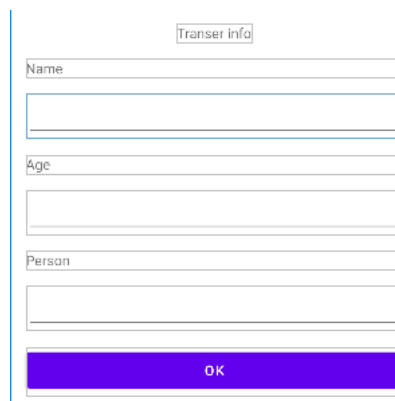
para poder hacer especificar que activity si llamara como main se hace con lo siguiente

```
<activity android:name=".login_activity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

## Agregar Activity.

Tenemos el siguiente *layout* para poder ingresar información, al final con el botón "OK" enviaremos la información al otro *layout*.



1. Crearemos una función en el *ActivityMain* *sendMessage()* que será llamada cuando el botón se pulsa para capturar la información de los *inputs*.

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

    //Creamos la funcion message para capturar la información del
```

```
//los inputs

public fun message(v: View?){
    //Aquí se capturan los inputs de los elementos view
    //recuerda importat las librerías necesarias para
    //el correcto funcionamiento de View
}

}
```

2. En el archivo del *layout* debemos ubicar el atributo *onClick* y seleccionar la función que acabamos de crear, de esta manera, cuando se presione el botón, el sistema llamará al método *message()*.

La función *message()* debe cumplir con los siguientes requisitos para que sea compatible con el atributo *onClick()*.

- Debe ser pública.
  - Debe ser void (Unit).
  - Recibe un objeto *View* como único atributo.
3. Completar el método para capturar la información.

```
import android.view.View

class MainActivity : AppCompatActivity(){
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    public fun message(v: View){
        val name = this.txt_name.text.toString()
        val age = this.txt_age.text.toString()
        val person = "My name is $name, and i got $age years old"
    }
}
```

## Cómo compilar un intent.

Como ya sabemos, un *Intent* es un objeto que proporciona vinculación en tiempo de ejecución entre componentes separados, como dos actividades. Representa una intención que tiene una app de realizar una tarea.

En *MainActivity*, agrega la constante **NAME** y el **intent** al código del *message()*.

```
import android.view.View
import android.content.Intent

//Se debe agregar la librería Intent

const val NAME = "Name"

class MainActivity : AppCompatActivity(){
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
```

```

        setContentView(R.layout.activity_main)
    }

    public fun message(v: View){
        val name = this.txt_name.text.toString()
        val age = this.txt_age.text.toString()
        val person = "My name is $name, and i got $age years old"
        val intent = Intent(this, Login::class.java).apply{
            putExtra(NAME, person)
        }
        startActivity(intent)
    }
}

```

La clase *Login* aun no es creada.

- El constructor *Intent* toma dos parámetros, un *Context* y una *Class*.
- El parámetro *Context* se usa primero por que la clase *Activity* es una subclase de *Context*.

El parámetro *Class* del componente de la app, al que el sistema entrega el *Intent*, es en este caso, la actividad que va a comenzar.

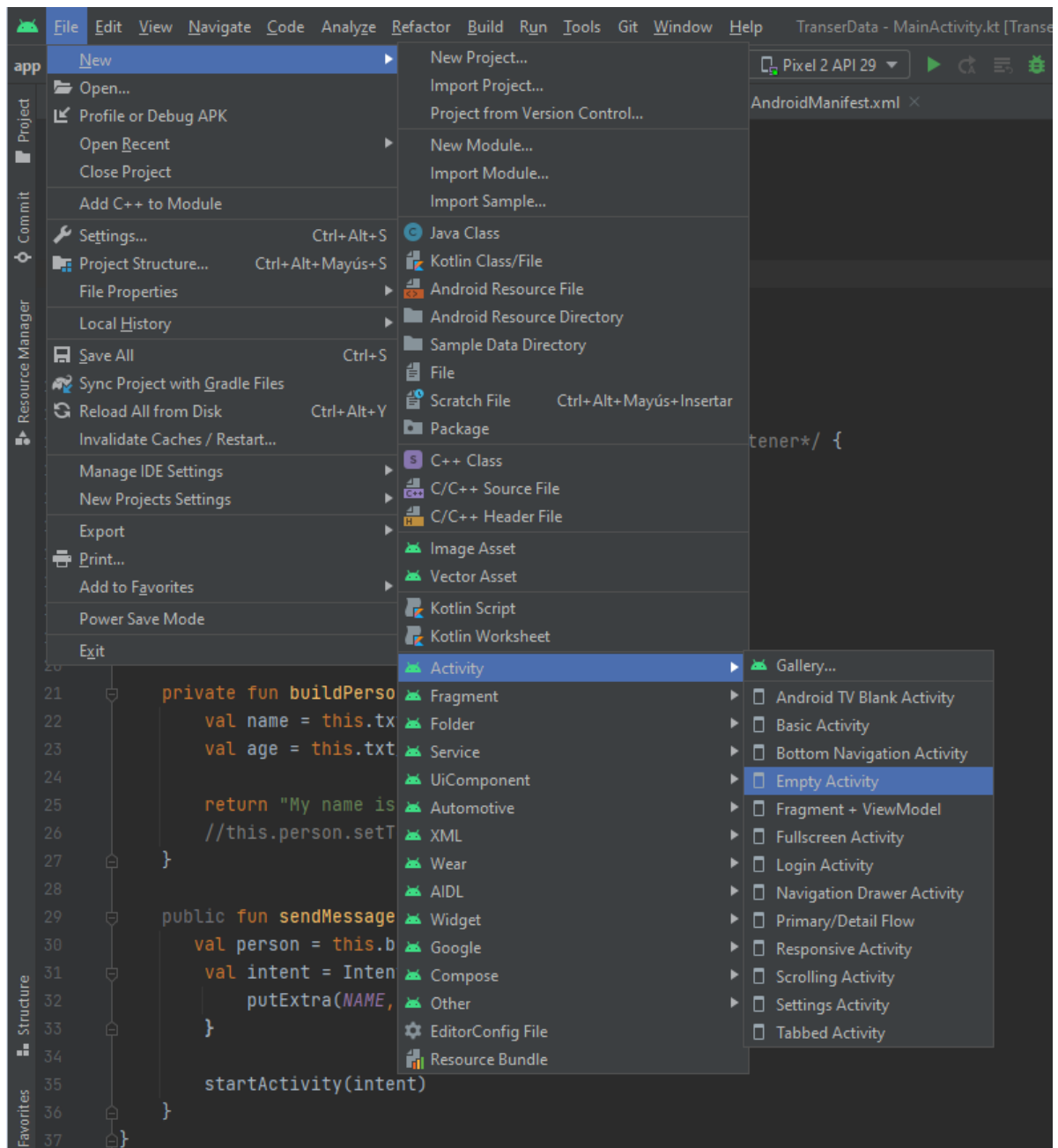
- El método *putExtra()* agrega el valor de *person* al intent. Un *Intent* puede transportar tipos de datos como pares clave-valor llamados *extras*.

Tu clave es una constante pública *NAME* por que la actividad siguiente usa la clase para obtener el valor del texto.

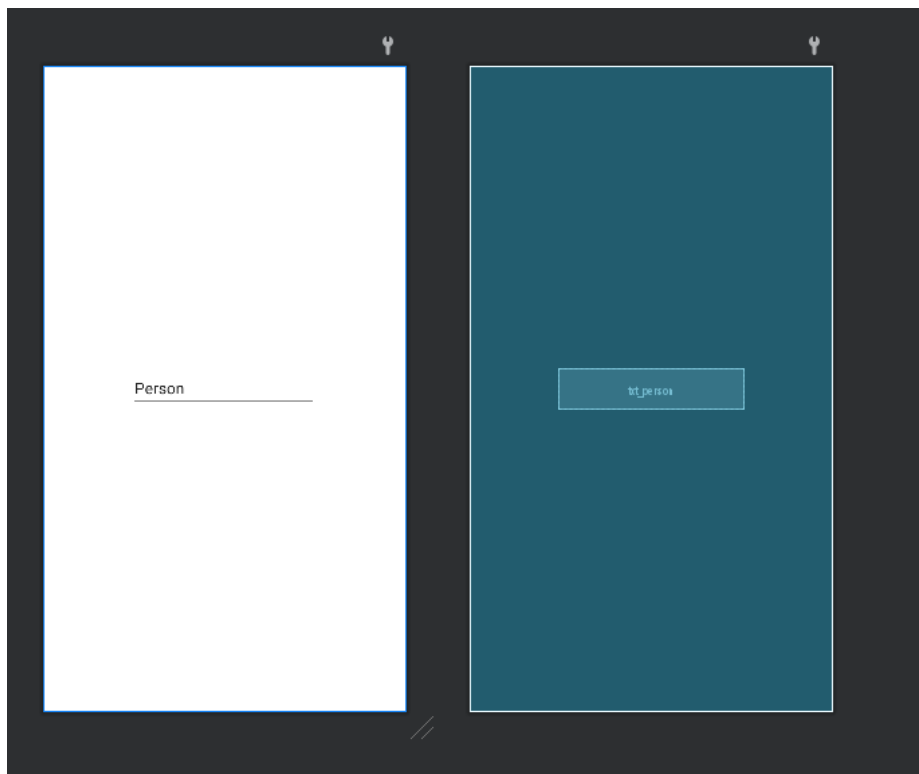
- El método *startActivity()* inicia una instancia de *Login* que especifica el *Intent*

## Cómo crear la segunda actividad.

---



Agregamos un *plain text* para mostrar el texto enviado



## Cómo mostrar el mensaje.

1. En *Login* agrega el siguiente código a *OnCreate()*.

```
import android.content.Intent //Importante

class Login : AppCompatActivity(){
    override fun onCreate(savedInstanceState: Bundle?){
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_login)

        val person = intent.getStringExtra(NAME)

        this.txt_person.setText(person) //Es la referencia al plain text
    }
}
```

## Cómo agregar navegación acendente.

En cada pantalla de tu app que no sea el punto de entrada principal, es decir, todas las pantallas que no sean la "pantalla principal", se debe proporcionar navegación para que el usuario pueda regresar a la pantalla superior lógica en la jerarquía de la app.

Para agregar el botón de arriba, debes declarar qué *Activity* es la superior lógica en el archivo *AndroidManifest.xml*.

```
<activity
    android:name=".Login"
    android:parentActivityName=".MainActivity">
</activity>
```

## Recycler View.

Facilita que se muestren de manera eficiente grandes conjuntos de datos. La biblioteca *RecyclerView* creará los elementos de forma dinámica cuando se los necesite.

Recicla esos elementos individuales. Cuando se desplaza fuera de la pantalla, *RecyclerView* no destruye su vista. En cambio, reutiliza la vista para los elementos nuevos que se desplazaron y ahora se muestran en pantalla.

## Beneficios de usar *RecyclerView*.

---

- Extremadamente flexible.
- La visualización de la lista se divide en distintas fases (*cada fase ofrece la posibilidad de personalizar*).
- Proporciona una gestión de pantalla eficiente (*Separa los detalles de los datos de la vista*).

## ¿Cómo funciona?

---

*LayoutManager* su responsabilidad es organizar el contenido dentro de una *RecyclerView*.

*Adapter* que es responsable de enviar el contenido al *RecyclerView*. Administra los datos y los *View* para mostrar esos datos.

## Instance State.

---

Las actividades tienen *Instance state*. Los valores almacenados en propiedades de la clase que se pierde cuando la *Activity* es destruida, lo que dificulta volver a mostrar el estado que tenía previamente el usuario.

Las *Activities* son destruidas. Cuando el sistema está bajo la presión de recursos o cuando lleva un largo periodo en background.

También perdemos el estado cuando la aplicación cambia de orientación. Esto provoca que la vista sea destruida y recreada.

## Guardar y establecer Instance State.

---

Es importante manejar efectivamente la instancia de nuestra *Activity*. Para lo cual usaremos el método: *onSaveInstanceState* nos da la oportunidad de guardar la instancia de nuestra actividad.

Guardar el estado en *Bundle*, todo lo que queremos mantener tiene que estar en *Bundle*. Cuando la vista es recreada en el método *onCreate*, recibe el *Bundle* con todos los valores que fueron guardados previamente.

El *Bundle* que recibe puede ser nulo, por que ante de usar los datos verificar que no sea nulo.

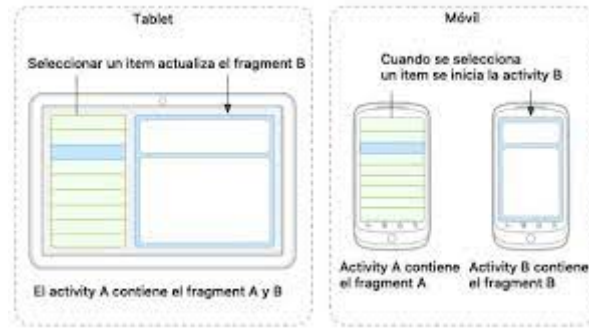
*(Un Android App Bundle es un formato de publicación que incluye todos los recursos y el código compilado de tu app, pero delegada la generación del APK y la firma de Google Play).*

## Fragments.

---

Se originan desde la versión 3 de Android, el objetivo era aprovechar la superficie de una Tablet. al tener mayor superficie debías aprovechar al máximo todo el espacio. pero en el móvil al no tener la misma superficie, la vista se fragmenta en varias partes. Permite hacer cosas modulares y permite mayor flexibilidad.





Representa un comportamiento o una parte de la interfaz de usuario en una *FragmentActivity*. Se pueden combinar varios fragmentos para crear una IU multi panel y volver a usar el fragmento en diferentes actividades.

- Tiene un ciclo de vida propio.
- Recibe sus propios eventos de entrada.
- puedes agregar o quitar mientras la actividad se esté ejecutando ( algo así como una "subactividad").
- Puedes volver a usar en diferentes actividades.

Un fragmento siempre debe estar alojado en una actividad y el ciclo de vida del fragmento se ve afectado directamente por el ciclo de vida de la actividad anfitriona.

Por ejemplo, cuando la actividad está pausada. también lo están todos sus fragmentos, y cuando la actividad se destruye, lo mismo ocurre con todos los fragmentos.

Sin embargo, mientras una actividad se está ejecutando, puedes manipular cada fragmento de una forma independiente, por ejemplo. para agregarlo o quitarlo.

Cuando se agrega un fragmento como parte de la presentación de la actividad, se encuentra en *ViewGroup* dentro de la jerarquía de vistas de la actividad y del fragmento define su propio diseño de vistas.

## Fragmentos y su ciclo de vida.

Para crear un fragmento, debes crear una subclase *Fragment* (o una subclase existente de ella). La clase *Fragment* tiene un código que se semeja bastante a una *Activity*.

Contiene métodos de devolución de llamada similares a las de una actividad. como *\*onCreate()*, *onStart()*, *onPause()* y *onStop()*.

Generalmente, debes implementar al menos los siguientes métodos del ciclo de vida:

- **onCreate()**

El sistema lo llama cuando crea el fragmento. En tu implementación, debes inicializar componente esenciales del fragmento que quieras conservar cuando el fragmento se pause o se detenga, y luego reanude.

- **onCreateView()**

El sistema llama cuando el fragmento debe diseñar su interfaz de usuario por primera vez. A fin de diseñar una IU para tu fragmento, debes mostrar un *View* desde este método, que será la raíz del diseño de tu fragmento. Puedes mostrar un valor nulo si el fragmento no proporciona una IU.

- **onPause()**

El sistema llama a este método como el primer indicador de que el usuario está abandonando el fragmento (aun que no siempre significa que el fragmento esté destruyendo). Generalmente, este es el momento en el debes confirmar los cambios que deban conservarse más allá de la sesión de usuario actual (por que es posible que el usuario no vuelva).

- **onAttach()**

Reciba una llamada cuando se asocia el fragmento con la actividad (aquí se pasa el *Activity*).

- **onDetach()**

Se llama cuando se desasocia el fragmento de la actividad.

## Tablayout.

---

Las *Tabs* nos van a permitir crear una interfaz de usuario basada en pestañas, donde, una forma muy intuitiva, podemos ofrecer al usuario diferentes contenidos, que son seleccionados al pulsar una pestaña.

para esto utilizaremos:

- ViewPager2.
- FragmentStateAdapter.
- TabLayoutMediator.

## View Pager 2.

---

Es una vista que nos permite mostrar una colección de **Fragments** o **Views** al usuario, en un formato que se pueda deslizar.

Utilizando especialmente en la visualización de contenido.

Para poder usarlo se requiere agregar la dependencia al *build.gradle*:

```
dependencies{
    implementation"androidx.viewpager2:viewpager2:1.0.0"
}
```

A diferencia de su predecesor este soporta diseño de derecha a izquierda, orientación vertical, colecciones de fragmentos modificables.

*ViewPager2* usa el componente *RecyclerView* para manejar la visualización del contenido que le asigna.

Usa la clase *LayoutManager* para el desarrollador tenga la capacidad de establecer la orientación del componente.

Usando el método *setOrientation* se puede establecer la orientación.

Podemos registrar una devolución de llamada de cambio de página usando *OnPageChangeCallback*, podemos escuchar:

- **onPageCrolled()**. Se activa cuando se produce un evento de desplazamiento usando la pagina actual.
- **onPageSelected()**. Se activa cuando se selecciona una nueva página.
- **onPageScrollStateChange()**. Se activa cuando cambia el estado de desplazamiento.

## Adapter.

---

Es un objeto que actúa como puente entre la vista y los datos. El adaptador proporciona acceso a los elementos de datos y es responsable de crear una vista por cada elemento. *(El objetivo es el puente entre las vistas y los datos).*

Para *TabLayout* usaremos el:

- **FragmentStateAdapter** del cual usaremos:
  - `getItemCount`.
  - `createFragment`.

## Companion object.

---

Es un *Singleton*, y se puede acceder a sus miembros directamente a través del nombre de la clase que lo contiene (aunque también puede instertar el nombre del objeto complementario si desea ser explícito sobre el acceso al objeto complementario).

En nuestro caso lo usaremos para definir una constante en la clase.

## Table layout mediator.

---

Un mediador para vincular un *TabLayout* con un *ViewPager2*. El mediador sincronizará la posición de *ViewPager2* con la pestaña cuando seleccione una pestaña, y la posición de desplazamiento de *TabLayout* cuando el usuario arrastre *ViewPager2*.

Esto lo hace mediante los siguientes:

- Escuchará *OnPageChangeCallback* de *ViewPager2* para ajustar la pestaña cuando *ViewPager2* se mueva.
- Escucha *OnTabSelectedListener* de *TabLayout* para ajustar *ViewPager2* cuando se mueve la pestaña.
- Escucha al *AdapterDataObserver* de *RecyclerView* para recrear el contenido de la pestaña cuando cambia el conjunto de datos.

por medio de *TabLayoutMediator.TabConfigurationStrategy\** en la que establece el texto de la pestaña y / o realizar cualquier estilo de las pestañas que necesite.