Interfaces de Usuario

Diseño de Interfaces Gráficas en Android

Las interfaces gráficas en Android se construyen usando 2 métodos:

- **Declarativo**: consiste en usar XML para declarar qué mostrará la UI, en forma similar a diseñar páginas web usando HTML. Se escriben *tags* que especifican elementos que aparecerán en la pantalla.

- **Programático**: consiste en escribir código Java o Kotlin para desarrollar la UI. Es similar a escribir una interface gráfica en AWT ó Swing en una aplicación JSE.

  Todo lo que se hace declarativamente se puede hacer programáticamente. Sin embargo en Java o Kotlin es posible especificar qué pasará cuando el usuario interactúa con una componente.

Diseño de Interfaces Gráficas en Android

Ventajas del enfoque declarativo:

- Uso de herramientas que facilitan la creación de UI. Android Studio viene equipado con herramientas visuales.
- XML es un lenguaje legible que puede ser entendido por personas no familiarizadas con la plataforma y el framework Android.

Desventaja del enfoque declarativo:

- Su uso es limitado: es ideal para declarar la apariencia de las componentes de UI, pero no provee manejo de eventos producidos por la interacción con los usuarios. Es necesario usar Java o Kotlin en este punto.

Diseño de Interfaces Gráficas en Android

¿Qué enfoque usamos?

La mejor práctica es usar ambos enfoques, declarativo y programático.
Es recomendable usar el enfoque declarativo, XML, para todo lo estático de la UI como el *layout* y las componentes o widgets. Y el enfoque programático para declarar qué pasa cuando el usuario interactúa con los widgets de la UI.
En síntesis en **XML** programamos **cómo se ve un widget** y en el **lenguaje de programación cómo reacciona** ante la interacción del usuario.

Aunque se utilicen dos enfoques para la creación de UI, el sistema Android creará objetos a partir de los XML. Finalmente se ejecutará código JAVA.
La clase R es un conjunto de referencias, generada automáticamente, que ayuda a conectar el mundo JAVA con el mundo XML
R.layout.estado apunta a /res/layout/estado.xml file.

Vistas y Layouts XML

Android organiza los elementos de UI en **vistas** y ***layouts***.

Vistas: es todo lo que se ve en la pantalla, botones, etiquetas, cajas de diálogo, etc. También se los conoce como widgets.

Layouts: organizan y agrupan las vistas. También se los conoce como ViewGroup. Las *vistas* están contenidas en *layouts*.

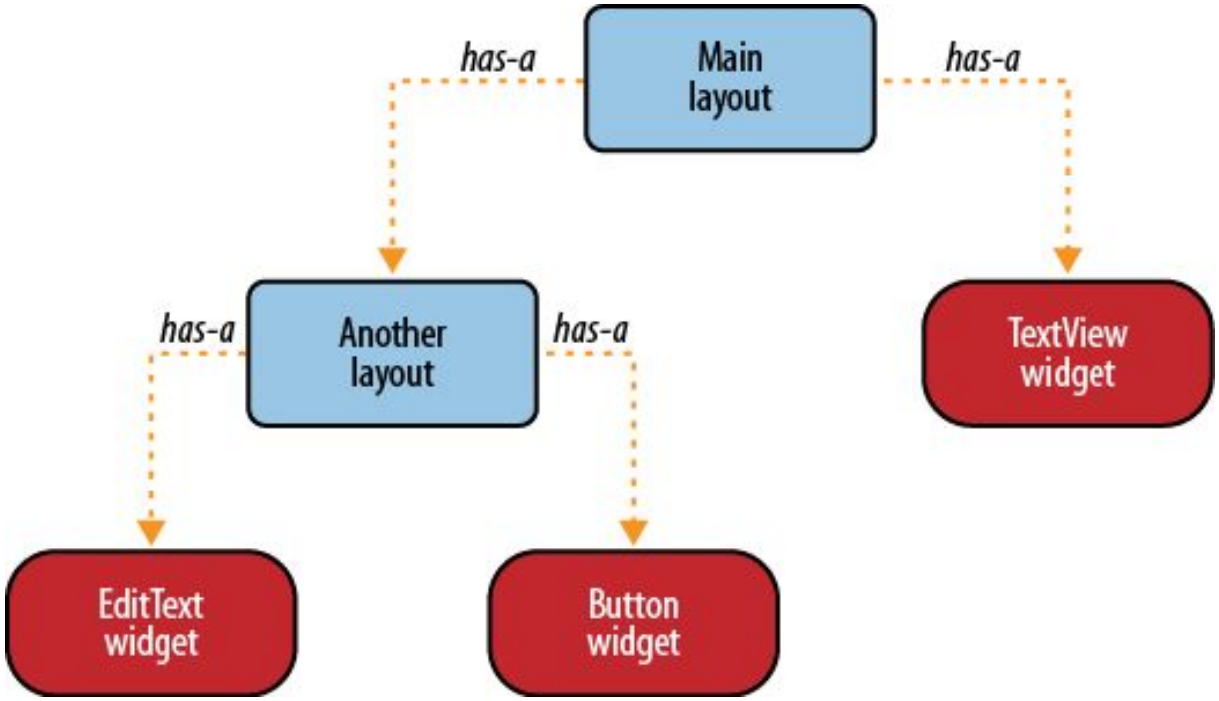
En términos de objetos de interfaz, los *layouts* son equivalentes a los **contenedores** y las *vistas* son las **componentes de UI**.

Los *layouts* se guardan en archivos XML ubicados en el directorio **res/layout** de Android.

Los **Activities** usan los *layouts* para mostrar las pantallas de la app.

Layouts XML

Un *layout* puede contener otros *layouts,* llamados hijos, permitiendo diseñar interfaces de usuario complejas.



Los *layouts* más representativos en Android son: **LinearLayout**, **RelativeLayout** y **GridLayout**

LinearLayout

El LinearLayout es un layout que alinea a todos sus hijos en una única dirección, que puede ser vertical u horizontal.

```
<LinearLayout 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:orientation="horizontal" >

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/app_test_boton1" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/app_test_boton2" />

   <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/app_test_boton3" />

</LinearLayout>
```

Las componentes hijas se muestran HORIZONTALMENTE

Vistas en el layout

LinearLayout

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:paddingLeft="16dp"
   android:paddingRight="16dp"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

   <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/to" />
   <EditText
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/subject" />
   <EditText
      android:layout_width="match_parent"
      android:layout_height="0dp"
      android:layout_weight="1"
      android:gravity="top"
      android:hint="@string/message" />
   <Button
      android:layout_width="100dp"
      android:layout_height="wrap_content"
      android:layout_gravity="right"
      android:text="@string/send" />
</LinearLayout>
```

Las componentes hijas se muestran VERTICALMENTE

IMPORTANCIA en relación al espacio que ocupa en la pantalla

ALINEACIÓN

match_parent y wrap_content

wrap_content: el ancho o la altura de la vista se establece como el tamaño mínimo necesario para adaptar el contenido a esta vista.

match_parent: el ancho o la altura de la vista se amplía hasta coincidir con el tamaño de la vista principal.

wrap_content

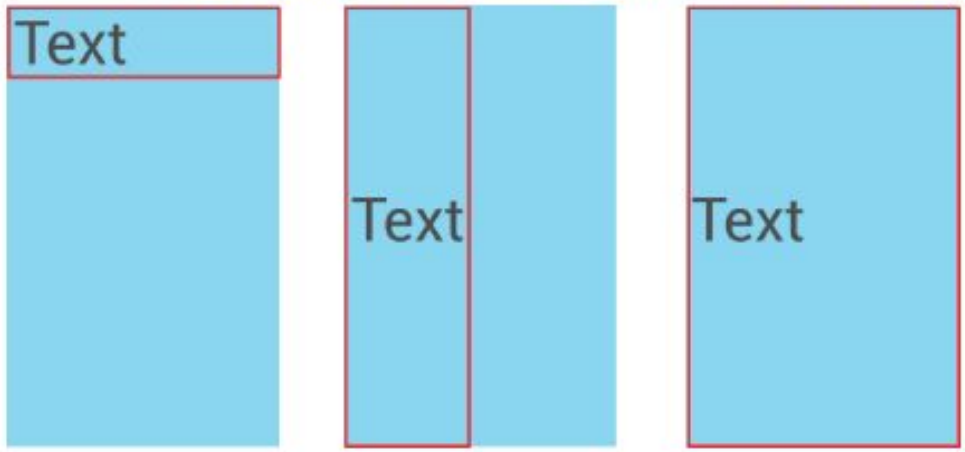
```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    ...
    />
```

match_parent

```
android:layout_width="wrap_content"
android:layout_height="match_parent"
```

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent"
```



RelativeLayout

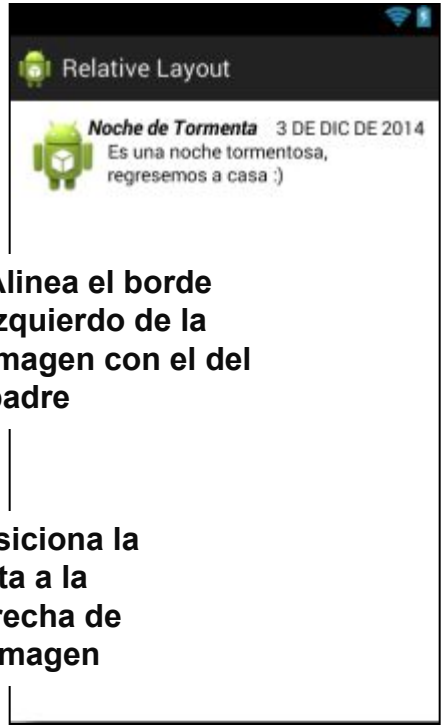
El **RelativeLayout** ubica sus vistas hijas en relación a sus hermanas y al padre. Usando **RelativeLayout** es posible posicionar una vista a la **izquierda**, **derecha**, **arriba** o **abajo** de sus vistas hermanas. También es posible posicionar una vista en relación a su padre alineado horizontalmente, verticalmente o según alguno de los bordes.

```
<?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="wrap_content"
   android:layout_height="wrap_content"
   android:layout_margin="5dp"
   android:padding="5dp" >
   <ImageView
      android:id="@+id/pic"
      android:layout_width="60dp"
      android:layout_height="60dp"
      android:layout_alignParentLeft="true"
      android:layout_gravity="center"
      android:layout_marginRight="3dp"
      android:src="@drawable/ic_launcher“/>
  <TextView
      android:id="@+id/info"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_below="@+id/title"
      android:layout_marginRight="3dip"
      android:layout_toRightOf="@+id/pic"
      android:gravity="left"
      android:text="Es una noche tormentosa, regresemos a casa :)"
      android:textColor="#000000"
      tools:ignore="HardcodedText" />
```

ID: identifica a la vista

Alinea el borde izquierdo de la imagen con el del padre

Posiciona la vista a la derecha de la imagen

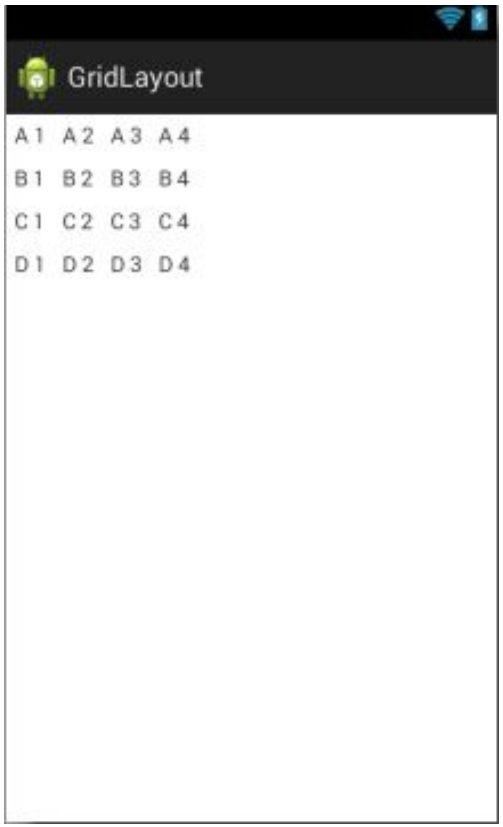



```
<TextView
      android:id="@+id/date"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_above="@+id/info"
      android:layout_alignParentRight="true"
      android:text="3 DE DIC DE 2014"
      android:textColor="#000000"
      tools:ignore="HardcodedText" />

  <TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignTop="@+id/pic"
      android:layout_marginRight="14dp"
      android:layout_toLeftOf="@+id/date"
      android:text="Noche de Tormenta"
      android:textColor="#040404"
      android:textSize="15sp"
      android:textStyle="bold|italic"
      android:typeface="sans"
      tools:ignore="HardcodedText" />

</RelativeLayout>
```

GridLayout

El **GridLayout** ubica sus hijos en una grilla rectangular. Este layout divide el área de dibujo en: filas, columnas y celdas. Soporta espaciado entre filas y columnas. Un widget puede ocupar un área rectangular de varias celdas contiguas.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:columnCount="4"
   android:orientation="horizontal"
   android:rowCount="4" >
   <TextView
      android:padding="5dp"
      android:text="A 1" />
   <TextView
      android:padding="5dp"
      android:text="A 2" />
   <TextView
      android:padding="5dp"
      android:text="A 3" />
   <TextView
      android:padding="5dp"
      android:text="A 4" />
   <TextView
      android:padding="5dp"
      android:text="B 1" />
   <TextView
      android:padding="5dp"
      android:text="B 2" />
 <TextView
      android:padding="5dp"
      android:text="B 3" />
 <TextView
      android:padding="5dp"
      android:text="B 4" />
 <TextView
      android:padding="5dp"
      android:text="C 1" />
  <TextView
      android:padding="5dp"
      android:text="C 2" />
  <TextView
      android:padding="5dp"
      android:text="C 3" />
  <TextView
      android:padding="5dp"
      android:text="C 4" />
  <TextView
      android:padding="5dp"
      android:text="D 1" />
  <TextView
      android:padding="5dp"
      android:text="D 2" />
  <TextView
      android:padding="5dp"
      android:text="D 3" />
  <TextView
      android:padding="5dp"
      android:text="D 4" />
</GridLayout>
```



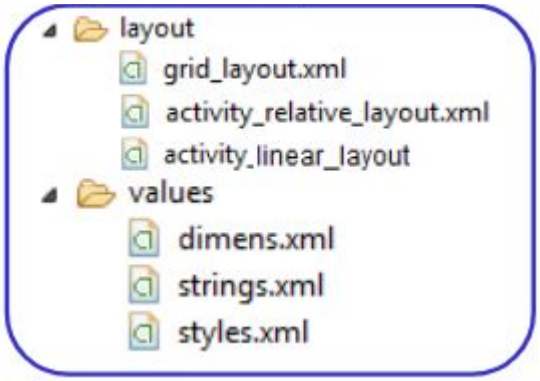
Layouts & Activity

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

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

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

inflating from XML



setContentView(): lee el archivo XML dado en el parámetro, lo parsea, crea todos los objetos correspondientes a los elementos XML, setea las propiedades de los objetos y (inflate) completa toda la vista. Luego la vista está lista para dibujarse.

¿Cómo referenciamos los widgets/vistas desde nuestra app?

ID: es un identificador entero único de un widget en un layout específico.
No todos los widgets necesitan tener un **ID.**
Los widgets que necesitan ser manipulados desde JAVA o Kotlin necesitan tener un **ID**.

El ID se define en XML, en el archivo de layout en el atributo **id.** Cuando la app se compila el ID es referenciado como un número entero y agregado a la **clase R**.

El formato del **ID** es el siguiente: **@+id/unNombre**

Ejemplo: **@+id/botonActualizar**

Código Kotlin que crea una instancia del widget para manipularlo:

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

Ejemplo

```
class StatuActivity : AppCompatActivity(), View.OnClickListener {
    val TAG: String = "StatuActivity"

    val editText: TextView = findViewById(R.id.editText)
    val updateButton: Button = findViewById(R.id.botonActualizar)

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)
        setContentView(R.layout.status)

        updateButton.setOnClickListener(this)
    }

    override fun onClick(v: View?) {
        Toast.makeText(applicationContext, editText.text, Toast.LENGTH_LONG)
        Log.d(TAG, "onClicked")
    }

}
```

Listener de Botón

Crea instancia a partir de los widgets

Registra el listener de los eventos del botón

Método de la interface OnClickListener, será invocado cuando se clickea el botón

Referenciando views vía vinculación de vista

La vinculación de vista genera una clase de vinculación para cada layout presente que contiene referencias directas a todas las *views* que tienen un ID en el archivo de layout correspondiente.

En la mayoría de los casos, la vinculación de vistas reemplaza a *findViewById*.

Para habilitar la vinculación de vista en un módulo, agrega el elemento *viewBinding* a su archivo *build.gradle*:

```
android {
        ...
        viewBinding {
                enabled = true
        }
    }
```

Ejemplo

result_profile.xml:

```
<LinearLayout ... >
<TextView android:id="@+id/name" />
<ImageView android:cropToPadding="true" />
<Button android:id="@+id/button" android:background="@drawable/rounded_button"/>
</LinearLayout>
```

```
private lateinit var binding: ResultProfileBinding

override fun onCreate(savedInstanceState: Bundle){
    super.onCreate(savedInstanceState)
    binding = ResultProfileBinding.inflate(layoutInflater)
    val view = binding.root
    setContentView(view)
}
```

La clase contiene referencias a la vista raíz y a todas las que tienen un ID. Esta clase tiene dos campos: un `TextView` llamado `name` y un `Button` llamado `button`. El campo `ImageView` del diseño no tiene ningún ID, por lo que no se hace referencia a él.

El nombre de la clase se genera a partir del nombre del archivo XML agregando la palabra "Binding" al final.

```
binding.name.text = "texto"
binding.button.setOnClickListener { Toast.makeText(applicationContext, "click",
Toast.LENGTH_LONG) }
```

Fragmentos

Un **fragmento** es una porción de la GUI que puede agregarse o eliminarse de la GUI de forma independiente al resto de elementos del Activity y puede reutilizarse en otros Activities.

Un fragmento corre dentro del contexto de un Activity, pero tiene su propio ciclo de vida.

Los fragmentos nos permiten dividir nuestra interfaz en varias porciones de forma que podamos diseñar diversas configuraciones de pantalla, dependiendo de su tamaño y orientación, sin tener que duplicar código. Por ejemplo

Fragmentos

Ciclo de vida

El ciclo de vida del fragmento está conectado al ciclo de vida de el Activity.

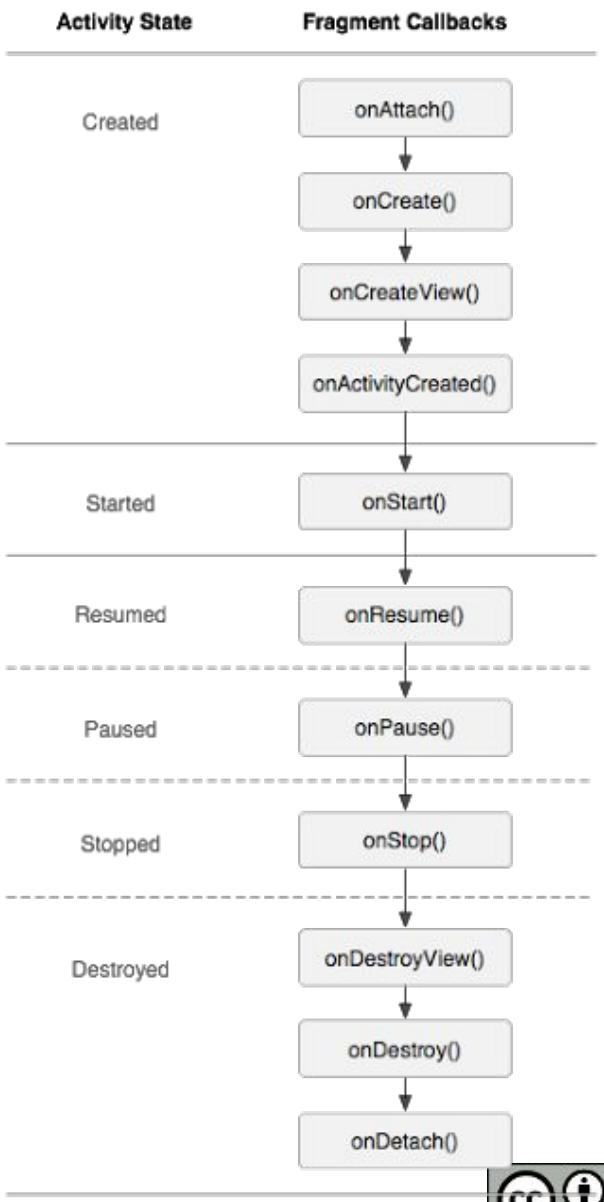
Si el Activity se detiene, sus fragmentos son detenidos; si el activity es destruido sus fragmentos también son destruidos

Generalmente se implementan los siguientes métodos:

onCreate() - El sistema llama a este método a la hora de crear el fragmento, normalmente iniciaremos los componentes esenciales del fragmento.

onCreateView() - El sistema llamará al método cuando sea la hora de crear la interface de usuario o vista, normalmente se devuelve la view del fragmento.

onPause() - El sistema llamará a este método en el momento que el usuario abandone el fragmento, por lo tanto es un buen momento para guardar información.



Fragmentos

Agregar un Fragmento a una Activity

A la hora de agregar un fragmento a una actividad lo podremos realizar de dos maneras:

1. Declarar el fragmento en el layout de la activity.
2. Agregar directamente el Fragment mediante programación Android.

Ejemplo declarativo de layout especificando un elemento fragment

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment
        android:name="com.fragments.FRAGMENTOS.FragmentUNO"
        android:id="@+id/fragment_uno"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
</LinearLayout>
```

Fragmentos

Agregar un Fragmento a una Activity

Para agregar o eliminar un fragmento, se debe usar un **FragmentManager** con el cual es posible crear una **FragmentTransaction. FragmentTransaction** tiene la API para agregar, eliminar, reemplazar y llevar a cabo otras transacciones con fragmentos.

Ejemplo de agregado de un Fragmento programáticamente

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.gestionar_fragments)
   val fragmentUNO: Fragment = FragmentUNO()
   val FT: FragmentTransaction = supportFragmentManager.beginTransaction()
   FT.add(R.id.fragment_container, fragmentUNO)
   FT.commit()
 }
```

Ejemplo de uso de Fragmentos

Ejemplo: CAMBIAR FRAGMENTOS

Cada vez que el usuario toque el botón se reemplazará el fragmento.

Ejemplo de uso de Fragmentos

layout/**gestionar_fragments.xml**

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

    <RelativeLayout
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_above="@+id/boton"
        android:layout_below="@+id/texto"/>

    <Button
        android:id="@+id/boton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="CAMBIA FRAGMENT"/>
</RelativeLayout>
```

Ejemplo de uso de Fragmentos

```
class GestionarFragments : AppCompatActivity() {
   private var bol = false
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout. gestionar_fragments)
       val fragmentUNO: Fragment = FragmentUNO()
       val fragmentDOS: Fragment = FragmentDOS()
       val FT: FragmentTransaction =  supportFragmentManager.beginTransaction()
       FT.add(R.id. fragment_container, fragmentUNO)
       FT.commit()
       val boton: Button = findViewById<View>(R.id. boton) as Button
       boton.setOnClickListener  {
            val FT: FragmentTransaction =
supportFragmentManager.beginTransaction()
            if (bol) {
                  FT.replace(R.id. fragment_container, fragmentUNO)
            } else {
                  FT.replace(R.id. fragment_container, fragmentDOS)
            }
            FT.commit()
            bol = !bol
       }
```

Ejemplo de uso de Fragmentos

```
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class FragmentUNO : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        return inflater.inflate(R.layout.fragment_uno, container, false)
    }

}
```

Idem para **FragmentDOS..**

Fragmentos

layout/**fragment_uno.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#FF8800" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:textColor="@android:color/white"
        android:text="FRAGMENT UNO"/>


</RelativeLayout>
```

ActionBar

Desde la versión 3.0, se introdujo en Android un nuevo elemento en la interfaz de usuario: la barra de acciones o **ActionBar**.

Situada en la parte superior de la pantalla, fue creada para que el usuario tuviera una experiencia unificada a través de las distintas aplicaciones.

Reúne varios elementos, los más habituales son:

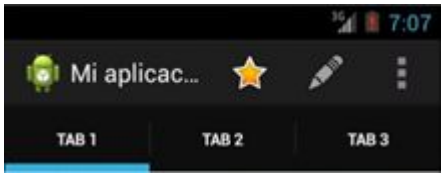
- el icono de la aplicación con su nombre
- los botones de acciones frecuentes.
- las acciones menos utilizadas (se sitúan en un menú desplegable, que se abrirá desde el botón Overflow).

ActionBar

Si la aplicación dispone de pestañas (tabs), estas podrán situarse en la barra de acciones.

También pueden añadirse otros elementos, como listas desplegables y otros tipos de widgets incrustados.

En caso de disponer de menos tamaño de pantalla el sistema puede redistribuir los elementos y pasar alguna acción al menú de «Overflow».



ActionBar

- Los dispositivos anteriores a la versión 3.0 requerían una ***tecla física*** para mostrar el menú de la actividad.

- Con esta versión la tecla física deja de ser un requisito para los dispositivos y ***los menús pasan a mostrarse en la barra de acciones***.

- En los dispositivos que dispongan del botón físico, es posible que los tres puntos que representan el menú de Overflow no se representen en la barra de acciones. En este caso hay que usar el botón físico para desplegar este menú.

- Desde finales de 2013 también podemos utilizar la ActionBar en versiones anteriores a la 3.0 descargando una librería de compatibilidad.

- A partir de la revisión 18 contamos con una versión del Action Bar conocida como **ActionBarCompat**, compatible con cualquier versión Android a partir de la 2.1 (API 7).

ActionBar

- Hay que agregar la última versión de la librería de compatibilidad: **Android Support Library** en el bloque dependencies del archivo **build.gradle** del módulo.

```
dependencies {
    ...
    implementation 'androidx.appcompat:appcompat:1.4.2'

}
```

ActionBar

Ejemplo de menú que se muestra en la barra de acciones

res/menu/menu_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/ tools"
      tools:context=".MainActivity">
       <item android:title="@string/action_settings"
             android:id="@+id/action_settings"
             android:icon="@android:drawable/ic_menu_preferences"
             android:orderInCategory="5"
             app:showAsAction="never"/>
       <item android:title="Acerca de..."
             android:id="@+id/acerca_de"
             android:icon="@android:drawable/ic_menu_info_details"
             android:orderInCategory="10"
             app:showAsAction="ifRoom|withText"/>
       <item android:title="Buscar"
             android:id="@+id/menu_buscar"
             android:icon="@android:drawable/ic_menu_search"
             android:orderInCategory="115"
             app:showAsAction="always|collapseActionView"/>
</menu>
```

- Las acciones que indiquen en el atributo **showAsAction** la palabra **always** **serán mostrados siempre**, sin importar si caben o no.
- Las acciones que indiquen **ifRoom** serán mostradas en la barra de acciones **si hay espacio disponible**, y serán movidas al menú de Overflow si no lo hay.
- Si se indica **never** la acción **nunca se mostrará** en la barra de acciones, sin importar el espacio disponible.
- Las acciones son ordenadas de izquierda a derecha según lo indicado en **orderInCategory**, con las **acciones con un número más pequeño más a la izquierda**.

ActionBar

Una vez definido el menú desde su archivo .xml (res/menu/menu_main.xml) hay que indicar en el layout de nuestra activity el **toolbar** con los atributos deseados.

```
//Layout
. . .
<androidx.appcompat.widget.Toolbar
   android:id="@+id/my_toolbar"
   android:layout_width="match_parent"
   android:layout_height="?attr/actionBarSize"
   android:background="?attr/colorPrimary"
   android:elevation="4dp"
   android:theme="@style/ThemeOverlay.AppCompat.ActionBar"
   app:popupTheme="@style/ThemeOverlay.AppCompat.Light"/>
```

ActionBar

Con el menú definido en el archivo .xml (res/menu/menu_main.xml) solo resta indicar que se soporta el uso de ActionBar y luego *inflarlo* desde nuestra activity.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        setSupportActionBar(findViewById(R.id.my_toolbar))
    }
    ...
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }
}
```

ActionBar

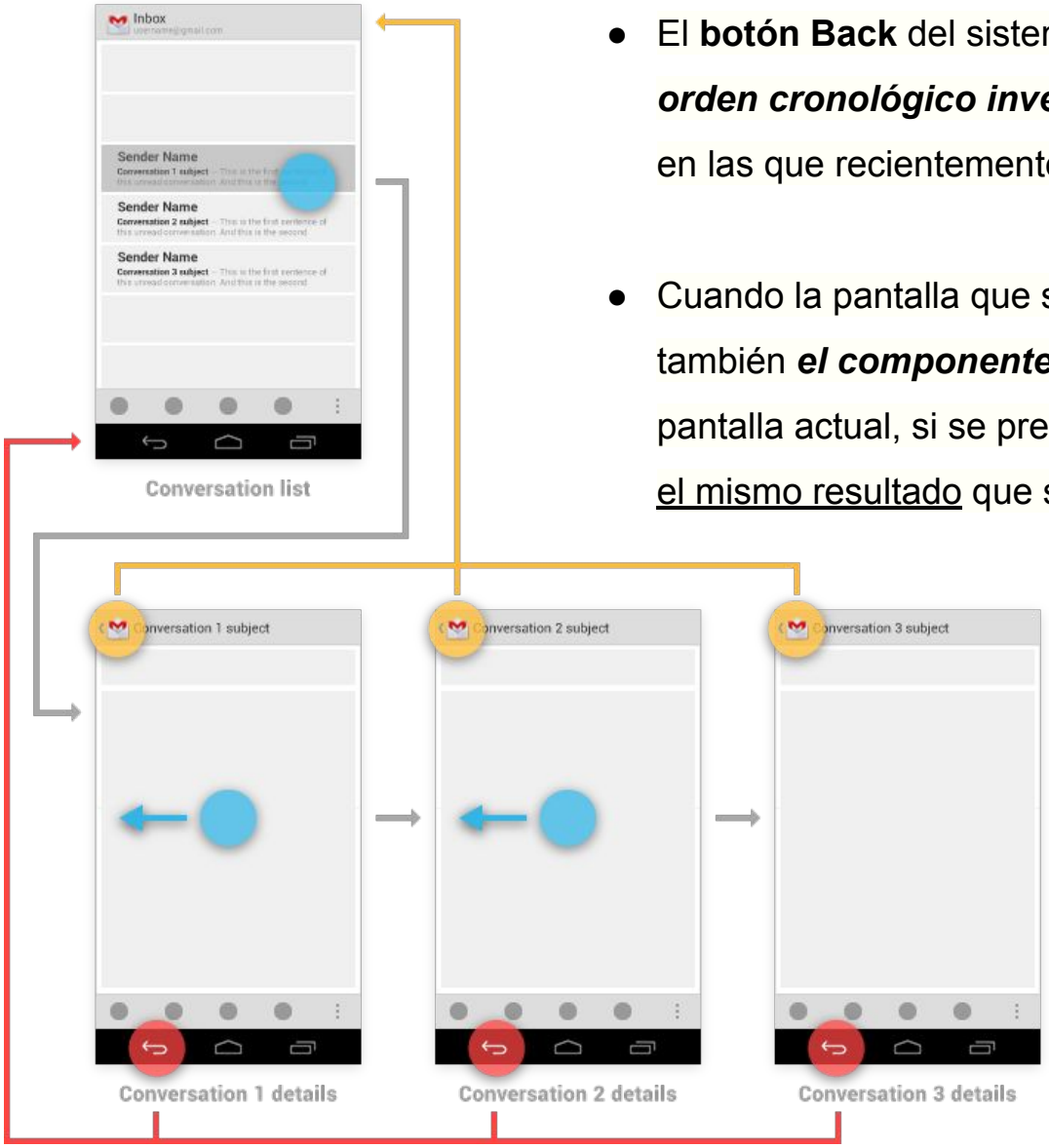
Luego cuando el usuario pulsa sobre el ActionBar, el activity recibirá el llamado a **onOptionsItemSelected**() con el MenuItem seleccionado

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.action_settings -> {
            Log.i("ActionBar", "action_settings")
            true
        }
        R.id.acerca_de -> {
            Log.i("ActionBar", "acerca_de")
            true
        }
        R.id.menu_buscar -> {
            Log.i("ActionBar", "menu_buscar")
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

Navegación con los botones Back y Up

- En Android 3.0, se introdujeron cambios significativos en el comportamiento global de la navegación.
- Utilizando las pautas de navegación con los botones Back y Up, la navegación en su aplicación será predecible y confiable para los usuarios.
- En Android 2.3 y versiones anteriores, se confió en el **botón Back** del sistema para respaldar la navegación dentro de una aplicación. Con la introducción de las barras de acciones en Android 3.0, apareció un segundo mecanismo de navegación: **el botón Up**, que consiste en el icono de la aplicación y una pequeña flecha a la izquierda.
- El **botón Up** se utiliza para navegar dentro de una aplicación sobre la base de las ***relaciones jerárquicas*** entre pantallas.
- Si una pantalla aparece en la parte superior de una aplicación (es decir, en el inicio de la aplicación), no debe incluir el botón Up.

Navegación con los botones Back y Up



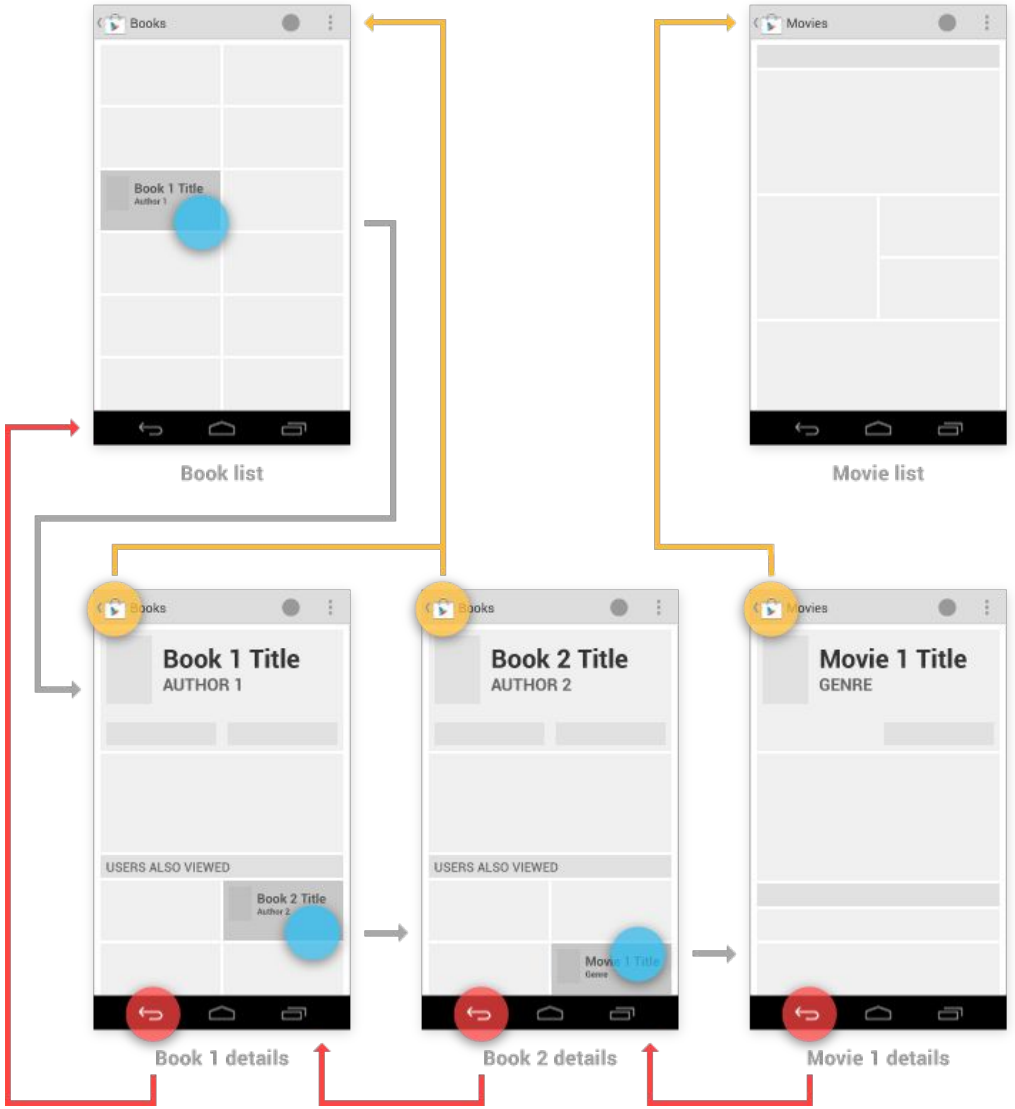
- El **botón Back** del sistema se utiliza para navegar, ***en orden cronológico inverso***, por el historial de pantallas en las que recientemente trabajó el usuario.

- Cuando la pantalla que se visitó anteriormente es también ***el componente jerárquico primario*** de la pantalla actual, si se presiona el botón Back, se obtendrá el mismo resultado que si se presiona el botón Up.

Navegación con los botones Back y Up

Se puede hacer que el botón Up sea más inteligente.

Por ejemplo en ese caso, mediante el botón Up ***podrá regresar a un contenedor*** *(Películas)* ***por el que el usuario no navegó anteriormente****.*



Implementando la navegación UP

- Para implementar la navegación UP, hay que declarar el parent de cada activity.
- Desde la versión de Android 4.1 (API level 16), se declara el parent de cada activity especificando el atributo android:parentActivityName en el elemento <activity>.
- Si la aplicación es menor a la versión 4, hay que incluir la librería Support Library, especificando el parent del activity mediante android.support.PARENT_ACTIVITY y el atributo android:parentActivityName.

```
<application . . .>
   <!-- Parent activity -->
   <activity
       android:name=".MainActivity"  . . .>
    . . .
   </activity>
   <!-- Child activity -->
   <activity
       android:name=".ChildActivity"
       android:label="@string/app_name"
       android:parentActivityName=".MainActivity" >
<!-- Parent activity meta-data to support 4.0 and lower -->
     <meta-data
          android:name="android.support.PARENT_ACTIVITY"
          android:value=".MainActivity" />

   </activity>
</application>
```

Implementando la navegación UP

- Para habilitar el botón Up en el ActionBar

```
override fun onCreate(savedInstanceState: Bundle?) {
  ...
  supportActionBar?.setDisplayHomeAsUpEnabled(true)
}
```

- No es necesario recuperar la acción en el método **onOptionsItemSelected()** del activity. En su lugar, se delega al método de la superclase que responde a la selección Up navegando al **activity parent**, tal cual está especificado en el **manifest.xml** de la aplicación.

Jetpack Compose

- Jetpack Compose es un framework para crear IU nativas de Android, simplificando el desarrollo utilizando con menos código Kotlin.
- Se basa en definir la IU de manera programática via funciones que admiten composición en lugar de enfocarse en el proceso de construcción de la IU.
- Para crear una función que admita composición, agrega `@Composable` al nombre de la función.

```
. . .
import androidx.compose.runtime.Composable

class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContent {
           Saludo("Mundo")
       }
   }
}

@Composable
fun Saludo(name: String) {
   Text(text = "Hola $name!")
}
```

Jetpack Compose: Vista previa en Android Studio

- La anotación `@Preview` permite obtener una vista previa sin tener que utilizar un emulador o dispositivo Android.
- La anotación se debe usar en una función que no acepte parámetros.
- En el ejemplo anterior, se crea una segunda función llamada `SaludoPreview` que llama a `Saludo` con el parámetro adecuado.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Saludo( name: "Mundo")
        }
    }
}

@Composable
fun Saludo(name: String) {
    Text(text = "Hola $name!")
}

@Preview
@Composable
fun SaludoPreview() {
    Saludo( name: "Mundo")
}
```

Jetpack Compose: Material Design

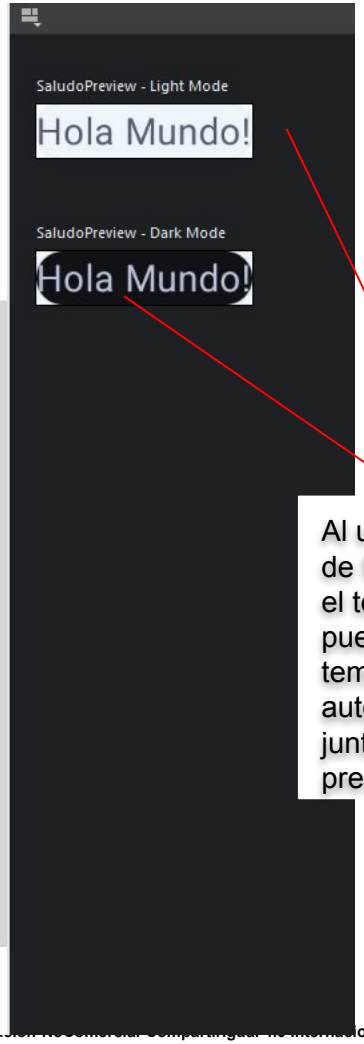
- Jetpack Compose ofrece una implementación de Material Design 3 con elementos de la IU listos para usar.
- Material Design se basa en tres pilares: `Color`, `Typography` y `Shape`.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyApplicationTheme {
                    Saludo( name: "Mundo")
            }
        }
    }
}

@Composable
fun Saludo(name: String) {
    Surface(shape = MaterialTheme.shapes.medium, shadowElevation = 1.dp) {
        Text(
            text = "Hola $name!",
            color = MaterialTheme.colorScheme.secondary,
            style = MaterialTheme.typography.bodyMedium
        )
    }

}

@Preview(name = "Light Mode")
@Preview(
    uiMode = Configuration.UI_MODE_NIGHT_YES,
    showBackground = true,
    name = "Dark Mode"
)
@Composable
fun SaludoPreview() {
    MyApplicationTheme {
        Saludo( name: "Mundo")
    }
}
```

`Shape` junto con Surface permite personalizar la forma, como por ejemplo el `shadowElevation` y el `padding` de un texto.

`MaterialTheme.colorScheme` aplica los colores del tema unido.

La tipografía del Material está disponibles en `MaterialTheme.typography`



Al utilizar los colores de Material Design, el texto y el fondo se puede habilitar el tema oscuro automáticamente, junto a varias vistas previas.