

Apuntes

[Descargar estos apuntes](#)

Tema 7. Fragments

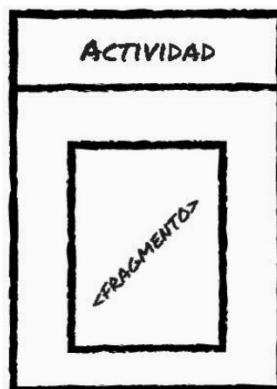
Índice

1. [Introducción](#)
2. [Ciclo de vida de un Fragment](#)
3. [Subclases de Fragment](#)
4. [Crear un Fragment](#)
5. [Agregar un Fragment a una Activity](#)
 1. [Agregar Fragments Estáticos](#)
 2. [Agregar Fragment Dinámico](#)
 3. [Gestionar Fragments.](#)
6. [Comunicar Fragments y Activitys](#)
 1. [Comunicación mediante ViewModel](#)
 2. [Comunicación mediante Interfaces](#)
7. [Pila de actividades](#)
 1. [Tasks y BackStacks](#)
 2. [FragmentManager](#)
 3. [Jetpack Navigation](#)
8. [Permisos con Fragments](#)
9. [DialogFragment](#)
10. [Vistas Deslizantes y Tabs](#)
 1. [ViewPager](#)
 2. [TabsLayout](#)

Introducción

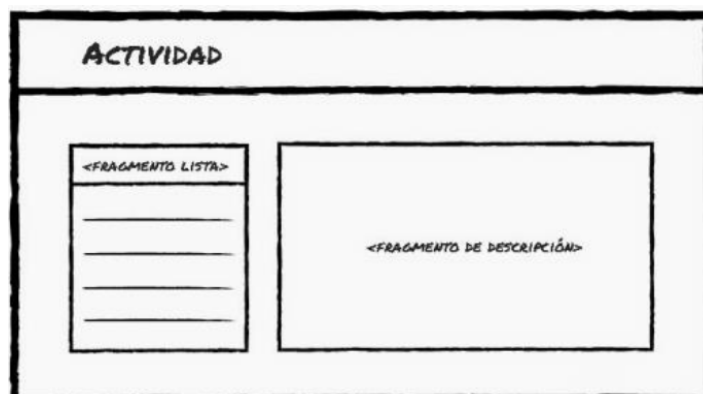
La necesidad de usar fragmentos nace con la versión 3.0 (API 11) de Android, debido a los múltiples tamaños de pantalla que estaban apareciendo en el mercado y a la capacidad de orientación de la interfaz (Landscape y Portrait). Estas características necesitaban dotar a las aplicaciones Android de la capacidad para adaptarse y responder a la interfaz de usuario sin importar el dispositivo.

Un **fragment** es una sección *modular* de interfaz de usuario embebida dentro de una actividad anfitriona, el cual permite versatilidad y optimización de diseño. Se trata de miniactividades contenidas dentro de una actividad anfitriona, manejando su propio diseño (un recurso layout propio) y ciclo de vida.



Estas nuevas entidades permiten reusar código y ahorrar tiempo de diseño a la hora de desarrollar una aplicación. Los fragmentos facilitan el despliegue de tus aplicaciones en cualquier tipo de tamaño de pantalla y orientación.

Otra ventaja de usarlos es que permiten crear diseños de interfaces de usuario de múltiples vistas. ¿Qué quiere decir eso?, que **los fragmentos son imprescindibles para generar actividades con diseños dinámicos**, como por ejemplo el uso de pestañas de navegación, expand and collapse, stacking, etc.

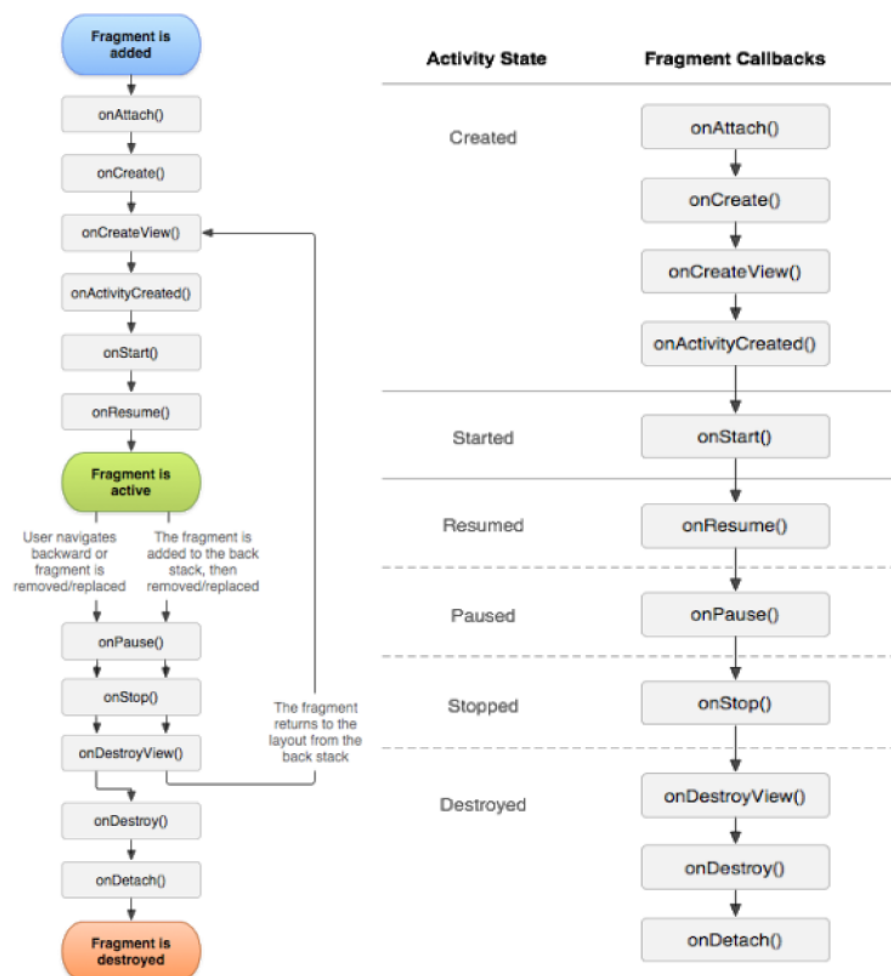


Un fragmento se ejecuta en el contexto de una actividad, pero tiene su propio ciclo de vida y por lo general su propia interfaz de usuario, recibe sus propios eventos de entrada, y se pueden agregar o quitar mientras que la actividad exista.

Un fragmento tiene que estar siempre integrado en una activity, de forma que se verá afectado por el propio ciclo de vida de la activity. Por ejemplo: *cuando una activity se detiene, lo hacen todos los fragmentos de la misma; cuando ésta se destruye, lo hacen también todos sus fragmentos. Sin embargo, mientras una activity está en ejecución, se puede manipular cada uno de los fragmentos incluidos en ella de forma independiente, tanto añadiéndolos como eliminándolos.*

Ciclo de vida de un Fragment

La imagen de la izquierda representa el ciclo de vida de un fragmento. Y la imagen de la derecha los estados de sus métodos.



Generalmente los métodos más usados a la hora de implementar un fragmento son los siguientes:

- **`onCreate()`** . El sistema llama a este método a la hora de crear el fragmento. Normalmente en él, iniciaremos los componentes esenciales del fragmento.
- **`onCreateView()`** . El sistema llama al método cuando es la hora de crear la interface de usuario o vista, es decir, asociar el layout al fragment, normalmente se devuelve la view del fragmento.

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

Aunque en el ciclo de vida nos encontramos con los siguientes métodos callback, relacionados con el ciclo de vida de una actividad. Averigüemos un poco sobre ellos:

- **onAttach():** . Es invocado cuando el fragmento ha sido asociado a la actividad anfitriona.
- **onActiviyCreated()** . Se ejecuta cuando la actividad anfitriona ya ha terminado la ejecución de su método *onCreate()*.
- **onCreate()** . Este método es llamado cuando el fragmento se está creando. En el puedes inicializar todos los componentes.
- **onCreateView()** . Se llama cuando el fragmento será dibujado por primera vez en la interfaz de usuario. En este método crearemos el view que representa al fragmento para retornarlo hacia la actividad.
- **onStart()** . Se llama cuando el fragmento esta visible ante el usuario. Obviamente depende del método *onStart()* de la actividad.
- **onResume()** . Es ejecutado cuando el fragmento está activo e interactuando con el usuario. Esta situación depende de que la actividad anfitriona este primero en su estado Resume.
- **onStop()** . Se llama cuando un fragmento ya no es visible para el usuario debido a que la actividad anfitriona está detenida o porque dentro de la actividad se está gestionando una operación de fragmentos.
- **onPause()** . Al igual que las actividades, onPause se ejecuta cuando se detecta que el usuario dirigió el foco por fuera del fragmento.
- **onDestroyView()** . Este método es llamado cuando la jerarquía de views a la cual ha sido asociado el fragmento ha sido destruida.
- **onDetach()** . Se llama cuando el fragmento ya no está asociado a la actividad anfitriona.

Subclases de Fragment

A parte de crear un Fragment directamente, Android nos ofrece la posibilidad de utilizar las siguientes subclases de Fragment, de estas subclases hablaremos posteriormente

- DialogFragment - Muestra un cuadro de dialogo flotante.
- ListFragment - Muestra una lista de elementos.
- PreferenceFragment - Muestra una lista de preferencias.

Crear un Fragment

Para crear un fragment primero deberemos extender la clase `Fragment` y sobrescribir el método `onCreateView()` en el que devolveremos la vista de dicho fragmento. Vamos a ver el ejemplo y lo explicamos a continuación.

FragmentUNO.kt

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

👉 Comentar que previamente se ha creado el layout para este fragmento y se corresponde con `fragment_uno.xml`.

```
<?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:background="#CC00CC00">  
    <TextView  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Información fragment1"/>  
</LinearLayout>
```

🔗 Al sobrescribir el método `onCreateView()` podemos observar que de serie nos da la posibilidad de utilizar un `LayoutInflater`, un `ViewGroup` y un `Bundle`. El `LayoutInflater` normalmente lo utilizaremos para inflar el layout de nuestro fragment. El `ViewGroup` será la vista padre donde se insertara el layout de nuestro fragment. Y por último el `Bundle` podremos utilizarlo para recuperar datos de una instancia anterior de nuestro fragment. De esta manera ya tendremos creado un fragment que nos devolverá una vista y que podremos insertar en cualquier activity de nuestro proyecto.

Agregar un Fragment 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. Este fragment tendrá la cualidad de no ser eliminado o sustituido por nada, de lo contrario tendremos errores. Se le da el nombre de **fragment estático o final**.
2. Agregar directamente el Fragment mediante programación Android. Éste sí que se podrá eliminar o sustituir por otro fragment u otro contenido. **Se les da el nombre de fragment dinámico**.

Agregar Fragments Estáticos

Lo primero que tenemos que hacer es crear el layout de nuestra activity especificando un elemento fragment.

activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_uno"
        android:name="com.ejemplos.b3.ejemplofragmentv1.FragmentUno"
        android:layout_weight="0.5"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"/>
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_dos"
        android:name="com.ejemplos.b3.ejemplofragmentv1.FragmentDos"
        android:layout_weight="0.5"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"/>
</LinearLayout>
```

✂ Simplemente creamos el elemento [FragmentContainerView](#) y especificamos a través del atributo **android:name** la ubicación de nuestro fragmento (nombre de paquete donde está ubicado el fragmento). `FragmentContainerView` es un contenedor personalizado para Fragmentos que extiende de `FrameLayout`, por lo que maneja las transacciones de manera más fiable.

Es recomendable crear un identificador único para cada fragmento, que nos puede servir para restaurar fragmentos o incluso para realizar transacciones o eliminación de estos.

Una vez creado el layout de la activity simplemente creamos una actividad y le aplicamos el método **setContentview()** indicando la id del layout que acabamos de crear. El resultado puede ser el siguiente, teniendo en cuenta que se debe añadir la clase `FragmentDos` similar a la `FragmentUno`:



✍️ **Crea una App FragmentEstatico, para probar los fragments estáticos del ejemplo anterior**

Agregar Fragment Dinámico

De esta manera podemos agregar un fragment a una activity en cualquier momento. Simplemente indicaremos la id de una vista padre (ViewGroup, recomendado usar `FragmentContainerView`) donde deberá colocarse el fragment.

En el ejemplo vamos a agregar un fragment dinámico junto a uno estático. Lo primero es definir en el layout de la activity un espacio donde poder añadir el fragment, para ello usaremos el segundo `FragmentContainerView` con id `fragment_container`.

```
<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"
    tools:context=".MainActivity">
    <androidx.fragment.app.FragmentContainerView
        android:name="com.ejemplos.b3.ejemplofragmentv1.FragmentUno"
        android:id="@+id/fragment_uno"
        android:layout_weight="0.5"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"/>
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_container"
        android:layout_weight="0.5"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:background="#3355CC00">
    </androidx.fragment.app.FragmentContainerView>
</LinearLayout>
```

Nos creamos una clase para el fragment2, similar a la anterior y un layout para gestionar el aspecto. Pasemos a añadir el fragment por código. Para ello tenemos que utilizar la API `FragmentManager` y a través de su método `add()` añadiremos

el fragmento a la vista padre. Después de añadir el fragment tendremos que terminar la transacción a través del método `commit()`.

Vamos a ver el ejemplo de la clase **MainActivity**:

```
class MainActivity : AppCompatActivity()
{
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        7 val fragmentManager=supportFragmentManager
        8 val fragmentTransaction=fragmentManager.beginTransaction()
        val fragmentDos=FragmentDos()
        10 fragmentTransaction.add(R.id.fragment_container,fragmentDos)
        11 fragmentTransaction.commit()
    }
}
```

👉 Si se quiere usar Binding con Fragments, informate de su uso en el siguiente enlace, [vincular vistas](#)

🔗 En este caso nuestra activity carga un layout con un View **fragment** para el fragmento estático y un **FrageLayout** que no muestra nada pero nos va a servir como contenedor para agregar el fragment dinámico programáticamente.

Lo primero que hacemos es crear una instancia de **FragmentManager** a través de su método `getSupportFragmentManager()`, **Línea 7**. De esta manera estamos creando un objeto que nos servirá para manejar los fragmentos.

A continuación creamos una instancia de **FragmentTransaction** para realizar la transacción de fragmentos. A través del método `beginTransaction()` **Línea 8**, indicamos que vamos a realizar una transacción de fragmentos.

Lo siguiente es crear una instancia de nuestro **FragmentDos** y añadirla a la transacción a través del método `add()`, que nos pide como parámetros la id del ViewGroup o vista padre donde se colocara dicho fragmento (en este caso es la id del FrameLayout que comentábamos antes) y como segundo parámetro nos pide la instancia del fragmento que se mostrara dicha vista **Línea 10**.

Para terminar la transacción siempre deberemos declarar el método `commit()` **Línea 11**.

🔗 **Crea una App fragmentV1, para probar el ejemplo anterior, el fragment1 será estático mientras que el fragment2 será añadido de forma dinámica desde el código**

Gestionar Fragments.

En el punto anterior hemos hablado de transacciones, una transacción simplemente es una acción que nos permite agregar, reemplazar, eliminar o incluso realizar otras acciones cuando trabajamos con fragmentos. Estas transacciones pueden ser apiladas por la activity de acogida, permitiendo así al usuario navegar entre fragmentos mientras la activity siga en ejecución.

Cada transacción es un conjunto de cambios que se realizan al mismo tiempo. Podremos realizar dichos cambios a través de los métodos `add()`, `replace()`, `remove()` terminando la transacción con el método `commit()`.

Para añadir la transacción a la pila de retroceso de la activity utilizaremos el método `addToBackStack()` para cada transacción que realicemos. Esta pila será administrada por la activity y permitirá al usuario volver a un fragmento anterior pulsando la tecla volver del smartphone o tablet.

Por otra parte a través del método `setTransaction()` podemos establecer el tipo de animación para cada transacción.

Para este caso se ha creado un ejemplo que muestra un layout con un ViewGroup para mostrar los fragmentos y un botón que servirá para añadir fragmentos a la pila de retroceso. Empezaremos creando el layout que mostrará la activity:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_above="@+id/boton">
    </androidx.fragment.app.FragmentContainerView>
    <Button
        android:id="@+id/boton"
        android:text="CAMBIAR"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"/>
</RelativeLayout>
```

Una vez creado el layout deberemos crear la siguiente activity:

Mainactivity.kt

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        val f1 = FragmentUno();  
        val f2 = FragmentDos();  
7        var bol=true  
8        val boton = findViewById<Button>((R.id.boton));  
        boton.setOnClickListener{  
10            val FT = getSupportFragmentManager().beginTransaction();  
            if (bol) {  
                FT.replace(R.id.fragment_container, f1);  
            } else {  
                FT.replace(R.id.fragment_container, f2);  
            }  
16            FT.addToBackStack(null);  
            FT.commit();  
18            bol = if(bol)false else true;  
        }  
    }  
}
```

✎ Comentar que previamente se han creado dos fragmentos "FragmentUNO" y "FragmentDOS", cada uno con su layout.

En la activity primero creamos un valor booleano que nos servirá para cambiar entre un fragmento u otro cada vez que el usuario pulse el botón **Línea 18**.

Establecemos el layout de la activity y creamos dos instancias de nuestros fragmentos. A continuación declaramos nuestro botón y le aplicamos un listener. Cada vez que el usuario pulse el botón se creara una transacción añadiendo a la pila un fragmento u otro a través del método `addToBackStack()` que pide como parámetro un tag o identificador para la transacción que se va a realizar **Línea 16**.

Se finaliza la transacción con el método `commit()` y cambiamos el valor booleano.

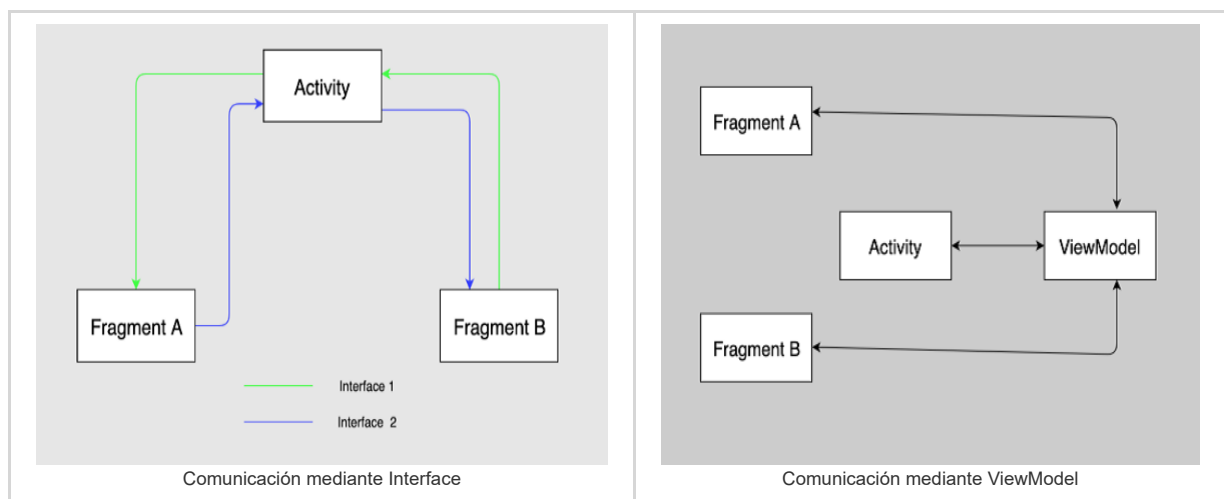
✎ **Prueba el anterior ejemplo FragmentV2: botón que te vaya cargando uno u otro fragment y al mismo tiempo que se acumulen en la pila. Probar que al pulsar el botón de retroceso, se pasa por cada uno de los fragments que se han cargado anteriormente.**

Comunicar Fragments y Activitys

Cuando hablamos sobre [la comunicación de fragmentos](#), debemos tener en cuenta las siguientes premisas:

- Los fragmentos no pueden ni deben comunicarse directamente.
- La comunicación entre fragmentos debe hacerse a través de los asociados activity.
- Los fragmentos no necesitan conocer quién es su actividad principal.

A partir de lo anterior, podemos deducir que dos objetos Fragment nunca deben comunicarse directamente. Hay varias maneras de realizar la comunicación entre estos, pero vamos a estudiar las dos más recomendadas. Estas dos formas son: mediante la **implementación de una interface** o mediante un elemento **ViewModel**.



Comunicación mediante ViewModel

[ViewModel](#) es una de las nuevas incorporaciones a la implementación de Android y la que google recomienda para la comunicación entre fragments.

Un ViewModel siempre se crea en asociación con un ámbito (un fragmento o una actividad) y se conservará mientras el ámbito esté activo. Por ejemplo, si es una Actividad, hasta que se termine. En otras palabras, esto significa que un ViewModel no se destruirá si su propietario se destruye por un cambio de configuración (por ejemplo, rotación). La nueva instancia del propietario se volverá a conectar al ViewModel existente.

El siguiente paso será crear una clase que derive de ViewModel parecida a la siguiente, y teniendo en cuenta que tipo de datos queremos pasar entre los fragments, en este caso un String aunque podría ser un objeto o incluso colecciones de estos:

```

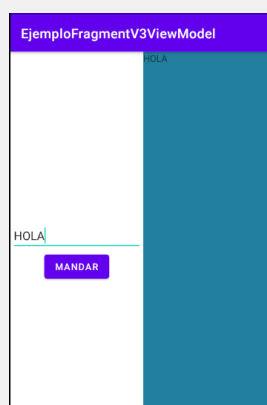
class ItemViewModel : ViewModel() {
    private val liveData=MutableLiveData<String>()
    val getItem: LiveData<String> get() = liveData
    fun setItem(item: String) {
        liveData.value = item
    }
}

```

✧ Como podemos ver en el código anterior, aparece un elemento nuevo para exponer los datos que lo hemos llamado `liveData` de tipo **MutableLiveData** que a su vez extiende de **LivData** , este elemento es un titular de datos que es capaz de ser observado para enviar solo actualizaciones de datos cuando su observador está activo, puede contener cualquier tipo de datos, y además de eso, es consciente del ciclo de vida para mandar las actualizaciones de datos solamente si el observador está activo.

Para observar un elemento **LivData**, tenemos la clase **Observer** , que a través de su objeto podremos saber si está en estado activo (su ciclo de vida está en el estado **STARTED** o **RESUMED**) o inactivo (en cualquier otro caso). **LivData** solo notifica a los observadores activos sobre las actualizaciones.

🎓 Vamos a suponer un ejemplo sencillo en el que, usando el **ViewModel** mostrado con anterioridad, en una **activity** se cargan dos **fragments** a la vez y se realiza **la comunicación entre los dos fragments**. Uno tendrá un **EditText** para introducir un valor y un botón, en el otro se mostrará el contenido del **EditText** al pulsar el botón. Para ello deberemos crear dos clase **Fragments**, una que será el detalle **FragmentSecundario** y que solo tendrá un texto y una clase **FragmentPrimario** que tendrá los elementos necesarios. El resultado será parecido al siguiente:



El layout del fragment primario podría ser, **fragment_primario.xml**:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">
    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="Introduce Texto a pasar"
        android:id="@+id/texto"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Mandar"
        android:id="@+id/boton"/>
</LinearLayout>
```

El layout del fragment detalle podría ser, **fragment_secundario.xml**:

```
<?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:background="#247f9e">
    <TextView
        android:text="TEXT0"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/texto"/>
</LinearLayout>
```

El layout de la actividad principal, **activity_main.xml**:

```
<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"
    tools:context=".MainActivity">
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_uno"
        android:layout_width="200dp"
        android:layout_height="match_parent"/>
    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_dos"
        android:layout_width="200dp"
        android:layout_height="match_parent">
    </androidx.fragment.app.FragmentContainerView>
</LinearLayout>
```

La actividad principal con la carga de los dos fragments podría quedar así,
MainActivity.kt:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val fM: FragmentManager = supportFragmentManager
        val fT: FragmentTransaction = fM.beginTransaction()
        fT.add(R.id.fragment_uno, FragmentPrimario())
        fT.add(R.id.fragment_dos, FragmentSecundario())
        fT.commit() }
}
```

El Fragment primario podría ser como el que sigue, **FragmentPrimario.kt**:

```
class FragmentPrimario : Fragment() {  
    2 private val model:ItemViewModel by activityViewModels()  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View?  
    8 {  
        val view: View = inflater.inflate  
            (R.layout.fragment_primario, container, false)  
        view.findViewById<Button>(R.id.boton).setOnClickListener{  
    12 model.setItem(requireActivity().  
            findViewById<EditText>(R.id.texto).text.toString())  
        }  
        return view  
    }  
}
```

✎ **Línea 2** se crea una propiedad del **ViewModel** creado con anterioridad, utilizando el delegado **activityViewModels** para obtener una referencia al **ViewModel** en el ámbito de su actividad y que se pueda compartir entre las distintas vistas o fragments, mientras el ciclo de vida lo permita. Para poder incluir este delegado, se tiene que añadir la siguiente dependencia (a día de hoy):

```
implementation 'androidx.fragment:fragment-ktx:1.3.2'
```

Línea 14, cuando se pulse el botón se modificará el valor del LiveData con la información del editText

El Fragment con el detalle, **FragmentSecundario.kt**:

```
class FragmentSecundario:Fragment() {  
    2 private val model:ItemViewModel by activityViewModels()  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?): View? {  
        val view: View = inflater.inflate  
            (R.layout.fragment_secundario, container, false)  
    9 val nameObserver = Observer<String>{cadena ->  
        view.findViewById<TextView>(R.id.texto).text=cadena}  
    11 model.getItem.observe(requireActivity(), nameObserver)  
        return view  
    }  
}
```

✧ Instanciamos el ViewModel **Línea 2**, Creamos el manejador con la funcionalidad de asignar al TextView el texto que está siendo observado **Línea 9**. **Línea 11** se asigna al delegado observe el manejador creado con anterioridad.

🎓 Vamos a codificar otra vez este caso, pero con un pequeño cambio que nos valdrá para presentar un elemento nuevo **ListFragment**, para ello el **FragmentPrimario** pasaremos a llamarlo **MyListFragment** que heredará **ListFragment** y que debido a esto no necesita tener asociado un xml. El resultado será parecido al siguiente:



En la única clase que deberemos realizar cambios es en la nueva clase que ahora se verá así **MyListFragment.kt**:

```
1 class MyListFragment: ListFragment() {
2     private val model:ItemViewModel by activityViewModels()
    private val valores =
        arrayOf<String?>("item1", "item2", "item3", "item4",
                        "item5", "item6", "item7", "item8")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
8        listAdapter = ArrayAdapter<Any?>(requireActivity(),
            android.R.layout.simple_list_item_1, valores)
    }
11    override fun onListItemClick(l: ListView, v: View,
        position: Int, id: Long) {
        super.onListItemClick(l, v, position, id)
14        valores[position]?.let { model.setItem(it) }
15    }
}
```

✧ **Línea 1** en este caso vemos que el fragment hereda de **ListFragment** en vez de **Fragment**, esto permite que el fragment esté formado por una lista de elementos y que no se tenga que crear ninguna vista específica para el. Para asociar los elementos a la lista, deberemos crear un **ArrayAdapter** indicando los valores y el layout de salida (predefinido en los recursos de Android) **Línea 8** (Este elemento se verá más extensamente en temas posteriores). **Línea de 11 a 15**, esta sobrecarga del método **onListItemClick** está asociada a la pulsación de cualquier elemento de la lista, entrando como parámetros la vista y la posición del elemento pulsado. **Línea 2** se crea una propiedad del **ViewModel** creado con anterioridad, utilizando el delegado **activityViewModels** para obtener una referencia al ViewModel en el ámbito de su actividad y que se pueda compartir entre las distintas vistas o fragments, mientras el ciclo de vida lo permita

🎓 Otro caso distinto, podemos tenerlo cuando **el fragment se comunica a través de la activity**, mediante el ViewModel. Vamos a suponer, el caso más común en el que una activity cargará una lista y al pulsar un elemento de esta se carga el otro fragmento. El layout del **fragment_secundario.xml** y **MyListFragment.kt** tendrán el mismo código que para el anterior ejemplo.

El layout de la actividad principal ahora solo tendrá un contenedor,
activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView xmlns:android="http://schemas.a
    android:id="@+id/contenedor_fragment"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
</androidx.fragment.app.FragmentContainerView>
```

La actividad principal con la carga del fragment con la lista y con el observador sobre el ViewModel, quedará así **MainActivity.kt**:

```
class MainActivity : AppCompatActivity() {
    2    private val model: ItemViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val fM: FragmentManager = supportFragmentManager
        var fT: FragmentTransaction = fM.beginTransaction()
        fT.add(R.id.contenedor_fragment, MyListFragment())
        fT.commit()
    10    val nameObserver = Observer<String>{cadena ->
        val bundle=Bundle()
        bundle.putString("DATO",cadena)
        var fragmentSecundario=FragmentSecundario()
        fragmentSecundario.arguments=bundle
        fT=fM.beginTransaction()
        fT.add(R.id.contenedor_fragment,fragmentSecundario )
        fT.commit()
        fT.addToBackStack(null)
    19    }
    20    model.getItem.observe(this, nameObserver)
    }
}
```


📌 **Línea 2**, instanciamos el ViewModel teniendo en cuenta que para la actividad se asocia el delegado **viewModels** . En el bloque desde la **Línea 10 hasta la 19**, construimos un delegado con el observador que se encargará de cargar el FragmentSecundario con la información pasada en un Bundle. **Línea 20**, se pone en observación el ViewModel con el delegado construido anteriormente.

En el **FragmentSecundario.kt** se deberá recuperar el bundle para poder modificar el TextView con el dato del elemento pulsado.

```
class FragmentSecundario:Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        val bundle=arguments
        val view: View = inflater.inflate(R.layout.fragment_secundario,
                                           container, false)


        view.findViewById<TextView>(R.id.texto).text=
            bundle?.getString("DATO")

        return view
    }
}
```

 Con la información que se pasa en los tres casos de estudio, reconstruye y prueba estos casos añadiendo el código que falte.

Comunicación mediante Interfaces

La [implementación de una interface](#), es la otra manera correcta para pasar información y controlar algún evento. Se trata de crear una interface en el fragmento y exigir a la activity que la implemente. De esta manera cuando el fragmento reciba un evento también lo hará la activity, que se encargara de recibir los datos de ese evento y compartirlos con otros fragmentos.

 Para ello vamos a volver a implementar el último ejemplo. Es decir la aplicación que comienza con el fragment de la lista, y que carga otro que muestra el texto del elemento pulsado. En este caso solamente tendremos que modificar el código de la clase principal y del fragmente de la lista, es resto quedará igual.

Primero crearemos la interface con el método que necesitemos,

PasoCadenaInterface.kt

```
interface PasoCadenaInterface {
    fun informacionCadena(dato:String)
}
```

La actividad principal **MainActivity.kt**:

```

0  class MainActivity : AppCompatActivity(), PasoCadenaInterface {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val fM: FragmentManager = supportFragmentManager
        var fT: FragmentTransaction = fM.beginTransaction()
        fT.add(R.id.contenedor_fragment, MyListFragment())
        fT.commit()
    }

10  override fun informacionCadena(dato: String) {
        val bundle=Bundle()
        bundle.putString("DATO",dato)
        var fragmentSecundario=FragmentSecundario()
        fragmentSecundario.arguments=bundle
        val fT=supportFragmentManager.beginTransaction()
        fT.add(R.id.contenedor_fragment,fragmentSecundario )
        fT.commit()
        fT.addToBackStack(null)

19  }
    }

```

🔗 **Línea 0** obligamos a que la actividad implemente la interface. **Línea 10 a 19**, método de la interface al que le llega el dato que mandaremos al fragment secundario.

Fragment con la lista, **MyListFragment.kt**:

```

class MyListFragment: ListFragment() {
2  lateinit var pasoCadenaInterface:PasoCadenaInterface
    private val valores =
        arrayOf<String?>("item1", "item2", "item3", "item4",
            "item5", "item6", "item7", "item8")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        listAdapter = ArrayAdapter<Any?>(requireActivity(),
            android.R.layout.simple_list_item_1, valores)
    }
    override fun onListItemClick(l: ListView, v: View,
        position: Int, id: Long) {
        super.onListItemClick(l, v, position, id)

14  valores[position]?
            .let { pasoCadenaInterface.informacionCadena(it) }
    }

17  override fun onAttach(context: Context) {
        super.onAttach(context)
        pasoCadenaInterface=context as PasoCadenaInterface

20  }
}

```

✂ **Línea 2** se crea una instancia de la interface. **Línea desde 17 a 20** se anula el método **onAttach** que es invocado en el momento que el fragment se enlaza en la actividad, en este método asignamos el contexto de entrada a la instancia de la interface creada. **Línea 14** con el objeto de la interface llamamos a su método pasando el valor del elemento pulsado.

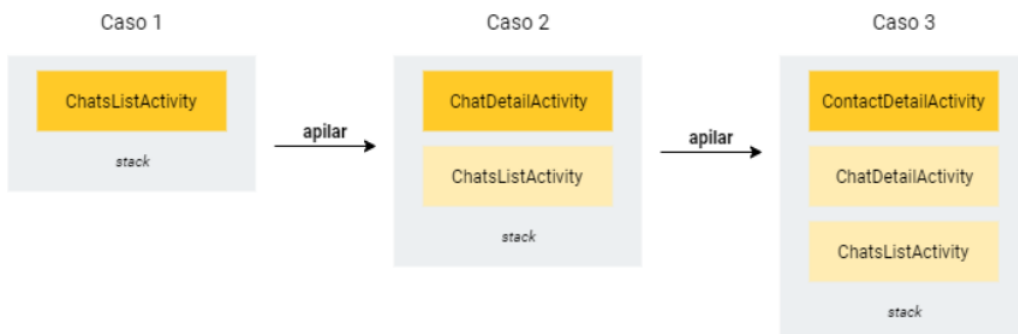
✍ **Reconstruye el ejemplo anterior y prueba su funcionamiento**

Pila de actividades

Tasks y BackStacks

Una **tarea** es una colección de metadatos e información entorno a un pila de actividades. Cuando iniciamos una app el sistema busca si esa tarea ya ha sido iniciada previamente y en ese caso la reanuda y regresar exactamente al punto donde estaba. Si no existe la tarea la inicia, lanzando la actividad principal como base en la **pila de tareas**.

Cada vez que se inicia una actividad o se realiza una transacción de fragmento añadiendo la transición a la pila **addToBackStack()**, se añade esa actividad/fragmento a la tarea lo que hace que la actividad que queda por debajo se pause o se detenga si la nueva actividad oculta por completo la anterior.



Las actividades **ámbar oscuro** representarán la actividad en primer plano

El botón de retroceso está directamente asociado a la pila de actividades. Cuando se presiona este botón, se elimina un fragmento o actividad de la pila se restaura el inmediatamente inferior en esta. La repetición del **BackPressed** eliminará todas las actividades hasta que no quede ninguna y se destruya la aplicación (tarea).



Gestionar Tareas Programáticamente

Existen casos específicos donde requeriremos manipular la forma en que una actividad es apilada o retirada en la back stack de una tarea. El framework nos provee atributos del Manifest para dicho fin, con `launchMode` podremos especificar el modo de lanzamiento de una actividad, sus posibles valores son los siguientes: `standard`, `singleTop`, `singleTask` y `singleInstance`, para más [información](#).

FragmentManager

`FragmentManager` es la clase responsable de realizar acciones en los fragmentos de tu app, como agregarlos, quitarlos o reemplazarlos, así como trabajar con la pila de actividades. Aunque en el punto siguiente se va a explicar la biblioteca **Jetpack Navigation** que facilita el trabajo, es importante conocer algo sobre esta clase.

Para acceder a `FragmentManager` con Kotlin, lo haremos a través de `supportFragmentManager`. `FragmentManager` agrupa los cambios realizados en lo que se llaman **transacciones** tratadas con la clase `FragmentTransaction`. Por lo que es común que se agrupen varias acciones en una sola transacción (agregar o reemplazar varios fragmentos en una sola transacción). Por ejemplo: *esta agrupación puede ser útil cuando tiene varios fragmentos de hermanos mostrados en la misma pantalla, como con vistas divididas*.

Para obtener una instancia de `FragmentTransaction` utilizamos `beginTransaction`

```
val fragmentTransaction = supportFragmentManager.beginTransaction()
```

Una instancia de **`FragmentTransaccion`** permite realizar varias operaciones como agregar, reemplazar, eliminar u ocultar fragmentos. Algunos de sus métodos principales son:

- `add()` recibe el ID del contenedor del fragmento (debe ser de tipo `FragmentContainerView`) y el fragmento que desea agregar. El fragmento agregado se traslada al estado de RESUMED a espera del próximo commit.

- **remove()** permite eliminar un fragmento que se debe recuperar del administrador de fragmentos a través de **findFragmentById()** o **findFragmentByTag()**. Si la vista del fragmento se agregó previamente a un contenedor, se eliminará tras la llamada a este método. El fragmento eliminado se traslada al estado de DESTROYED.
- **replace()** reemplaza un fragmento existente en un contenedor con una instancia de uno nuevo que se proporcione como argumento. Es equivalente a llamar remove y add en un mismo contenedor.
- **show()** y **hide()** son métodos que permiten mostrar y ocultar la vista de los fragmentos que se han agregado a un contenedor. Estos métodos establecen la visibilidad de las vistas del fragmento sin afectar el ciclo de vida del fragmento.
- **addToBackStack()** De forma predeterminada, los cambios realizados en a FragmentTransaction no se agregan a la pila de actividades, hay que usar este método para hacerlo (se puede añadir un tag identificativo)
- **commit** programa una confirmación de las transacciones pendientes de forma asíncrona, que se realizará la próxima vez que ese hilo esté listo.

FragmentManager también tiene varios métodos de utilidad, entre los que se encuentran:

- **popBackStack** desapila fragmentos de la pila de retroceso, simulando el uso del comando Atrás por parte del usuario. También se puede seleccionar un fragment en concreto para desapilar.
- Para obtener fragmentos que ya existen en la actividad con **findFragmentById()** (para fragmentos que proporcionan una IU en el diseño de la actividad) o **findFragmentByTag()** (para fragmentos con o sin IU).

```
fT.add(R.id.contenedor, Fragment2(), "f2")
val f=fM.findFragmentByTag("f2")
```

Jetpack Navigation

El componente [Navigation](#) de Android Jetpack, permite implementar la navegación de los elementos de la aplicación de forma más coherente y sencilla, desde simples clics de botones hasta patrones más complejos, como las barras de apps y los paneles laterales de navegación. Es importante seguir el sistema establecido de [principios](#), para que el usuario pueda entender correctamente el funcionamiento de la app. Podemos encontrar tres elementos fundamentales:

- **Gráfico de navegación:** Es un recurso XML que contiene toda la información relacionada con la navegación.
- **NavHost:** Es un contenedor vacío que muestra los destinos de tu gráfico de navegación. Por defecto está implementado el `NavHostFragment` para fragments y que será el que usemos.
- **NavController:** Es el objeto que administra la navegación de la app dentro de un NavHost.

👉 Será importante decidir como queremos pasar de unos fragments a otros, **porque lo que no se debe hacer es mezclar el sistema de Navigation con el que usamos con FragmentManager.**

Para usar estas características deberemos incluir la librería:

```
implementation 'androidx.navigation:navigation-ui-ktx:2.3.5'
implementation 'androidx.navigation:navigation-fragment-ktx:2.3.5'
```

NavHost

Para comenzar lo primero que debemos hacer es crear el `NavHost`, para ello usaremos el contenedor donde vamos a insertar los fragments. Este elemento debe ser de tipo `FragmentContainerView` y habrá que añadir los siguientes atributos:


```

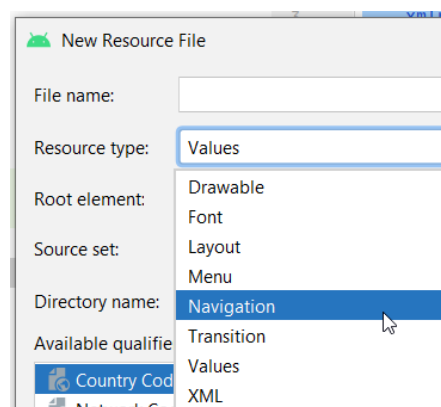
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    tools:context=".MainActivity">
9      <androidx.fragment.app.FragmentContainerView
10         android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
13         app:defaultNavHost="true"
14         app:navGraph="@navigation/navegacion_fragments"
15         android:id="@+id/contenedor"/>
    </LinearLayout>

```

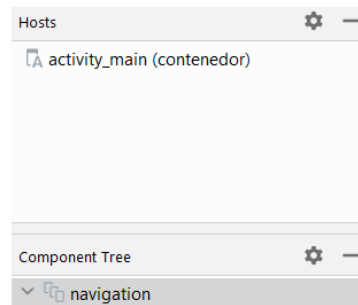
✍ En la actividad principal, por ejemplo, podemos crear el espacio para insertar los fragmentes, en este caso desde **Línea 9 a 15**. Los atributos nuevos y que se tienen que poner para el funcionamiento correcto son: el nombre asignado a la clase **Línea 10**, se usa el **NavHostFragment** que ya está implementado; **Línea 13**, el atributo **defaultNavHost** garantiza que tu **NavHostFragment** intercepte el botón Atrás del sistema; En el **navGraph** se indica el nombre del gráfico de navegación. Elemento que vamos a explicar ahora **Línea 14**.

NavGraph

Para crear el **NavGraph** debemos crear un nuevo recurso de tipo navigation **res->new->android resource file**.



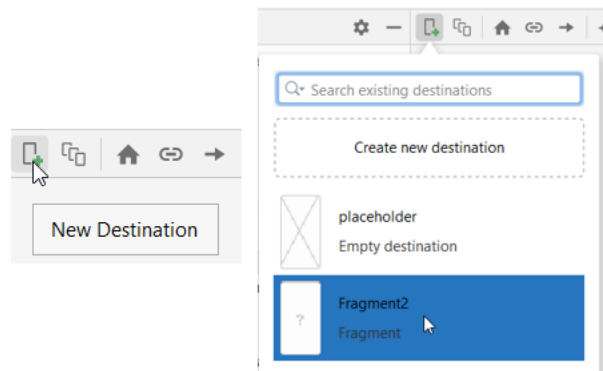
Al hacer esto se creará una nueva carpeta navigation con el recurso dentro de ella. El archivo **.xml** se podrá trabajar de forma gráfica o mediante código, aunque en este caso es más sencillo utilizar la parte visual. Podremos añadir los elementos por los que queremos que navegue la app y crear las acciones entre ellos, todo es muy visual y sencillo.



Podemos ver que en la parte del Host se ha añadido el elemento **NavHost** al crear el contenedor de fragmentos como hemos explicado anteriormente (enlazando, *por nombre*, con el recurso en la línea

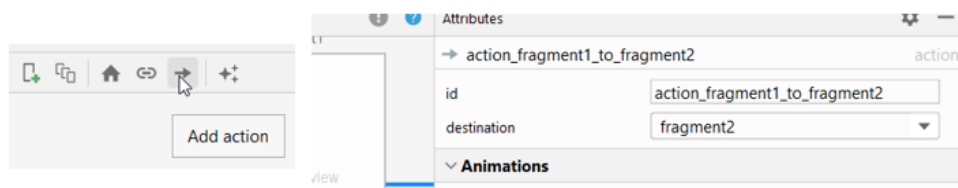
`app:navGraph="@navigation/navegacion_fragments"`). Si no se crea correctamente el Host no funcionarán las acciones.

Para añadir **elementos** pulsaremos el símbolo de **+** y seleccionaremos de la lista de elementos navegables que tengamos en la app.

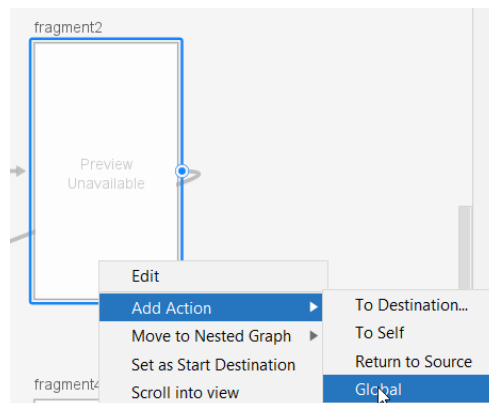


Una vez agregados los elementos, podemos añadir las **acciones** entre ellos.

Simplemente seleccionando elemento e indicando, de entre los elementos añadidos en el gráfico, el destino donde nos llevará esa acción, el id de la acción se crea automáticamente y se usará posteriormente en el código.



Hay un tipo de **acciones globales** que permiten llegar a un destino desde cualquier lugar. Se muestran como una `->` corta enlazada solamente por el destino, y que se crean mediante el menú contextual del destino seleccionado.



✎ Cuando estamos creando el gráfico de navegación podemos optar por perfeccionar las acciones para que al ocurrir estas y saltar a un lugar concreto, se desapilen los elementos intermedios innecesarios. Esto evitará que al pulsar el `BackPressed`, se pase por cada uno de los fragments apilados en las distintas acciones. Para ello solo tenemos que ir a la propiedad **Pop Behavior** de la acción deseada, e indicar en **popUpTo** hasta que elemento se quiere desapilar, con **popUpToInclusive** a `true`, indicamos que también se desapile ese elemento.

▼ Pop Behavior	
popUpTo	fragment1
popUpToInclusive	<input checked="" type="checkbox"/> true

NavController

Este elemento ya lo usaremos de forma programática y es el que nos permite ejecutar las acciones que hemos creado con el `NavGraph`. Cada `NavHost` tiene su propio `NavController` correspondiente. Como se explica en la documentación oficial, hay varias formas de hacerlo. Por ejemplo **desde un fragmento** podemos hacerlo de la siguiente manera:

```
val navController= NavHostFragment.findNavController(this)
if (navController.currentDestination?.id == R.id.fragment1)
    navController.navigate(R.id.action_fragment1_to_fragment2)
```

`NavController` también permite [pasar datos entre fragment mediante bundle](#), el proceso es similar al usado en otras explicaciones.

✎ Primero usamos una de las maneras existentes para referenciar al `NavController` usando el `NavHostFragment` que tenemos por defecto. Antes de

navegar a una acción, es interesante comprobar si estamos en el fragmento que tiene creada esa acción, de esta forma evitaremos excepciones. Y por último, navegaremos al fragmento deseado usando el **id** de la acción registrada con anterioridad. *Lógicamente, este código lo tendremos que añadir al botón u otro elemento que queramos que ejecute la navegación.*

👉 Vamos a mostrar el código de una actividad con tres fragmentos de forma que: del primero se pueda navegar al segundo, del segundo al tercero y del tercero al primero siempre que se pulse el **botón aceptar**, mientras que si se pulsa el **botón cancelar** se navegará siempre al primero (la navegación al primero se realizará con una acción global). Para ello se tendrán que construir los *.xml* de los Fragments, todos serán iguales, cambiando el color y el texto. Código **fragment1.xml** con dos botones *cancelar* y *aceptar*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:background="#2196F3"
    android:gravity="center"
    android:layout_height="match_parent">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="Fragment1"/>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">
        <com.google.android.material.button.MaterialButton
            android:id="@+id/aceptar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.5"
            android:backgroundTint="#2196F3"
            android:text="Aceptar" />
        <com.google.android.material.button.MaterialButton
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.5"
            android:backgroundTint="#2196F3"
            android:text="Cancelar"
            android:id="@+id/cancelar"/>
    </LinearLayout>
</LinearLayout>
```

El *.xml* de la actividad principal será como el que hemos puesto al principio de la explicación de **Navigation** . Mientras que **MainActivity.kt** tendrá el siguiente código:

```
class MainActivity : AppCompatActivity() {
    lateinit var navHostFragment: NavHostFragment
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        6      navHostFragment = supportFragmentManager.
                findFragmentById(R.id.contenedor) as NavHostFragment
    }
    fun cancelar()
    {
        11      val navController = navHostFragment.navController
        12      navController.navigate(R.id.action_global_fragment1)
    }
}
```

✂ Se ha creado un método **cancelar** que permite ejecutar la acción global, se ha hecho de esta forma para no repetir código en los fragments y de paso conocer como se ejecuta una acción desde la main. **Línea 6**, creamos la instancia del **NavHostFragment** . **Línea 11 y 12** instanciamos el **NavController** y ejecutamos la acción global de navegación al **Fragment1**.

El **Fragment1.kt** y por ende, el resto de Fragments. Con el botón **aceptar** que realizará la acción de moverse al siguiente fragment (en **Fragment3** será la global de moverse al **fragment1**) y el botón **cancelar** que lanzará la acción global.

```

class Fragment1: Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        val view=inflater.inflate(R.layout.fragment1,container,false)
        view.findViewById<MaterialButton>(R.id.aceptar).
            setOnClickListener { aceptar() }
        view.findViewById<MaterialButton>(R.id.cancelar).
            setOnClickListener{(requireActivity() as MainActivity)
                .cancelar()}

        return view
    }
    fun aceptar()
    {
        val navController= NavHostFragment.findNavController(this)
        if (navController.currentDestination?.id == R.id.fragment1)
            navController.navigate(R.id.action_fragment1_to_fragment2)
    }
}

```

✎ **Línea 9** acción del botón aceptar que llama al método aceptar que ejecuta la acción de navegación. **Línea 10** acción del botón cancelar que llamará al método cancelar de MainActivity, realizando un casteo del contexto.

✎ **A partir del código expuesto en el ejemplo anterior, crea el ejercicio completo. Recuerda que debes crear las acciones de navegación creando un recurso de tipo navigation.**

Permisos con Fragments

Para realizar la petición de permisos al usuario en las clases que heredan de Fragment, se codifica de forma distinta a como lo explicamos para las activities, aunque el concepto es el mismo. Registramos la actividad para esta tarea mediante el método, ya utilizado para los intents, `registerForActivityResult` con un contrato específico para pedir permisos `ActivityResultContracts.RequestPermission`. Luego usaremos el `ActivityResultLauncher` creado, para lanzar la tarea.

```

...
lateinit var registerPermisosStorage:ActivityResultLauncher<String>
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    registerPermisosStorage=
        registerForActivityResult(ActivityResultContracts.RequestPermission())
        {
            if(it ==true) tomarGaleria()
        }
}
...
binding.imagen.setOnLongClickListener {
    registerPermisosStorage.launch(Manifest.permission.READ_EXTERNAL_STORAGE)
    true
}
...

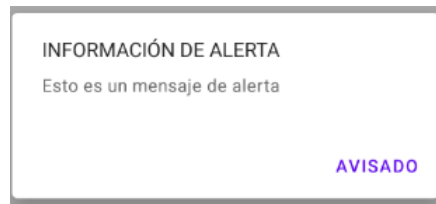
```

DialogFragment

Como hemos visto al principio del tema, la clase Fragment tiene varias clases hijas que nos permiten realizar tareas concretas, una de ellas es la clase [DialogFragment](#). Al heredar de ella podremos crear distintos tipos de diálogos y es la manera aconsejada para hacerlo. Como hemos dicho, para utilizar este tipo de diálogos deberemos crearnos una clase derivada de DialogFragment e implementar en ella el tipo de dialogo que queramos (como esta parte ya la conocemos de temas anteriores, solo vamos a comentar un par de AlertDialog diferentes). Como en este caso los diálogos están implementados con fragments para mostrarlos necesitaremos usar un **FragmentManager**, no la típica llamada al método show. Vamos a pasar a ver un ejemplo:

Diálogo de Alerta

Este tipo de diálogo se limita a mostrar un mensaje sencillo al usuario, y un único botón de OK para confirmar su lectura. Veamos un ejemplo:



```
class DialogoAlerta:DialogFragment(){
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        super.onCreateDialog(savedInstanceState)
        val builder=MaterialAlertDialogBuilder(requireActivity())
        builder.setMessage("Esto es un mensaje de alerta")
            .setTitle("INFORMACIÓN DE ALERTA")
            .setPositiveButton("Avisado",
                DialogInterface.OnClickListener{dialogo,id->dialogo.cancel()})
        return builder.create()
    }
}
```

✎ Como se puede ver en el código, la forma de crear el dialogo es igual que la vista en el bloque6, pero al introducirlo en un fragment lo hacemos más reutilizable.

```
val dialogoAlerta=DialogoAlerta()
dialogoAlerta.show(supportFragmentManager,"DialogoAlerta")
```

✎ Para lanzar este diálogo, por ejemplo, desde nuestra actividad principal, creamos un objeto de tipo DialogoAlerta y lo mostramos mediante el método show(), pasando una referencia al Fragment Manager mediante una llamada a la propiedad **supportFragmentManager** , y una etiqueta identificativa del diálogo. Este segundo argumento, debe ser un nombre de etiqueta único y que el sistema usará para guardar y restaurar el estado del fragmento cuando sea necesario. La etiqueta también permite obtener un controlador para el fragmento llamando a **findFragmentByTag()** .

Si por algún motivo se necesita pasar información al dialogo, podremos crear un constructor al que le llegue la información mediante Bundle o de otro modo. En el siguiente código se puede ver un posible caso:


```

class DialogoAlerta(bundle: Bundle):DialogFragment(){
    lateinit var bundle: Bundle
    init{
        this.bundle=bundle
    }
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        super.onCreateDialog(savedInstanceState)
        val builder=MaterialAlertDialogBuilder(requireActivity())
        builder.setMessage(bundle.getString("DATO"))
            .setTitle("INFORMACIÓN")
            .setPositiveButton("Aceptar",
                DialogInterface.OnClickListener{dialogo,id->dialogo.cancel()})
        return builder.create()
    }
}

```

Y en la creación del dialog fragment:

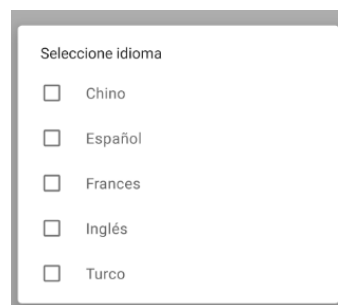
```

val bundle=Bundle()
    bundle.putString("DATO","AVISO DE PROXIMA CONSULTA MÉDICA")
    val dialogoAlerta=DialogoAlerta(bundle)
    dialogoAlerta.show(supportFragmentManager,"DialogoAlerta")

```

Diálogo de Selección Múltiple

Este tipo de dialogo nos permite implementar una selección múltiple, como podemos ver en la siguiente imagen:



```

class DialogoSeleccionMultiple:DialogFragment() {
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        super.onCreateDialog(savedInstanceState)
        val builder= MaterialAlertDialogBuilder(requireActivity())
        val datos= arrayOf("Chino","Español","Frances","Inglés","Turco")
        builder.setTitle("Seleccione idioma")
            .setMultiChoiceItems(datos,null,
                DialogInterface.OnMultiChoiceClickListener{dialogo,item,isChecked->
                    Toast.makeText(requireActivity(),"Se ha seleccionado "
                        +datos[item],Toast.LENGTH_SHORT).show()
                })
        return builder.create()
    }
}

```

Cada vez que se selecciona un elemento de la lista se ejecutará el escuchador `OnMultiChoiceClickListener` , llegando a este el ítem pulsado y si esta *checked* o *unchecked*.

Ejercicio Propuesto Dialogo Personalizado

Vistas Deslizantes y Tabs

ViewPager

Las [vistas deslizantes o paginación horizontal](#), permiten navegar entre pantallas relacionadas. Por ejemplo, entre pestañas, con un gesto horizontal del dedo o un deslizamiento. Para crear, a día de hoy, vistas deslizantes se utiliza el elemento `ViewPager2` .

Vamos a retomar el ejemplo de **FragmentV3ViewModel** y añadir el código necesario para conseguir el efecto de desplazamiento entre fragments. Los pasos son los siguientes:

1. El layout de la actividad donde se van a añadir los fragments, deberá de incluir un `ViewPager2` . Si la pantalla muestra los fragments completos, este elemento puede ser la raíz. Por ejemplo, el `.xml` del **MainActivity** podría ser:

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.viewpager2.widget.ViewPager2 xmlns:android="http://schemas.android
/apk/res/android"
    android:id="@+id/ViewPager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

```

2. Se deberá crear una clase que manejará la carga de los fragments con el desplazamiento. Esta clase deberá heredar de la clase abstracta `FragmentStateAdapter`, por lo que obligará a implementar el método `createFragment()` a fin de aprovisionar instancias de Fragments a la paginación, y el método `getItemCount()`, que debe devolver el número exacto de fragments que se paginarán.

```
class FragmentPagerAdapter(fragment: FragmentActivity):
    FragmentStateAdapter(fragment)
{
    override fun getItemCount()=3
    override fun createFragment(position: Int): Fragment {
        return when (position) {
            0 -> MyListFragment()
            1 -> FragmentSecundario()
            2 -> FragmentTercero()
            else -> MyListFragment()
        }
    }
}
```

👉 Lógicamente las clases con los fragments y sus layouts relacionados deben existir, en este caso se ha duplicado `FragmentSecundario` del ejemplo anterior y se ha renombrado a `FragmentTercero`.

3. Por último se deberá conectar el `FragmentStateAdapter` a los objetos `ViewPager2`.

```
class MainActivity : FragmentActivity()
{
    lateinit var viewPager:ViewPager2
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        7 viewPager = findViewById(R.id.ViewPager)
        8 val pagerAdapter = FragmentPagerAdapter(this)
        9 viewPager.adapter = pagerAdapter
    }
    11 override fun onBackPressed() {
        if (viewPager.currentItem == 0) super.onBackPressed()
        13 else viewPager.currentItem = viewPager.currentItem - 1 }
}
```

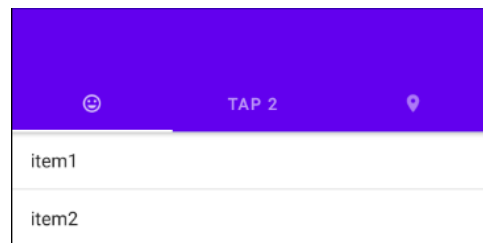
🔗 En este caso nos ha convenido que `MainActivity` herede de `FragmentActivity`, para que de esta forma no nos de problemas la llamada al constructor de `FragmentStateAdapter`. **Línea 7** instanciamos el objeto

`ViewPager2` de `activity_main.xml`. **Línea 8** instanciamos un objeto de la clase que hereda de `FragmentStateAdapter` . **Línea 9** relacionamos ambos objetos para hacer la conexión. **Bloque 11 a 13**, anulamos el `BackPressed` para que realice el comportamiento correcto de cambio de fragments. Como se puede ver se hace mediante la propiedad `currentItem` decrementándola en 1.

✍ **Reconstruye el ejemplo explicado con anterioridad para probar el funcionamiento del ViewPager. Podrás comprobar que el paso de información con ViewModel sigue funcionando correctamente**

TabLayout

Un objeto `TabLayout` proporciona una forma crear pestañas horizontales. Cuando se usa junto con un `ViewPager2` nos permite el deslizamiento al mismo tiempo que se marca la pestaña activa.



Las pestañas se pueden insertar en la `ToolBar` (la manera más común) o independientes a esta. Nosotros vamos a seguir un ejemplo que añade las Tabs a la `ToolBar` y para ello retomaremos el anterior ejercicio, pero ahora a la `activity_main.xml` le añadiremos una `MaterialToolBar` como ya vimos en el Bloque 4 y dentro de esta meteremos el `TabLayout` .

```

<?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">
    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.MaterialComponents.Dark.ActionBar">
        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"/>
        <com.google.android.material.tabs.TabLayout
            android:id="@+id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            style="@style/Widget.MaterialComponents.TabLayout.PrimarySurface"/>
        </com.google.android.material.appbar.AppBarLayout>
    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/viewpager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    </LinearLayout>

```

Las nuevas líneas a añadir en la actividad principal para referenciar al TabLayout creado y para crear un objeto de tipo [TabLayoutMediator](#) encargado de relacionar el tabLayout con el ViewPager.

```

val tabLayout = findViewById<TabLayout>(R.id.tabs)
TabLayoutMediator(tabLayout, viewPager) { tab, position ->
    when (position) {
        0 -> tab.setIcon(R.drawable.ic_emoticon)
        1 -> tab.text="Tap 2"
        2 -> tab.setIcon(R.drawable.ic_place)
    }
}.attach()

```

✎ Al crear una instancia de la clase **TabLayoutMediator** se tiene que implementar la interface **TabLayoutMediator.TabConfigurationStrategy** para configurar la pestaña de la página en la posición especificada (en esta interface se establecerá el texto de la pestaña y cualquier estilo de las pestañas que se necesite).

TabLayoutMediator sincronizará la posición de ViewPager2 con la pestaña seleccionada cuando se seleccione una pestaña, y la posición de desplazamiento de TabLayout cuando el usuario arrastre ViewPager2.

Además **TabLayoutMediator** escucha diferentes elementos, para controlar las distintas situaciones:

- **OnPageChangeListener** de ViewPager2 para ajustar la pestaña cuando ViewPager2 se mueva.
- **OnTabSelectedListener** de TabLayout para ajustar VP2 cuando se mueve la pestaña.
- **AdapterDataObserver** de RecyclerView para recrear el contenido de la pestaña cuando cambia el conjunto de datos.

Un ejemplo del escuchador del ViewPager2 sería:

```
tabLayout.addTabSelectedListener (object:TabLayout.OnTabSelectedListener{
    override fun onTabSelected(tab: TabLayout.Tab?) {
        TODO("Not yet implemented")
    }
    override fun onTabUnselected(tab: TabLayout.Tab?) {
        TODO("Not yet implemented")
    }
    override fun onTabReselected(tab: TabLayout.Tab?) {
        TODO("Not yet implemented")
    })
```

 **Construye el ejemplo y prueba su funcionamiento**