```
FileError: resource 'https://fonts.googleapis.com/css?family=Catamaran' g
  Error: getaddrinfo ENOTFOUND fonts.googleapis.com
 in input on line 13, column 1:
12
13 @import url(https://fonts.googleapis.com/css?family=Catamaran);
14
```

Apuntes

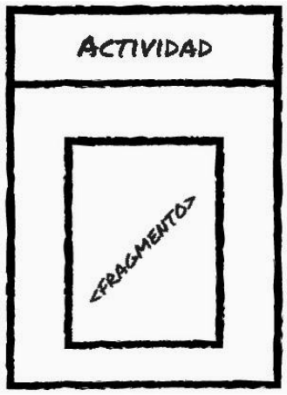
Descargar estos apuntes

Tema 7. Fragments

Índice

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.



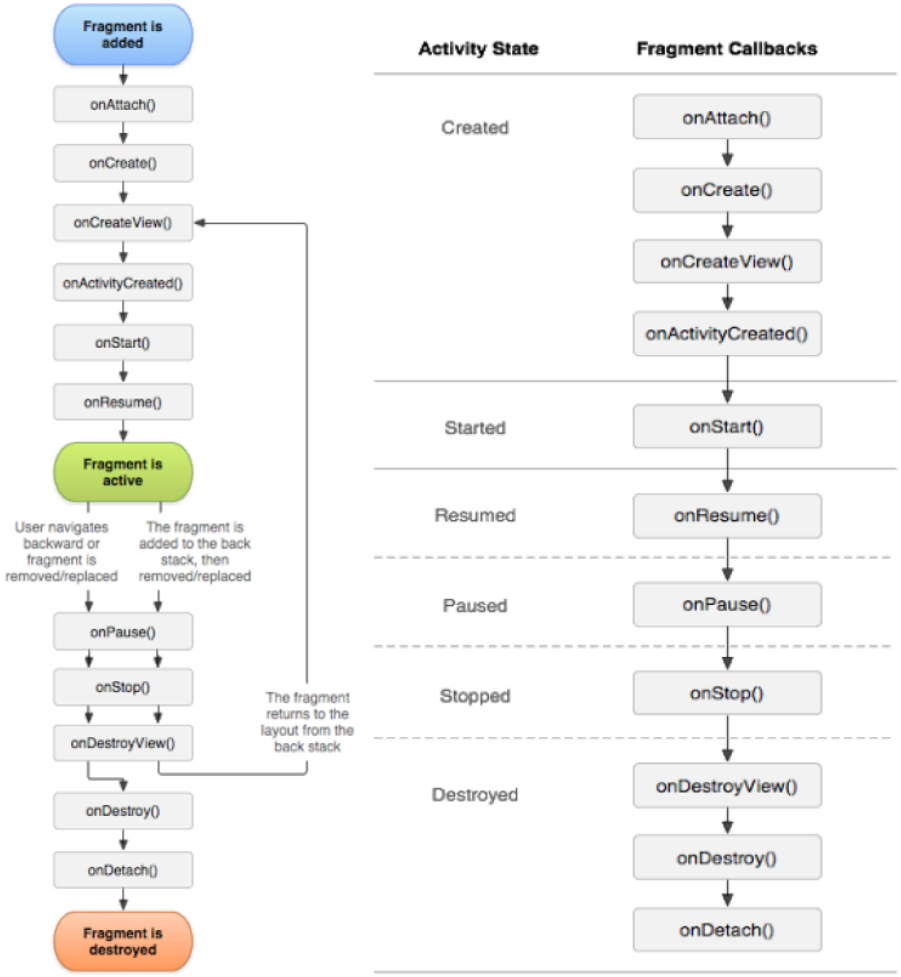
Estas nuevas entidades permiten reusar código y ahorrar tiempo de diseño a la hora de desarrollar una aplicación. Los fragmentos facilitan el despliegue de tus aplicaciones en cualquier tipo de tamaño de pantalla y orientación.
Otra ventaja de usarlos es que permiten crear diseños de interfaces de usuario 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 `FragmentTransaction` y a través de su método `add()` Q,

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 framentManager=supportFragmentManager
8       val fragmentTransaction=framentManager.beginTransaction()
        val fragmentDos=FragmentDos()
10      fragmentTransaction.add(R.id.fragment_container,fragmentDos)
11      fragmentTransaction.commit()
    }
}
```

> 📌En este caso nuestra activity carga un layout con un View `fragment` para el fragmento estático y un `FragmeLayout` 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 `getSupportFragmentManaget()` , **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.

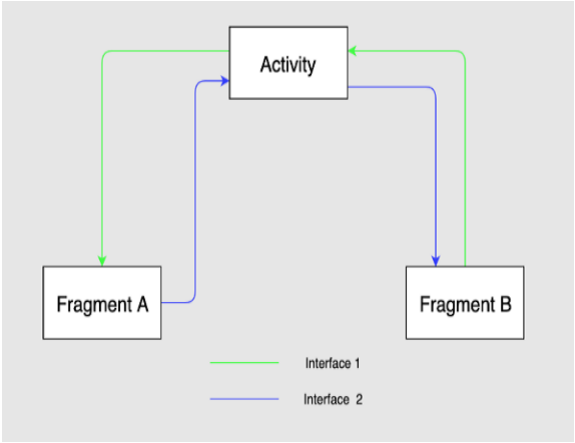
🖊Prueba el anterior ejemplo FragmentV2: botón que te vaya cargando uno u otro fagment y al mismo tiempo que se acumulen en la pila. Probar que al pulsar el botón de retroceso, se pasa por cada uno de los fragments que se han cargado anteriormente.

Comunicar Fragments y Activitys

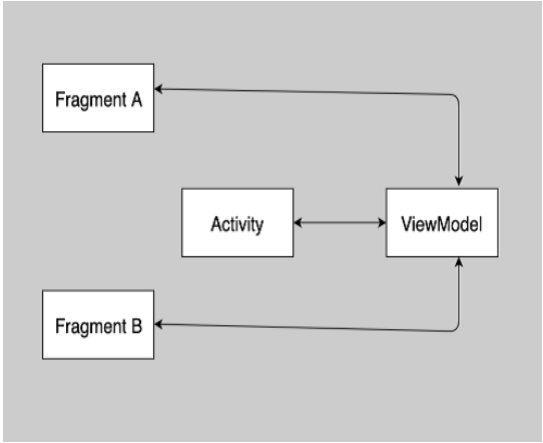
Cuando hablamos sobre la comunicación de fragmentos, debemos tener en cuenta las siguientes premisas:

- 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 Interface



Comunicación mediante 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 selectedItem: LiveData<String> get() = liveData
        fun selectItem(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 `LiveData` , 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 LiveData, 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). LiveData solo notifica a los observadores activos sobre las actualizaciones.

🎓 Vamos a suponer un ejemplo en el que, usando el ViewModel del ejemplo anterior, en una activity se cargan dos fragments a la vez y se realiza **la comunicación entre los dos fragments**. Uno tendrá una lista y en el otro se mostrará el elemento pulsado de la lista. Para ello deberemos crear dos clase Fragments, una que será el detalle y que solo tendrá un texto y una clase **ListaFragment** que heredará de ListFragment y que debido a esto no necesita tener asociado un xml. El resultado será parecido al siguiente:

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="TEXTO"
        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, MyListFragment())
        fT.add(R.id.fragment_dos,FragmentSecundario>())
        fT.commit() }
}
```

El Fragment con la lista podría ser como el que sigue, **MyListFragment.kt**:

```
1  class MyListFragment: ListFragment() {
2      private val model:ItemViewModel by activityViewModels()
       private val valores =
                  arrayOf<String?>("item1", "item2", "item3", "item4", "item5
       override fun onCreate(savedInstanceState: Bundle?) {
           super.onCreate(savedInstanceState)
7          listAdapter = ArrayAdapter<Any?>(requireActivity(), android.R.lay
       }
9      override fun onListItemClick(l: ListView, v: View, position: Int, id:
           super.onListItemClick(l, v, position, id)
11         valores[position]?.let { model.selectItem(it) }
12     }
   }
```

📌**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 7** (Este elemento se verá más extensamente en temas posteriores). **Línea de 9 a 11**, 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. 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 11, cuando se pulse un elemento de la lista se modificará el valor del LiveData con la información de esa posición del array (siempre que no sea nula *let{}*)

🎓 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.selectedItem.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?,
        saveInstanceState: Bundle?
    ): View? {
        val bundle=arguments
        val view: View = inflater.inflate(R.layout.>fragment_secundario, cont
        view.findViewById<TextView>(R.id.texto).text=bundle?.getString>("DATO
        return view
    }
}
```

Con la información que se pasa en los dos ejemplos, reconstruye y prueba ambos 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
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            listAdapter = ArrayAdapter<Any?>(requireActivity(), android.R.lay
        }
        override fun onListItemClick(l: ListView, v: View, position: Int, id:
            super.onListItemClick(l, v, position, id)
11          valores[position]?.let { pasoCadenaInterface.informacionCadena(it
        }
13      override fun onAttach(context: Context) {
            super.onAttach(context)
            pasoCadenaInterface=context as PasoCadenaInterface
16      }
    }
```

📌 **Línea 2** se crea una instancia de la interface. **Línea desde 13 a 16** se anual 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 11** con el objeto de la interface llamamos a su método pasando el valor del elemento pulsado.

Reconstruye el ejemplo anterio y prueba su funcionamiento

Pila de fragments
Tasks y backStacks

Diálogo Fragments

Como hemos visto al principio del tema, la clase Fragment tiene varias derivadas que nos permiten trabajar con ellas para realizar tareas concretas. Para terminar este tema, veremos los Dialogos derivados de la clase DialogFragment. Para utilizar este tipo de diálogos deberemos crearnos una clase derivada de DialogFragment e implementar en ella el tipo de dialogo que queramos, usando un objeto de la clase AlertDialog (como esta parte ya la conocemos de temas anteriores, solo vamos a comentar un par de AlertDialog diferentes). El paso diferente al de los diálogos sin fragments, es que en vez demostrarlos directamente con show() tendremos que cargar el fragment mediante un FragmentManager. 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: