Programación Multimedia y Dispositivos Móviles

UD 7. Fragments

Javier Carrasco

Curso 2024 / 2025

Fragments

7. Fragments

Los **fragmentos** en Android, o ***fragments***, representan el comportamiento o una parte de la interfaz de usuario utilizando la clase `Fragment`. Se pueden combinar diferentes *fragments* para crear una actividad única. Además, un mismo *fragment* podrá reutilizarse en varias actividades.

Debes tener en cuenta que un *fragment* siempre deberá estar alojado en una actividad y se verá afectado por el ciclo de vida de la misma. Por ejemplo, si la actividad anfitriona pasa a pausa, todos los *fragments* que contenga pasarán a pausa. Pero, muy importante, los *fragments* tendrán su propio ciclo de vida. En la imagen puedes ver el ciclo de vida de un *fragment* mientras la actividad que lo contiene está activa.

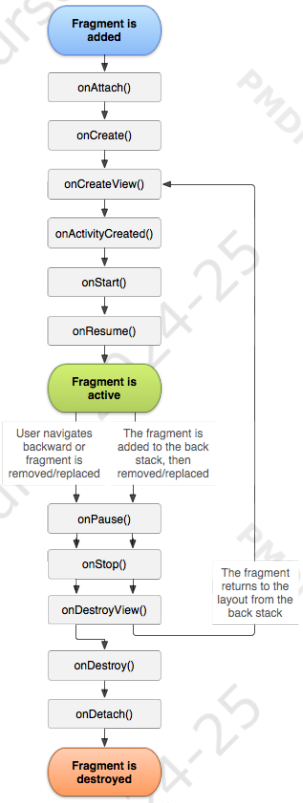


Figura 1

También deberás tener en cuenta que cuando se añade un *fragment* como parte de una actividad, éste se encontrará en un `ViewGroup` dentro de la jerarquía de vistas de la actividad, y el *fragment* definirá su propio diseño de vistas (visto en capítulos anteriores).

7.1. Primer uso de fragments

A continuación, se creará una nueva aplicación para trabajar con *fragments*, ésta permitirá ver la potencia que tienen y, para poder observar el ciclo de vida de los *fragments* se añadirán líneas al *log* que muestren su evolución.

Se creará el proyecto *"My Fragments"* utilizando **Android+Kotlin** con API mínima 25 para este ejemplo. Crea los siguientes *layouts* como se muestran a continuación.

activity_main.xml

```
<?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"
    android:padding="8dp"
    tools:context=".MainActivity">

    <com.google.android.material.button.MaterialButton
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/btn_text" />
```

```
    <FrameLayout
        android:id="@+id/fragmentHolder"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Los siguientes dos ficheros XML se crearán utilizando la opción ***New > Layout resource file*** que aparece al pulsar con el botón derecho sobre la carpeta ***res > layout*** del proyecto.

fragment_one.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="@color/frgOne"
    android:orientation="vertical"
    android:padding="8dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="@string/txt_fragment1"
        android:textAppearance="@style/TextAppearance.AppCompat.Display2" />
</LinearLayout>
```

fragment_two.xml será igual que *fragment_one.xml* cambiando los valores de las propiedades `text` por *"Fragment TWO"* y la propiedad `background` del *LinearLayout* por otro color.

Llegados a este punto, ya tendrás creados los *layouts* necesarios para continuar. Importante, no son actividades, por lo que no debes añadirlas al *manifest*. El siguiente paso será crear una nueva clase Kotlin llamada `FragmentOne`, que tendrá el siguiente código:

```
class FragmentOne : Fragment() {
    val TAG = "FragmentONE"

    override fun onAttach(context: Context) {
        Log.println(Log.INFO, TAG, "onAttach")
        super.onAttach(context)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        Log.println(Log.INFO, TAG, "onCreate")
        super.onCreate(savedInstanceState)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View? {
```

```
        Log.println(Log.INFO, TAG, "onCreateView")
        return inflater.inflate(R.layout.fragment_one, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        Log.println(Log.INFO, TAG, "onViewCreated")
        super.onViewCreated(view, savedInstanceState)
    }

    override fun onStart() {
        Log.println(Log.INFO, TAG, "onStart")
        super.onStart()
    }

    override fun onResume() {
        Log.println(Log.INFO, TAG, "onResume")
        super.onResume()
    }

    override fun onPause() {
        Log.println(Log.INFO, TAG, "onPause")
        super.onPause()
    }

    override fun onDestroyView() {
        Log.println(Log.INFO, TAG, "onDestroyView")
        super.onDestroyView()
    }

    override fun onDestroy() {
        Log.println(Log.INFO, TAG, "onDestroy")
        super.onDestroy()
    }

    override fun onDetach() {
        Log.println(Log.INFO, TAG, "onDetach")
        super.onDetach()
    }
}
```

Si te fijas, esta clase hereda de la clase `Fragment` y se sobrecargan (***Code > Override Methods...***) parte de sus métodos, concretamente los que intervienen en el ciclo de vida de un *fragment*, de esta forma se puede comprobar su comportamiento en el *log*, y ver como evoluciona durante su tiempo de vida.

Ahora, crea una segunda clase, `FragmentTwo`, que será exactamente igual a la anterior, cambiando la variable `TAG` a "*FragmentTWO*" y en el *return* del método `onCreateView()` deberá indicarse `R.layout.fragment_two` como *layout*.

Observa como quedaría la clase `MainActivity.kt` para darle vida a la aplicación utilizando los *fragments* preparados.

```kotlin
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    // Controla si está cargado o no FragmentOne.
    var isFragmentOneLoaded = true

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se muestra en pantalla el primer Fragment.
        showFragment(FragmentOne())

        // Listener del botón.
        binding.button.setOnClickListener {
            if (isFragmentOneLoaded)
                showFragment(FragmentOne())
            else showFragment(FragmentTwo())
        }
    }

    private fun showFragment(fragmentToLoad: Fragment) {
        // Se establece la transacción de fragments, necesarios para añadir,
        // quitar o reemplazar fragments.
        val transaction = supportFragmentManager.beginTransaction()

        // Indicamos el elemento del layout donde haremos el cambio.
        transaction.replace(R.id.fragmentHolder, fragmentToLoad)

        // Se establece valor a null para indicar que se está interesado en volver a ese
        // fragment al pulsar atrás, o se indicaría el nombre del fragment al que se quiere
        // volver. Se puede utilizar disallowAddToBackStack() para evitar la pila.
        transaction.addToBackStack(null)
        //transaction.disallowAddToBackStack()

        // Se muestra el fragment.
        transaction.commit()
        isFragmentOneLoaded = !isFragmentOneLoaded
    }
}
```

El resultado que se obtendrá con este código lo puedes ver en la siguiente figura, el *fragment* cambiará con cada pulsación del botón dentro del contenedor añadido en la actividad principal. En las pestañas *Logcat* y *Run* podrás observar la información producida por los *fragments* mediante la clase *Debug* y bajo las etiquetas *FragmentONE* y *FragmentTWO*.

En este ejemplo se han creado los *fragments* de manera manual, por un lado el *resource layout* y, por otro lado la clase *Kotlin*. Pero existe otra forma, de manera automática, utilizando la opción ***New > Fragment*** del menú *File* o desde el menú contextual. Esta segunda opción genera, para los conocimientos adquiridos hasta este punto, demasiado código que seguramente no se utilice o no se entienda para que sirve de momento.

Figura 2

7.2. Paso de parámetros entre fragments

En el ejemplo anterior se han creado varios *fragments* para representar distinta información, pero en ocasiones, esto puede resultar muy engorroso, ya que se tendría que generar un *fragment* para cada información a mostrar. Esto se puede evitar mediante el paso de información desde la actividad al *fragment* que se desee mostrar.

Para este ejemplo, el proyecto *"My Fragments 2"*, se creará un único *fragment* como el que se muestra a continuación (`fragment_new.xml`), en este caso se utiliza la creación automática del *fragment* (***New > Fragment > Fragment (Blank)***):

```xml
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout 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:background="@color/colorImpar"
    tools:context=".NewFragment">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="@string/new_fragment"
        android:textAppearance="@style/TextAppearance.AppCompat.Display2" />

</FrameLayout>
```

A continuación, observa la clase *Kotlin* creada, esta se encarga de gestionar el *fragment*, muy similar a la vista en el ejemplo anterior, al crearse de manera automática, se puede hacer uso de la función auto-generada `newInstance()`. Observa como quedaría adaptada a las necesidades del ejemplo que se va a desarrollar.

```kotlin
class NewFragment : Fragment() {
    private lateinit var binding: FragmentNewBinding

    private var numFrag: Int? = null
    private var colorBack: Int? = null
    private val TAG = "FragmentNew"

    override fun onAttach(context: Context) {
        Log.d(TAG, "onAttach")
        super.onAttach(context)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Si existen argumentos pasados en el Bundle desde la llamada, se asignan a las
        // propiedades. ARG_NUMFRAG y ARG_COLORBACK están declaradas en MainActivity.
        arguments?.let {
            this.numFrag = it.getInt(ARG_NUMFRAG)
            this.colorBack = it.getInt(ARG_COLORBACK)
        }
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentNewBinding.inflate(inflater)
        return binding.root
    }

    companion object {
        /**
         * Use this factory method to create a new instance of
         * this fragment using the provided parameters.
         *
         * @param nFrag Parameter 1.
         * @param cBack Parameter 2.
         * @return A new instance of fragment NewFragment.
         */
        // TODO: Rename and change types and number of parameters
        @JvmStatic
        fun newInstance(nFrag: Int, cBack: Int) =
            NewFragment().apply {
                arguments = Bundle().apply {
                    putInt(ARG_NUMFRAG, nFrag)
                    putInt(ARG_COLORBACK, cBack)
                }
            }
    }

```

```
    // Se modifican las propiedades en este método para asegurar que la activity
    // está creada y evitar así posibles fallos.
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        Log.d(TAG, "onViewCreated")
        binding.frame.setBackgroundColor(colorBack!!)
        binding.textView.text = getString(R.string.new_fragment, numFrag)
        super.onViewCreated(view, savedInstanceState)
    }

    // El resto de métodos sobrecargados son exactamente igual que los utilizados en el
    // ejemplo MyFragments.
    ...
```

En el método `onCreate()` se utiliza la propiedad `arguments` que es un `Bundle`, ésta es una variable de la clase `Fragment` que se encarga de recoger los parámetros que se le pasen al *fragment*.

¿Qué es un `Bundle`? Bueno, pues viene a ser un *MAP* en el cual se almacena, siguiendo el formato clave-valor, todo aquello que pueda ser de interés, en este caso, los parámetros que se quieren pasar al *fragment*.

A continuación, observa como quedaría la clase principal `MainActivity.kt` para cargar el *fragment*, y como crear el *bundle* (empaquetado) para los argumentos que se quieren pasar. La UI de la actividad sería como la vista en el ejemplo anterior.

```
internal const val ARG_NUMFRAG = "numFrag"
internal const val ARG_COLORBACK = "colorBack"

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private var numfrag = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        // Se muestra el primer fragment.
        showFragment()
        binding.button.setOnClickListener {
            showFragment()
        }
    }

    private fun showFragment() {
        val transaction = supportFragmentManager.beginTransaction()

        // Declaración básica del Fragment.
        val fragment = NewFragment()
```

```
        // Se crea la variable Bundle para "empaquetar" los datos a pasar.
        val bundle = Bundle()
        bundle.putInt(ARG_NUMFRAG, ++numfrag)
        bundle.putInt(
            ARG_COLORBACK,
            (if ((numfrag % 2) == 0)
                getColor(R.color.colorPar)
            else getColor(R.color.colorImpar))
        )
        fragment.arguments = bundle

        transaction.replace(binding.fragmentHolder.id, fragment)
        transaction.disallowAddToBackStack()
        transaction.commit()
    }
}
```

Fíjate en el método `showFragment()` como se crea una variable de tipo `Bundle` y se añaden los datos que quieren pasarse al *fragment*, una vez *"empaquetados"*, se asignan a la variable `fragment`. El resto quedaría igual. Si quieres ahorrarte la creación del *Bundle* puedes hacer uso del método `newInstance` generado en la clase `NewFragment`, cambiando la declaración del *fragment* como se muestra a continuación y la asignación de parámetros.

```
// Declaración del Fragment mediante newInstance.
val fragment = NewFragment.newInstance(
    ++numfrag,
    (if ((numfrag % 2) == 0) getColor(R.color.colorPar) else getColor(R.color.colorImpar))
)
```

Observa como se hace uso de `companion object`, recuerda que son elementos comunes a todas las instancias de una clase, si se hace un símil con Java, serían las propiedades de tipo estático.

Figura 3

7.3. Comunicación bidireccional fragment-activity

En este tercer ejemplo, se creará una aplicación que permita devolver datos a la actividad contenedora del *fragment*. Partiendo del ejemplo anterior, en el que se puede ver que pasar información de una actividad a un *fragment* es relativamente sencillo, pero al contrario no es así. Existen varias formas de afrontar esta situación, aquí se verá la que puede, quizá, resultar algo más sencilla.

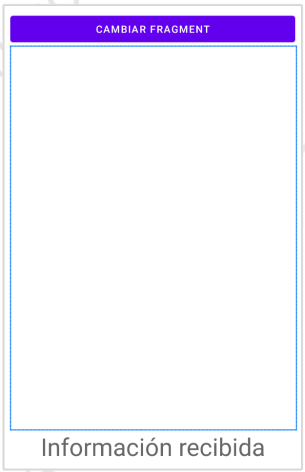


Figura 4

Figura 5

Partiendo del diseño anterior, se añadirá un *TextView* a la `activity_main.xml` bajo el *FrameLaoyut* para mostrar la información que se devuelva desde el *fragment*.

En la vista `fragment_new.xml`, se añadirá un *EditText* y un *Button* al *layout* para enviar datos de vuelta a la actividad contenedora. El aspecto final puede ser parecido al que se muestra en la imagen adjunta.

De vuelta al código, deberá modificarse la clase `NewFragment.kt`, la idea es añadir una interface para que todas las actividades que quieran utilizar la clase deban implementarla, esta interface permitirá evitar posibles errores de memoria pasando la referencia de la actividad.

```
class NewFragment : Fragment() {
    ...
    interface RequestData {
        fun resultData(dato: String)
    }

    private var fragmentData: RequestData? = null
    ...
```

También se modificará el método `onAttach()` (método que se ejecuta cuando el *fragment* es adjuntado a la actividad) para notificar el error en caso de no implementarse la interface.

```
.myfragments3      D  onAttach
.myfragments3      D  Shutting down VM
.myfragments3      E  FATAL EXCEPTION: main
                      Process: es.javiercarrasco.myfragments3, PID: 10840
                      java.lang.RuntimeException: es.javiercarrasco.myfragments3.MainActivity@c09b490 debes implementar la interface RequestData
                          at es.javiercarrasco.myfragments3.NewFragment.onAttach(NewFragment.kt:49)
```

Figura 6

```
override fun onAttach(context: Context) {
    Log.d(TAG, "onAttach")
    super.onAttach(context)

```

```
    if (context is RequestData)
        fragmentData = context
    else throw RuntimeException("${requireContext()} debes implementar la interface " +
            "RequestData")
}
```

También se añadirá a la clase `NewFragment.kt` un método que se encargará de pasar la información a la actividad.

```
fun updateData() {
    if (!binding.editTextFrag.text!!.isEmpty())
        fragmentData?.resultData(binding.editTextFrag.text.toString())
    else fragmentData?.resultData("Sin información")
}
```

En los *fragments*, a diferencia de las actividades, deben añadirse las funcionalidades al método `onViewCreated()` para evitar posibles problemas.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    Log.d(TAG, "onViewCreated")
    binding.frame.setBackgroundColor(colorBack!!)
    binding.textView.text = getString(R.string.new_fragment, numFrag)

    binding.button.setOnClickListener {
        updateData()
    }

    super.onViewCreated(view, savedInstanceState)
}
```

Ahora, sólo queda eliminar la referencia a la información que se pasa al quitar el *fragment* de la vista, para ello se utilizará el método `onDetach()`.

```
override fun onDetach() {
    Log.d(TAG, "onDetach")
    super.onDetach()
    fragmentData = null
}
```

Por último, de vuelta a la clase `MainActivity.kt`, se implementará la interface creada y se mostrará la información obtenida desde el *fragment* en el *TextView* destinado para ello.

```
class MainActivity : AppCompatActivity(), NewFragment.RequestData {
```

Al añadir la interface, verás como se marca un error, eso se debido a que falta la implementación del método `resultData()` de la interface.

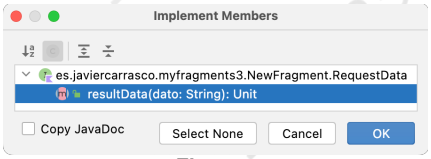


Figura 7

Una vez sobrecargado el método, se añadirá el siguiente código.

```
override fun resultData(dato: String) {
    binding.tvInfoRecibida.text = dato
}
```

Con esto se consigue establecer una comunicación bidireccional de forma nativa, sin la necesidad de añadir librerías externas y evitando posibles fallos de memoria. Puedes ver el resultado en la figura que aparece a continuación.

Este tipo de comunicación puede utilizarse para que la actividad que contiene los *fragments*, pueda pasar la información de un *fragment* a otro, utilizando para ello la actividad contenedora como intermediaria. Puedes ver la *app* funcionando en Google Play[1].

Figura 8

7.4. Pestañas + ViewPager2

Las pestañas, o *tabs*, permiten mostrar información de una manera más ordenada. Sistema muy utilizado en aplicaciones con varias vistas y diseñadas siguiendo el modelo *single-activity*. A continuación, se verá un ejemplo tomando como base las recomendaciones de *Material Design*[2].

La versión anterior, *ViewPager* está marcada como *deprecated*, debido a que en 2019 se lanzó ***ViewPager2***[3], que varía ligeramente con respecto a la primera versión, pero que recomienda su uso desde entonces, esta será la que se utilizará.

Partiendo de un nuevo proyecto llamado *"MyTabs"*, en la actividad `activity_main.xml` se añadirá el código necesario para poder cargar las pestañas y el componente para *ViewPager2* para mostrar los *fragments*. En primer lugar se añadirá el contenedor de las pestañas.

1 My Fragments 2 (https://play.google.com/store/apps/details?id=es.javiercarrasco.myfragments2&gl=ES)
2 *Material Design* (https://material.io/components/tabs/)
3 *ViewPager2* (https://developer.android.com/reference/kotlin/androidx/viewpager2/widget/ViewPager2)

```xml
<?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"
    tools:context=".MainActivity">

    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tabLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Ahora, en la clase se la actividad principal, se añadirán mediante código las pestañas en el método *onCreate* y, desde el método *onStart*, se controlará que pestaña se pulsa mediante un *listener*.

```
// En onCreate
with(binding.tabLayout) {
    addTab(newTab().setText(getString(R.string.txt_animals)))
    addTab(newTab().setText(getString(R.string.txt_food)))
    addTab(newTab().setText(getString(R.string.txt_persons)))
}

// En onStart
binding.tabLayout.addOnTabSelectedListener(object : TabLayout.OnTabSelectedListener {
    override fun onTabSelected(tab: TabLayout.Tab?) {
        Log.i("TabSelected", tab!!.text.toString())
    }
    override fun onTabUnselected(tab: TabLayout.Tab?) {
        Log.i("TabUnselected", tab!!.text.toString())
    }
    override fun onTabReselected(tab: TabLayout.Tab?) {
        Log.i("TabReselected", tab!!.text.toString())
    }
})
```

Podrás observar en *Logcat* como va indicando lo que va sucediendo con cada selección de una pestaña. A continuación, hay que hacer que muestre algo la aplicación, un *fragment* para cada pestaña. Crea tres *fragments* como el que se muestra a continuación, uno para animales, otro para comida y un último para personas. El *layout* para el *fragment* de animales puede resultar así.

```xml
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout 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"
    tools:context=".AnimalsFragment">
```

```
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center_horizontal|center_vertical"
        android:text="@string/txt_animals" />

</FrameLayout>
```

Y su clase asociada quedaría de la siguiente forma.

```
class AnimalsFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return FragmentAnimalsBinding.inflate(inflater, container, false).root
    }
}
```

Bastante sencillo, ya que únicamente se mostrará un texto indicando el *fragment* que se ha cargado. Ahora se añade el *ViewPager2* al *layout* de la actividad principal, que permitirá mostrar la información de cada pestaña, los *fragments* creados. Tras el *TabLayout* añade el siguiente código.

```
    ...
    </com.google.android.material.tabs.TabLayout>

    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/viewPager2"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Con esto, ya estaría el diseño de la actividad principal, ahora habría que conectarlo todo y hacerlo funcionar. Para conectar los *fragments* con el *ViewPager2* (componente que permite pasar páginas de izquierda a derecha, y viceversa) se creará la clase ***CustomPagerAdapter*** que implementará *FragmentStateAdapter*[4]. A esta clase se le pasará como parámetro un *FragmentManager* y el ciclo de vida de la aplicación. Además, se sobrecargarán los métodos que ayudarán a obtener el *fragment* a mostrar, el número de pestañas, o *fragments*, y se añadirán dos métodos más, uno para añadir el *fragment* junto con el título y otro para obtener el título.

```
class CustomPagerAdapter(fragmentManager: FragmentManager, lifecycle: Lifecycle) :
    FragmentStateAdapter(fragmentManager, lifecycle) {

    private val mFragmentList = ArrayList<Fragment>()
    private val mFragmentTitleList = ArrayList<String>()

```

4 FragmentStateAdapter (https://developer.android.com/reference/kotlin/androidx/viewpager2/adapter/FragmentStateAdapter)

```
    // Devuelve el tamaño del array de fragments.
    override fun getItemCount(): Int {
        return mFragmentList.size
    }

    // Devuelve el frament en la posición indicada.
    override fun createFragment(position: Int): Fragment {
        return mFragmentList[position]
    }

    // Añade un fragment a la lista.
    fun addFragment(fragment: Fragment, title: String) {
        mFragmentList.add(fragment)
        mFragmentTitleList.add(title)
    }

    // Devuelve el título de la pestaña.
    fun getPageTitle(position: Int): CharSequence {
        return mFragmentTitleList[position]
    }
}
```

Observa como el adaptador lleva el control de la pestaña asociada al *fragment* mediante las dos listas, de otra manera debería hacerse desde fuera. De esta forma, todo lo relacionado con las pestañas y las páginas está junto.

La clase `FragementStateAdapter` se utiliza cuando se trabaja con una vista con múltiples *fragments*, de forma que cuando un *fragment* no es visible al usuario, éste puede ser destruido, guardando el estado. Esto permite una menor carga de memoria. Observa ahora como quedaría la clase principal, su método *onCreate*, para poder mostrar el contenido en el *ViewPager2* y tener unas pestañas completamente funcionales.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    // Se instancia el adaptador para el ViewPager2.
    val adapter = CustomPagerAdapter(supportFragmentManager, lifecycle)
    adapter.addFragment(AnimalsFragment(), getString(R.string.txt_animals))
    adapter.addFragment(FoodFragment(), getString(R.string.txt_food))
    adapter.addFragment(PersonsFragment(), getString(R.string.txt_persons))

    // Se asigna el adapter al ViewPager2.
    binding.viewPager2.adapter = adapter

    TabLayoutMediator(binding.tabLayout, binding.viewPager2) { tab, pos ->
        tab.text = adapter.getPageTitle(pos)
    }.attach()
}
```

Observa que se pasa el ciclo de vida de la aplicación, necesario para la gestión de los *fragments*, y se establece la vinculación entre el *ViewPager2* y el *TabLayout* mediante el uso del método `TabLayoutMediator()` [5].

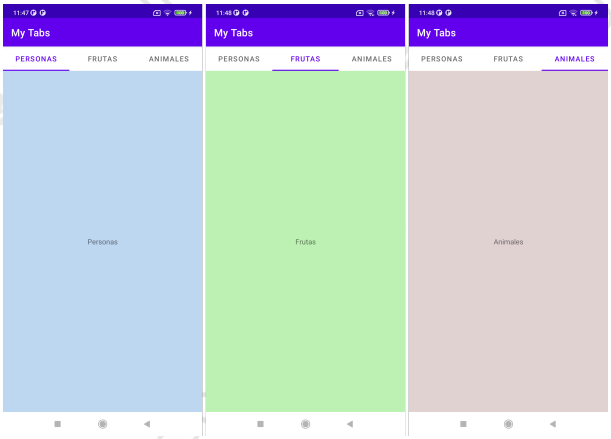


Figura 9

Se puede añadir un efecto al cambiar de página, para ello se utilizará el método `setPageTransformer()`, al que se le pasará un objeto de tipo `ViewPager2.PageTransformer`. Por ejemplo, se crea la clase `DepthPageTransformer` con el código necesario para poder aplicar un efecto de profundidad[6], lo encontrarás en la documentación oficial de Android, también podrás encontrar el efecto zoom[7]. Añade la siguiente línea antes de cargar las pestañas, por ejemplo.

```
// Efectos para el ViewPager2.
binding.viewPager2.setPageTransformer(DepthPageTransformer())
```

Clase DepthPageTransformer

```
class DepthPageTransformer : ViewPager2.PageTransformer {
    private val MIN_SCALE = 0.75f

    override fun transformPage(view: View, position: Float) {
        view.apply {
            val pageWidth = width
            when {
                position < -1 -> { // [-Infinity,-1)
                    // This page is way off-screen to the left.
                    alpha = 0f
                }
                position <= 0 -> { // [-1,0]
                    // Use the default slide transition when moving to the left page
                    alpha = 1f
                    translationX = 0f
```

5 Agregar pestañas con un objeto *TabLayout* (https://developer.android.com/guide/navigation/navigation-swipe-view-2)
6 Profundidad de página (https://developer.android.com/training/animation/screen-slide-2#depth-page)
7 Zoom de página (https://developer.android.com/training/animation/screen-slide-2#zoom-out)

```
                    translationZ = 0f
                    scaleX = 1f
                    scaleY = 1f
                }
                position <= 1 -> { // (0,1]
                    // Fade the page out.
                    alpha = 1 - position
                    // Counteract the default slide transition
                    translationX = pageWidth * -position
                    // Move it behind the left page
                    translationZ = -1f
                    // Scale the page down (between MIN_SCALE and 1)
                    val scaleFactor = (MIN_SCALE + (1 - MIN_SCALE) *
                            (1 - Math.abs(position)))
                    scaleX = scaleFactor
                    scaleY = scaleFactor
                }
                else -> { // (1,+Infinity]
                    // This page is way off-screen to the right.
                    alpha = 0f
                }
            }
        }
    }
}
```

Por último, se pueden añadir iconos a las pestañas y darle un toque distintivo a la aplicación. Aprovechando el adaptador creado, que como recordarás, se encarga de asociar cada *fragment* con su pestaña y su título, se añadirán también los iconos. En primer lugar se añadirá una nueva propiedad a la clase *CustomPagerAdapter*.

```
private val mFragmentIcon = ArrayList<Drawable>()
```

Se modifica el método encargado de añadir los *fragments* añadiendo un tercer parámetro.

```
fun addFragment(fragment: Fragment, title: String, icon: Drawable) {
    mFragmentList.add(fragment)
    mFragmentTitleList.add(title)
    mFragmentIcon.add(icon)
}
```

Y se añade un nuevo método para obtener el icono.

```
fun getIconTab(position: Int): Drawable {
    return mFragmentIcon[position]
}
```

En la clase principal también cambiará la llamada al método para crear los *fragments*, debiendo pasarle el icono como tercer parámetro.

```
adapter.addFragment(
    AnimalsFragment(),
    getString(R.string.txt_animals),
    AppCompatResources.getDrawable(this, R.drawable.ic_animals)!!
)
```

Y ya por último, se asignará el icono a la pestaña en el *TabLayoutMediator*.

```
TabLayoutMediator(binding.tabLayout, binding.viewPager2) { tab, pos ->
    tab.text = adapter.getPageTitle(pos)
    tab.icon = adapter.getIconTab(pos)
}.attach()
```

Como resultado se obtendrá algo parecido a lo que se muestra en la siguiente imagen.

Figura 10

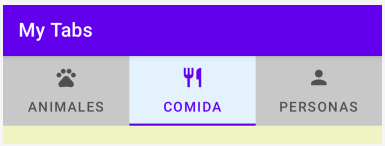
A continuación se verá un cambio de imagen para las *tabs*, si quieres cambiar los colores de fondo de la pestaña seleccionada, o las no seleccionadas, deben realizarse una serie de modificaciones en los ficheros `res`. El primer paso consistirá en añadir los colores deseados para las pestañas en el fichero `colors.xml`.

```
<color name="tabSelected">#E3F2FD</color>
<color name="tabUnselected">#C8C8C8</color>
```

Seguidamente, se creará un nuevo fichero, que para este caso se llamará `tab_color_selected.xml`, que se ubicará en `res/drawable`, cuyo contenido será algo parecido a lo que se muestra a continuación.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/tabSelected" android:state_selected="true" />
    <item android:drawable="@color/tabUnselected" />
</selector>
```

Por último, de vuelta al *layout* `activity_main.xml`, donde se encuentra el *widget TabLayout*, se asignará a su propiedad `tabBackground` el valor `@drawable/tab_color_selected`, verás como el indicador de color aparece mostrando un cuadro partido en dos con los colores seleccionados.



En el ejemplo que se acaba de hacer, se crean tres *fragments* fijos con sus tres pestañas, pero puede darse el caso que se necesite crea los *fragments* y sus pestañas de forma dinámica, según se vayan necesitando.

7.4.1. Creación dinámica de fragments y tabs

Para este nuevo proyecto, que será una nueva versión del anterior, el `activity_main.xml` será igual al visto, pero únicamente se creará un *fragment* que hará las veces de plantilla para el resto de *fragments* a crear.

fragment_page.xml

```
<?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"
    android:padding="8dp"
    tools:context=".PageFragment">

    <TextView
        android:id="@+id/nameFragment"
        style="@style/TextAppearance.AppCompat.Display1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:gravity="center"
        android:text="@string/txt_fragment" />

    <TextView
        android:id="@+id/txtFragment"
        style="@style/TextAppearance.AppCompat.Body2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/txt_to_show" />

</LinearLayout>
```

El código de la clase `PageFragment` será el que se muestra a continuación, como se crearán de manera dinámica, es necesario el método visto anteriormente para instanciar nuevos *fragments*.

```
class PageFragment : Fragment() {
    private lateinit var binding: FragmentPageBinding
    private var nombre: String? = null
    private var texto: String? = null

    companion object {
```

```
        @JvmStatic
        fun newInstance(name: String, text: String) =
            PageFragment().apply {
                arguments = Bundle().apply {
                    putString(ARG_NAME, name)
                    putString(ARG_TEXT, text)
                }
            }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        arguments?.let {
            this.nombre = it.getString(ARG_NAME)
            this.texto = it.getString(ARG_TEXT)
        }
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentPageBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.nameFragment.text = getString(R.string.txt_fragment, nombre)
        binding.txtFragment.text = getString(R.string.txt_to_show, texto)
    }
}
```

Observa como quedaría la clase principal, que únicamente varía con respecto a la versión anterior en la creación de los *fragments* y el paso de parámetros.

```
const val ARG_NAME = "name"
const val ARG_TEXT = "text"

class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private val NUM_FRAGMENTS = 5 // Número de fragments a crear.

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se instancia el adaptador para el ViewPager2.
        val adapter = CustomPagerAdapter(supportFragmentManager, lifecycle)
```

```
        // Se añaden los fragments y los títulos de pestañas.
        for (num in 1..NUM_FRAGMENTS) {
            adapter.addFragment(
                PageFragment.newInstance(
                    getString(R.string.txt_fragment, num.toString()),
                    getString(R.string.txt_to_show, num.toString())
                ),
                "Frg $num",
                AppCompatResources.getDrawable(this, R.drawable.ic_emoticon)!!
            )
        }

        // Se asigna el adapter al ViewPager2.
        binding.viewPager2.adapter = adapter

        // Permite elegir la dirección del paginado.
        binding.viewPager2.orientation = ViewPager2.ORIENTATION_HORIZONTAL

        TabLayoutMediator(binding.tabLayout, binding.viewPager2) { tab, pos ->
            tab.text = adapter.getPageTitle(pos)
            tab.icon = adapter.getIconTab(pos)
        }.attach()

        // Efectos para el ViewPager2.
        binding.viewPager2.setPageTransformer(DepthPageTransformer())
    }
}
```

Se ha creado una constante en la que se indica el número de *fragments* a crear, pero el límite lo puede marcar otro factor, según sea tu necesidad. También se ha modificado la propiedad `orientation` del *ViewPager2*, ésta permite elegir el tipo de desplazamiento de las páginas, vertical u horizontal. La clase `CustomPagerAdapter` quedará tal y como se vio en el ejemplo anterior, se reutiliza. El resultado puedes verlo en la figura que se muestra a continuación.

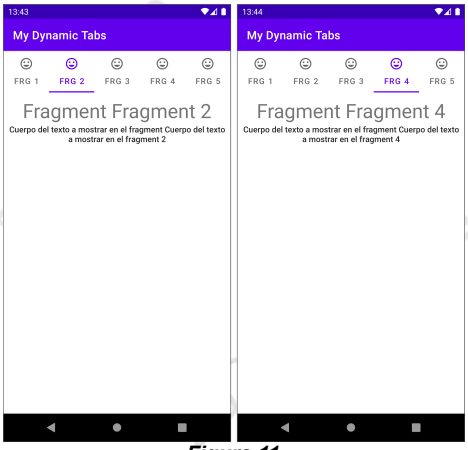


Figura 11

7.5. Navigation

Ahora que conoces el uso de *fragments* y *activities*, se verá un componente que puede hacerte la vida más sencilla. El componente *Navigation* de Android JetPack[8]. Este componente te permitirá añadir navegación a tu aplicación de una forma fácil, permitiéndote hacer desde navegaciones sencillas, hasta patrones más complejos como barras de apps, paneles laterales, etc.

El componente *Navigation* está formado por tres partes que debes conocer para su correcta implementación.

- El gráfico de navegación: recurso XML que centralizará todo el sistema de navegación de la aplicación, facilitando su entendimiento y gestión de la navegación.
- El `NavHost`: será un contenedor vacío que muestra los destinos del gráfico de navegación. El componente *Navigation* tiene una implementación `NavHost` predeterminada, `NavHostFragment`, que mostrará los destinos de los *fragments*.
- El `NavController`: este objeto administra la navegación de la aplicación dentro del `NavHost`. `NavController` se encargará de gestionar el intercambio del contenido en el objeto `NavHost` según los usuarios navegan por la aplicación.

Para añadir el componente *Navigation* al proyecto, se comenzará por añadir las siguientes dependencias al fichero `build.gradle(Module: app)`.

```
// Navigation
implementation "androidx.navigation:navigation-fragment-ktx:2.5.3"
implementation "androidx.navigation:navigation-ui-ktx:2.5.3"

// Feature module Support
implementation "androidx.navigation:navigation-dynamic-features-fragment:2.5.3"

// Testing Navigation
androidTestImplementation "androidx.navigation:navigation-testing:2.5.3"

// Jetpack Compose Integration
implementation "androidx.navigation:navigation-compose:2.6.0-alpha07"
```

Además, se recomienda añadir el *plugin* ***androidx.navigation.safeargs.kotlin*** en el fichero `build.gradle(Module: app)` para añadir seguridad de tipos al navegar y pasar datos entre los destinos. Deberás añadirlo en la sección *plugins* de la siguiente forma.

```
plugins {
    ...
    id 'androidx.navigation.safeargs.kotlin' version '2.5.3'
}
```

Una vez hecho esto, ya puedes sincronizar el *Gradle* para comenzar con el ejemplo que se muestra a continuación.

8 Navigation (https://developer.android.com/guide/navigation)

Antes de añadir el componente *FragmentContainerView*, que es el que contendrá el `NavHost`, debe existir al menos un *fragment* creado. Por lo tanto, para este ejemplo, se crearán tres *fragments* (*FragmentUno*, *FragmentDos* y *FragmentTres*).

A continuación, se muestra uno de ellos, para ilustrar el funcionamiento del componente *Navigation*, se diferenciarán por un *TextView* y el color de fondo, además tendrán un *Button* para pasar al siguiente *fragment* o destino.

fragment_uno.xml

```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:gravity="center_vertical"
    android:background="@color/frgOne"
    android:orientation="vertical"
    android:padding="8dp">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/txt_fragment"
        android:textAppearance="@style/TextAppearance.AppCompat.Display2" />

    <com.google.android.material.button.MaterialButton
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="@string/txtBtn1" />

</LinearLayout>
```

UnoFragment.kt

```kotlin
class UnoFragment : Fragment() {
    private lateinit var binding: FragmentUnoBinding

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = FragmentUnoBinding.inflate(inflater, container, false)
        return binding.root
    }
}
```

Una vez creados los tres *fragments*, con sus correspondientes clases (*UnoFragment.kt*, *DosFragment.ky* y *TresFragment.kt*), ya se puede pasar a construir el *Navigation*, para ello, añade un nuevo fichero de recurso de tipo *Navigation*.

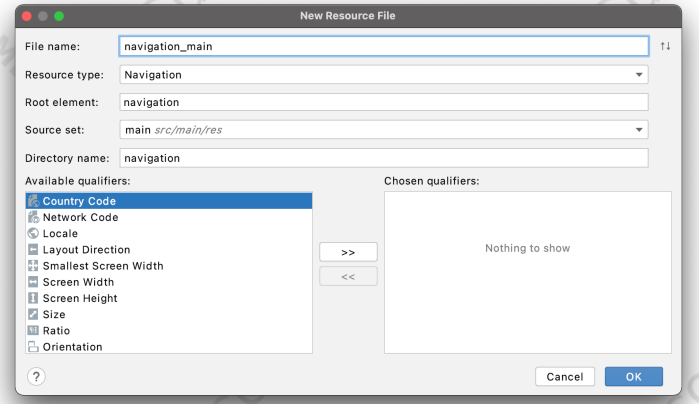


Figura 12

Seguidamente, en la `activity_main.xml` deberás añadir el componente ***NavHostFragment***, este se añadirá como un *FragmentContainerView*. El resultado de la *activity* puede ser como se muestra a continuación.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragmentContainerView"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/navigation_main"
        tools:layout="@layout/fragment_uno" />

</LinearLayout>
```

La propiedad `tools:layout` permite ver una vista previa del *fragment* asociado en la vista diseño, lo que puede facilitar la tarea de diseño. La propiedad `app:defaultNavHost` a *true* permite interceptar la pulsación del botón “atrás”.

Vuelve a `navigation_main.xml` y diseña el sistema de navegación de la aplicación. Para este caso se implementará un sistema de navegación secuencial, es decir, del *fragment 1* se pasará la *fragment 2* y de este al 3.

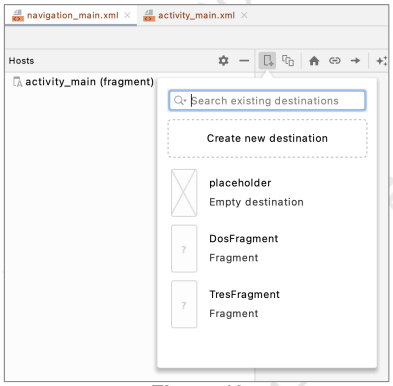


Figura 13

Fíjate en la figura, muestra como añadir destinos al sistema *Navigation*. Puedes añadir destinos ya creados, como es el caso de los *fragments* ya disponibles, o destinos vacíos que más adelante podrás completar.

Otra acción interesante de la que dispones es la *"casita"* que tienes en la barra de herramientas, esta permitirá especificar cual será el destino inicial de la navegación.

Siguiendo las indicaciones dadas, el resultado gráfico de `navigation_main.xml` puede verse como se muestra en la siguiente figura.

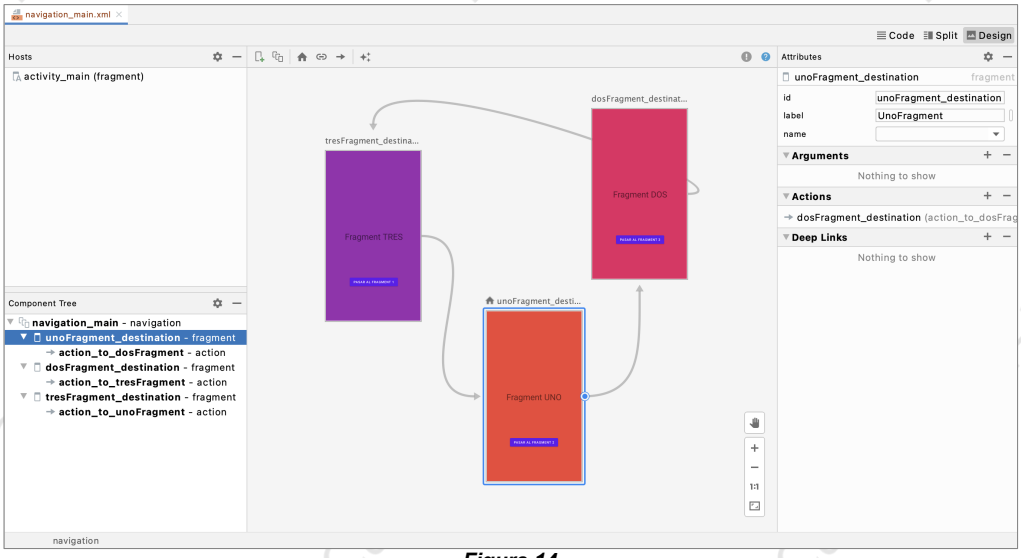


Figura 14

Si no consigues ver la vista previa de los *fragments* en el *Navigation*, puedes añadir la siguiente propiedad a cada uno de los *fragments*.

```
1  tools:layout="@layout/fragment_uno"
```

Para crear las acciones bastará con seleccionar uno de los destinos y utilizar el tirador azul que aparece para hacer llegar el enlace al nuevo destino.

Llegados a este punto, ya dispones de la estructura *Navigation* montada, a continuación, se creará la funcionalidad de los botones para poder realizar las acciones de navegación. En el método `onCreateView()` de cada *fragment* se añadirá el siguiente código después de instanciar la variable `binding`.

```
binding = FragmentUnoBinding.inflate(inflater, container, false)

binding.button.setOnClickListener {
    findNavController().navigate(UnoFragmentDirections.actionUnoFragmentToDosFragment())
}
```

Este código será el mismo en los tres fragmentos, sustituyendo en cada caso el nombre del argumento que se pasa al método `navigate()` de `findNavController()`.

Al probar la aplicación, verás que los cambios son algo bruscos, si quieres cambiar eso, en las acciones, puedes establecer una animación para realizar el cambio, tanto de salida como de entrada, estas propiedades son `enterAnim` y `exitAnim`. Puedes probar con las animaciones que vienen por defecto, como *slide in left* y *slide out right*.

Ahora, se verá como pasar parámetros entre los distintos destinos. Para ilustrar este ejemplo, se pasará como argumento el número de *fragment* que se mostrará en el *TextView*. El primer paso será añadir el parámetro que se quiere en cada uno de los *fragments*, ya que los tres van a funcionar exactamente igual.

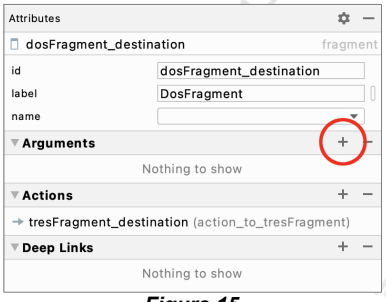


Figura 15

En `navigation_main.xml`, selecciona uno a uno los destinos y, en el lateral de atributos, añade a cada uno el argumento `numFragment` de tipo *String*, añade también un valor por defecto al argumento para poder controlar el estado del mismo, por ejemplo, el texto *"nulo"*.

Una vez hecho, si seleccionas las acciones (las flechas), verás como aparece en atributos, en la sección *"Arguments"*, los argumentos que intervienen en la navegación hacia cada uno de los destinos del gráfico de navegación. El siguiente paso será modificar la llamada al destino en el *listener* de los botones.

```
binding.button.setOnClickListener {
    findNavController().navigate(
        UnoFragmentDirections.actionUnoFragmentToDosFragment(
            getString(R.string.txt_fragment, "DOS")
        )
    )
}
```

De esta forma, se puede pasar el argumento directamente en la llamada a la acción, el siguiente paso será recogerlo en el *fragment* destino, para ello, se utilizará un *Bundle*, que es el método utilizado por el sistema para pasar los parámetros de un destino a otro, como verás, este sistema, es mucho más sencillo que el visto anteriormente.

```
if (UnoFragmentArgs.fromBundle(requireArguments()).numFragment != "nulo")
    binding.textView.text = UnoFragmentArgs.fromBundle(
        requireArguments()
    ).numFragment
else binding.textView.text = "INICIO"
```

En este fragmento de código se añade una comprobación del estado del argumento, ya que si se hace directamente en el primer *fragment*, la aplicación fallará al no existir el argumento, de ahí el motivo de asignar un valor por defecto.