Programación Multimedia y Dispositivos Móviles

UD 7. Fragments

Javier Carrasco

Curso 2021 / 2022

Última actualización: septiembre de 2021.
Versión impresa en Amazon: https://www.amazon.es/dp/B08NM4XV4Q

Fragments

7. *Fragments*

Los **fragmentos** en Android, o *fragments*, representan el comportamiento o una parte de la interfaz de usuario utilizando la clase `FragmentActivity`. Se pueden combinar diferentes fragmentos para crear una *activity* única. Además, los fragmentos podrán reutilizarse en diferentes *activities*.

Debes tener en cuenta que un fragmento siempre deberá estar alojado en una *activity* y se verá afectado por el ciclo de vida de la misma. Por ejemplo, si la *activity* anfitriona pasa a pausa, todos los fragmentos que contenga también pasarán a este estado. Pero, muy importante, los fragmentos 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.

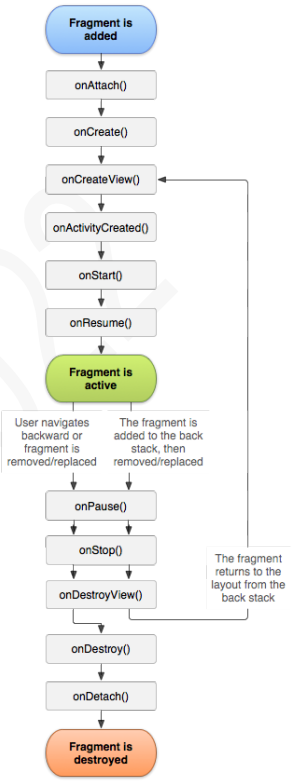

Figura 1

También deberás tener en cuenta que cuando se añade un fragmento como parte de una *activity*, éste se encontrará en un `ViewGroup` dentro de la jerarquía de vistas de la actividad, y el fragmento definirá su propio diseño de vistas (visto en el capítulo 4).

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 partirá de un nuevo proyecto **Android+Kotlin** con API mínima 21 para este caso de estudio. 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"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/btn_change"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="10dp"
```

```
        android:text="@string/btn_text"
        android:textAllCaps="false"/>

    <FrameLayout
        android:id="@+id/fragment_holder"
        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"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/relativeLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#33ccff">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="@string/txt_fragment1"
        android:textColor="@color/colorPrimary"
        android:textSize="30sp"
        android:textStyle="bold" />
</RelativeLayout>
```

fragment_two.xml será igual que *fragment_one.xml* cambiando los valores de las propiedades `text` por *"Fragment TWO"* y `textColor` por *colorAccent* en el *TextView*, y la propiedad `background` del *RelativeLayout* por *#ffcc33*.

Llegados a este punto, ya tendrás creados los *layouts* necesarios para continuar. Importante, no son *activities*, 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.d(TAG, "onAttach")
        super.onAttach(context)
    }
```

```
8       override fun onCreate(savedInstanceState: Bundle?) {
9           Log.d(TAG, "onCreate")
10          super.onCreate(savedInstanceState)
11      }
12
13      override fun onCreateView(
14          inflater: LayoutInflater,
15          container: ViewGroup?,
16          savedInstanceState: Bundle?
17      ): View? {
18          Log.d(TAG, "onCreateView")
19          return inflater.inflate(R.layout.fragment_one, container, false)
20      }
21
22      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
23          Log.d(TAG,"onViewCreated")
24          super.onViewCreated(view, savedInstanceState)
25      }
26
27      override fun onStart() {
28          Log.d(TAG, "onStart")
29          super.onStart()
30      }
31
32      override fun onResume() {
33          Log.d(TAG, "onResume")
34          super.onResume()
35      }
36
37      override fun onPause() {
38          Log.d(TAG, "onPause")
39          super.onPause()
40      }
41
42      override fun onDestroyView() {
43          Log.d(TAG, "onDestroyView")
44          super.onDestroyView()
45      }
46
47      override fun onDestroy() {
48          Log.d(TAG, "onDestroy")
49          super.onDestroy()
50      }
51
52      override fun onDetach() {
53          Log.d(TAG, "onDetach")
54          super.onDetach()
55      }
56  }
```

Si te fijas, esta clase hereda de la clase `Fragment` y se sobrecarga (***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*.

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.
        showFragmentOne()

        // Listener del botón.
        binding.btnChange.setOnClickListener {
            if (isFragmentOneLoaded)
                showFragmentTwo()
            else showFragmentOne()
        }
    }

    private fun showFragmentOne() {
        // Se establece la transacción de fragments, necesarios para añadir,
        // quitar o reemplazar fragments.
        val transaction = supportFragmentManager.beginTransaction()
        // Se instancia el fragment a mostrar.
        val fragment = FragmentOne()

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

        // Se establece valor a null para indicar que no se está interesado
        // en volver a ese fragment más tarde, en caso contrario,
        // se indicaría el nombre del fragment, por ejemplo fragment.TAG,
        // aprovechando la variable creada en la clase.
        transaction.addToBackStack(null)

```

```
39          // Se muestra el fragment.
40          transaction.commit()
41          isFragmentOneLoaded = true
42      }
43
44      // Igual que showFragmentOne() pero para el segundo.
45      private fun showFragmentTwo() {
46          val transaction = supportFragmentManager.beginTransaction()
47          val fragment = FragmentTwo()
48
49          transaction.replace(R.id.fragment_holder, fragment)
50          transaction.addToBackStack(null)
51          transaction.commit()
52          isFragmentOneLoaded = false
53      }
54  }
```

El resultado que se obtendrá lo puedes ver en las siguientes imágenes, el *fragment* cambiará con cada pulsación del botón. En las pestañas *Logcat* y *Run* podrás observar la información producida por los *fragments*.

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 utilices o no entiendas para que sirve.

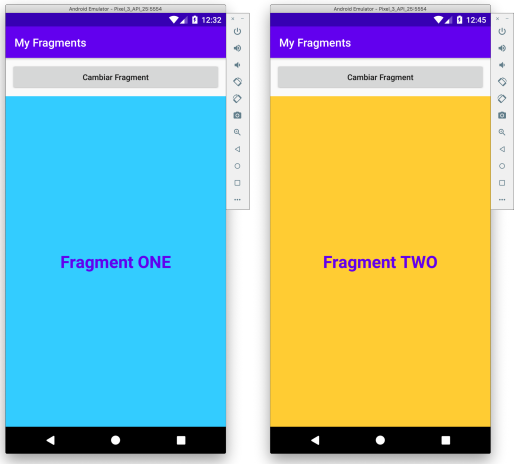

Figura 2

Ejercicios propuestos

7.1. Añade un tercer *fragment* a la aplicación de ejemplo que has reproducido para que, a cada pulsación del botón cambie del *FragmentOne* al *FragmentTwo* y luego al *FragmentThree* y vuelta a empezar.

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 *activity* al *fragment* que se desee mostrar.

Para este ejemplo, en un proyecto nuevo, se creará un único *fragment* como el que se muestra a continuación (`fragment_new.xml`), en este caso se ha utilizado la creación automática del *fragment*:

```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:id="@+id/frameRoot"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#CCFFDD"
    tools:context=".NewFragment">

    <TextView
        android:id="@+id/tv_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginStart="10dp"
        android:layout_marginTop="10dp"
        android:layout_marginEnd="10dp"
        android:layout_marginBottom="10dp"
        android:textAlignment="center"
        android:textSize="36sp"
        tools:text="Etiqueta para mostrar información" />
</FrameLayout>
```

A continuación, se crea la clase *Kotlin*, encargada 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()` .

```kotlin
class NewFragment : Fragment() {
    private lateinit var binding: FragmentNewBinding
    private var numFrag: Int? = null
    private var colorBack: Int? = null
    private val TAG = "FragmentNew - ${numFrag}"

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

    override fun onCreate(savedInstanceState: Bundle?) {
        Log.d(TAG, "onCreate")
        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 {
            numFrag = it.getInt(ARG_NUMFRAG)
```

```
21              colorBack = it.getInt(ARG_COLORBACK)
22          }
23      }
24
25      override fun onCreateView(
26          inflater: LayoutInflater, container: ViewGroup?,
27          savedInstanceState: Bundle?
28      ): View {
29          Log.d(TAG, "onCreateView")
30          binding = FragmentNewBinding.inflate(inflater)
31          return binding.root
32      }
33
34      companion object {
35          /**
36           * Use this factory method to create a new instance of
37           * this fragment using the provided parameters.
38           *
39           * @param nFrag Número de fragment.
40           * @param cBack Color de fondo.
41           * @return A new instance of fragment NewFragment.
42           */
43          @JvmStatic
44          fun newInstance(nFrag: Int, cBack: Int) =
45              NewFragment().apply {
46                  arguments = Bundle().apply {
47                      putInt(ARG_NUMFRAG, nFrag)
48                      putInt(ARG_COLORBACK, cBack)
49                  }
50              }
51      }
52
53      // Se modifican las propiedades en este método para asegurar que la activity
54      // está creada y se evitan así posibles fallos.
55      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
56          Log.d(TAG, "onViewCreated")
57          binding.frameRoot.setBackgroundColor(colorBack!!)
58          binding.tvFragment.text = "Fragment\n${numFrag}"
59          super.onViewCreated(view, savedInstanceState)
60      }
61
62      /* El resto de métodos sobrecargados son exactamente igual que los
63      utilizados en el ejemplo MyFragments. */
64      ...
65  }
```

En el método `onCreate()` se utiliza la propiedad `arguments` que es de tipo `Bundle`, esta es una variable de la clase `Fragment` que se encarga de recoger los parámetros que se le pasen a un *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 quieren pasar.

```kotlin
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.btnChange.setOnClickListener {
            showFragment()
        }
    }

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

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

        // Se crea la variable tipo Bundle para "empaquetar" los datos a pasar.
        val bundle = Bundle()
        bundle.putInt(ARG_NUMFRAG, ++numfrag)
        bundle.putInt(ARG_COLORBACK, (if ((numfrag % 2) == 0) RED else GREEN))

        fragment.arguments = bundle
        transaction.replace(R.id.fragment_holder, fragment)
        transaction.addToBackStack(null)
        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 es 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.

```
// Declaración del Fragment mediante newInstance.
val fragment = NewFragment.newInstance(
        ++numfrag,
        (if ((numfrag % 2) == 0) RED else GREEN)
)
```

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.

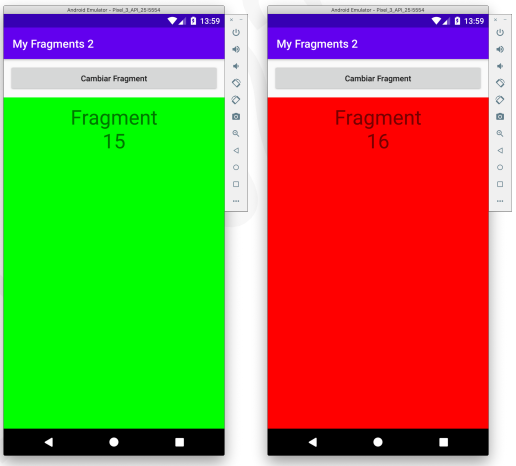

Figura 3

Ejercicios propuestos

7.2. Crea una aplicación, parecida al ejemplo, en la que una *activity* contenga un *fragment*, al cual deberás pasar la información que contenga un *EditText* en la propia *activity* al pulsar un botón, el *fragment* deberá actualizarse para mostrar esa información en un *TextView*.

7.3. Comunicación bidireccional entre *fragment* y *activity*

A continuación, modifica el ejemplo anterior para que puedan devolverse datos a la *activity* contenedora del *fragment*. En el ejemplo visto, se podía ver que pasar información de una *activity* 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.

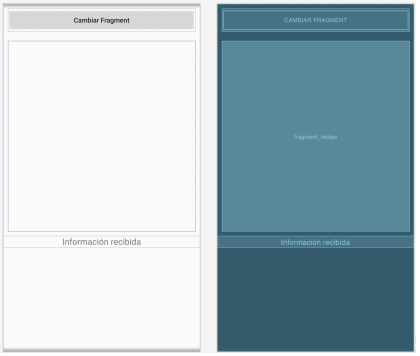

Figura 4

En primer lugar, se añadirá un *TextView* a la `activity_main.xml` para mostrar la información que se devuelva desde el *fragment*. También se ajusta la altura del *FragmentLayout* a 400dp y así poder tener visible la nueva etiqueta.

También deberás modificar `fragment_new.xml`, añadiendo un *TextView* y un *Button* al *layout* para poder enviar datos de vuelta a la actividad contenedora. El aspecto final debe ser parecido al que se muestra en la siguiente imagen.

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

También se deberá modificar el método `onAttach()` (este método se ejecutará cuando el *fragment* sea adjuntado a la *activity*) para notificar del error en caso de no implementarse la *interface*.

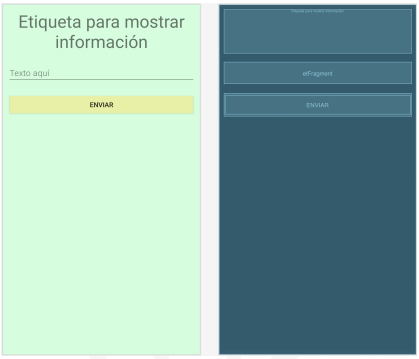

Figura 5

```
D/AndroidRuntime: Shutting down VM
E/AndroidRuntime: FATAL EXCEPTION: main
    Process: es.javiercarrasco.myfragments2, PID: 30964
    java.lang.RuntimeException: Unable to start activity ComponentInfo{es.javiercarrasco.myfragments2/es.javiercarrasco.myfragments2.MainActivity}: java.lang.RuntimeException: es.javiercarrasco.myfragments2.MainActivity@ac96be8 debe implementar DatoDevuelto.
```

Figura 6

```kotlin
class NewFragment : Fragment() {
    ...
    interface DatoDevuleto {
        fun datoActualizado(dato: String)
    }

    private var datoFragment: DatoDevuleto? = null

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

        if (context is DatoDevuleto)
            datoFragment = context
        else throw RuntimeException(
            requireContext.toString() + " debe implementar DatoDevuelto."
        )
    }
```

También deberá añadirse a la clase `NewFragment.kt` el método que se encargará de pasar la información a la *activity* que lo contiene.

```kotlin
// Método encargado de pasar la información a la activity contenedora.
fun actualizarDato() {
    if (!binding.etFragment.text.isEmpty())
        datoFragment?.datoActualizado(binding.etFragment.text.toString())
    else datoFragment?.datoActualizado("Sin información")
}
```

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

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    Log.d(TAG, "onViewCreated")
    binding.frameRoot.setBackgroundColor(colorBack!!)
    binding.tvFragment.text = "Fragment\n${numFrag}"

    binding.btnFragment.setOnClickListener {
        actualizarDato()
    }
    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()
    datoFragment = null
}
```

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

```
class MainActivity : AppCompatActivity(), NewFragment.DatoDevuleto
```

Y se sobrecargará el método `datoActualizado()`.

```
override fun datoActualizado(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.

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

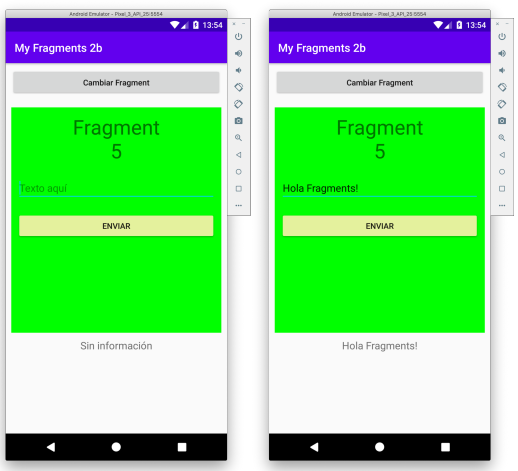

Figura 7

1 My Fragments 2 (https://play.google.com/store/apps/details?id=es.javiercarrasco.myfragments2&gl=ES)

Ejercicios propuestos

7.3. Crea una aplicación, utilizando **Android Studio+Kotlin**, en la que la actividad principal contenga un *TextView*, un botón y dos *fragments*. El ***fragment 1*** contendrá un *EditText* y un botón, el cual pasará la información del *EditText* a la actividad principal, la cual mostrará la información en el *TextView*, recibida la información, actualizará el ***fragment 2*** para que esa misma información aparezca en un *TextView* del propio *fragment 2*. Un intercambio de información entre *fragments* pasando por la actividad que los contiene.

7.4. Pestañas/Tabs + ViewPager

Las pestañas, o *tabs*, permiten mostrar información de una manera más ordenada. Sistema muy utilizado actualmente en aplicaciones como *WhatsApp*. A continuación se verá un ejemplo basándonos en las recomendaciones de *Material Design*[2].

Partiendo de un nuevo proyecto, se comenzará por modificar el tema de la aplicación. En el fichero `themes.xml` y se cambiará el valor de *parent* del tema base de la aplicación por uno que permita desactivar la barra de acción.

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.MyTabs"
        parent="Theme.MaterialComponents.DayNight.NoActionBar">
    ...
```

A continuación, en la actividad `activity_main.xml` se añadirá el código que permitirá cargar las pestañas y una nueva barra de aplicación.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.appbar.AppBarLayout
        android:id="@+id/appbarLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_scrollFlags="scroll|enterAlways">
        </androidx.appcompat.widget.Toolbar>
```

2 *Material Design* (https://material.io/components/tabs/)

```
        <com.google.android.material.tabs.TabLayout
            android:id="@+id/tabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:tabMode="scrollable" />
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.viewpager.widget.ViewPager
        android:id="@+id/viewPager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/appbarLayout" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Como puedes ver, se parte de un *ConstraintLayout* como elemento raíz de la vista. Se añade una *AppBarLayout* utilizando *material* y dentro se añade una nueva *ToolBar* y las pestañas, *TabLayout* también de *material*. Por último se añade un *ViewPager* que permitirá mostrar la información de cada pestaña, para lo que se utilizarán *Fragments*. Para este ejemplo se utiliza la versión 1 de *ViewPager*.

Ahora, como se van a mostrar tres pestañas, se deberán crear tres *fragments*, uno por cada pestaña (personas, frutas y animales), y cada uno de ellos contendrá la información a mostrar.

fragment_persons.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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"
    tools:context=".PersonsFragment">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/text_persons"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

PersonsFragment.kt

```
class PersonsFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
```

```
        savedInstanceState: Bundle?
    ): View? {
        Log.d("Personas", "onCreateView")
        return inflater.inflate(R.layout.fragment_persons, container, false)
    }
}
```

Los otros dos *fragments* serán exactamente igual. Ahora, se necesitará conectar los *fragments* con el *ViewPager*[3], este componente permite pasar páginas de izquierda a derecha, y viceversa. Para esta conexión se creará la clase *ViewPagerAdapter* que implementará *FragmentStatePagerAdapter*. A esta clase se le pasará una lista con los *fragments* a instanciar y los títulos de las pestañas. Además, se sobrecargarán los métodos que ayudarán a obtener el *fragment* a mostrar, el número de pestañas, o *fragments*, y el título de la pestaña.

```
class ViewPagerAdapter(supportFragmentManager: FragmentManager) :
    FragmentStatePagerAdapter(
        supportFragmentManager,
        BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT
    ) {

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

    override fun getItem(position: Int): Fragment {
        return mFragmentList[position]
    }

    override fun getCount(): Int {
        return mFragmentList.size
    }

    override fun getPageTitle(position: Int): CharSequence? {
        return mFragmentTitleList[position]
    }

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

Además se creará el método `addFragment()` en el *ViewPagerAdapter* para añadir *fragments*, y por ende, *tabs*.

La clase `FragementStatePagerAdapter` 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. Veamos ahora como quedaría la clase principal para poder mostrar el contenido en el *ViewPager* y tener unas pestañas completamente funcionales.

3 *ViewPager* (https://developer.android.com/reference/kotlin/androidx/viewpager/widget/ViewPager.html)

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

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

        // Se carga la toolbar.
        setSupportActionBar(binding.toolbar)

        // Se crea el adapter.
        val adapter = ViewPagerAdapter(supportFragmentManager)

        // Se añaden los fragments y los títulos de pestañas.
        adapter.addFragment(PersonsFragment(), "Personas")
        adapter.addFragment(FruitsFragment(), "Frutas")
        adapter.addFragment(AnimalsFragment(), "Animales")

        // Se asocia el adapter.
        binding.viewPager.adapter = adapter

        // Se cargan las tabs.
        binding.tabs.setupWithViewPager(binding.viewPager)
    }
}
```

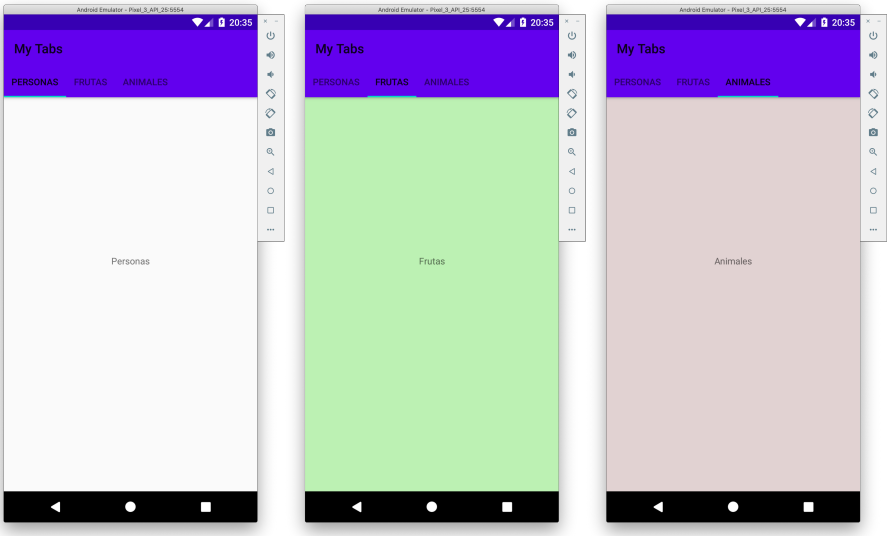

Figura 8

A continuación se verá un cambio de imagen para las *tabs*, si quieres añadir un color de fondo diferente a la pestaña seleccionada, deben realizarse una serie de modificaciones en los ficheros `res`.

El primer paso consistirá en añadir el color deseado para la pestaña seleccionada, en el fichero `colors.xml` se añadirá la siguiente línea, elige tu propio color.

```
1   <color name="tabSelected">#FF64b5f6</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.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <selector xmlns:android="http://schemas.android.com/apk/res/android">
3      <item android:drawable="@color/tabSelected" android:state_selected="true"/>
4      <item android:drawable="@color/blue_200"/>
5   </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.

Ejercicios propuestos

7.4. Sobre el código de ejemplo visto en este punto, haz que aparezca, en cada *fragment* su correspondiente lista. Un listado de personas, un listado de frutas y un listado de animales.

7.5. ViewPager2

Habrás comprobado que algunos métodos del *ViewPager* aparecen marcados como *deprecated*, en 2019 Google lanzó ***ViewPager2***[4], que varía ligeramente con respecto a la primera versión, pero que recomienda su uso desde entonces. A continuación, se verá el ejemplo anterior, adaptado a la nueva versión de *ViewPager*.

4 ViewPager2 (https://developer.android.com/reference/kotlin/androidx/viewpager2/widget/ViewPager2)

Lo único que no se modificará para esta **segunda versión** son los *fragments*, aprovechando los ya creados en la primera versión, tanto los *layouts* como las clases. Tampoco se eliminará la barra de acción del tema, dejando la que viene por defecto, esto se debe a que para añadir pestañas se añadirá un *TabLayout* directamente a la vista, sin necesidad de modificar la barra. El *layout* `activity_main.xml` quedará como se muestra a continuación.

```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:orientation="vertical">

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

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

El siguiente paso será crear el adaptador del *ViewPager2*, muy parecido al visto en la versión anterior, se seguirá aprovechando el adaptador para llevar el control de la pestaña asociada al *fragment*, de otra manera debería hacerse desde fuera. De esta forma, todo lo relacionado con las pestañas y las páginas está junto.

```
class ViewPager2Adapter(
    fragmentManager: FragmentManager,
    lifecycle: Lifecycle
) : FragmentStateAdapter(fragmentManager, lifecycle) {

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

    override fun getItemCount(): Int {
        return mFragmentList.size
    }
    override fun createFragment(position: Int): Fragment {
        return mFragmentList[position]
    }
    fun addFragment(fragment: Fragment, title: String) {
        mFragmentList.add(fragment)
        mFragmentTitleList.add(title)
    }
    fun getPageTitle(position: Int): CharSequence {
        return mFragmentTitleList[position]
    }
}
```

Esta nueva versión, además de los parámetros que se le pasan, cambia dos métodos, `getCount()` por `getItemCount()`, y `getItem()` por `createFragment()`. Pero si comparas el funcionamiento, son muy similares. A continuación, se muestra la nueva clase principal, que difiere ligeramente de la primera versión.

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

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

        val viewPager2 = binding.viewPager2

        // Se crea el adapter.
        val adapter = ViewPager2Adapter(supportFragmentManager, lifecycle)

        // Se añaden los fragments y los títulos de pestañas.
        adapter.addFragment(PersonsFragment(), "Personas")
        adapter.addFragment(FruitsFragment(), "Frutas")
        adapter.addFragment(AnimalsFragment(), "Animales")

        // Se asocia el adapter al ViewPager2.
        viewPager2.adapter = adapter

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

        // Carga de las pestañas en el TabLayout.
        TabLayoutMediator(binding.tabLayout, viewPager2) { tab, position ->
            tab.text = adapter.getPageTitle(position)
        }.attach()
    }
}
```

Observa que es bastante similar al anterior, salvo que se elimina la carga de la *Toolbar*, cambia la creación del *adapter*, al que se le añade el segundo parámetro, el ciclo de vida de la aplicación, y se añade, por un lado, la posibilidad de añadir un efecto al *ViewPager2*, algo no se estaba disponible en la versión anterior, y se cambia el sistema de creación de las pestañas, ahora se hará mediante el uso del método `TabLayoutMediator()`[5].

Con respecto al efecto[6], se utiliza el método `setPageTransformer()`, al cual se le debe pasar un objeto de tipo `ViewPager2.PageTransformer`. Para este ejemplo se ha creado la clase `DepthPageTransformer` con el código necesario para poder realizar el efecto de profundidad, este lo puedes encontrar en la documentación oficial de Android, también puedes encontrar el efecto zoom[7].

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)

```kotlin
@RequiresApi(21)
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
                    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
                }
            }
        }
    }
}
```

En este ejemplo, al igual que el anterior visto, se crean tres *fragments* con sus tres pestañas, pero puede darse el caso que se necesiten crear los *fragments* y pestañas de forma dinámica.

Creación dinámica de *fragments* y *tabs*

Para esta **tercera versión**, el `activity_main.xml` será igual al visto en la segunda versión, pero se creará únicamente un solo *fragment*.

fragment_page.xml

```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">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAlignment="center"
        android:textAppearance="@style/TextAppearance.AppCompat.Display1"
        tools:text="Título fragment" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginStart="5dp"
        android:layout_marginEnd="5dp"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1"
        tools:text="Texto a mostrar" />
</LinearLayout>
```

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

```kotlin
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, texto: String) =
            PageFragment().apply {
                arguments = Bundle().apply {
                    putString("name", name)
                    putString("texto", texto)
                }
```

```
            }
    }

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

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

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        binding.textView.text = nombre
        binding.textView2.text = 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.

```
private const val NUM_FRAGMENTS = 5 // Número de fragments a crear.

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

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        val viewPager2 = binding.viewPager2

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

        // Se crea el adapter.
        val adapter = ViewPager2Adapter(supportFragmentManager, lifecycle)

        // Se añaden los fragments y los títulos de pestañas.
        for (num in 1..NUM_FRAGMENTS) {
            val fragment: PageFragment = PageFragment()
            val bundle = Bundle()
```

```
            bundle.putString("name", "Fragment ${num}")
            bundle.putString("texto", getString(R.string.textAMostrar, num))
            fragment.arguments = bundle
            adapter.addFragment(fragment, "Frg ${num}")
        }

        // Se asocia el adapter al ViewPager2.
        viewPager2.adapter = adapter

        // Efectos para el ViewPager2.
        viewPager2.setPageTransformer(ZoomOutPageTransformer())

        // Carga de las pestañas en el TabLayout.
        TabLayoutMediator(binding.tabLayout, viewPager2) { tab, position ->
            tab.text = adapter.getPageTitle(position)
        }.attach()
    }
}
```

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` de *ViewPager2*, esta permite elegir el tipo de desplazamiento de las páginas, vertical u horizontal. La clase `ViewPager2Adapter` quedará tal y como se vio en el ejemplo anterior. El resultado lo puedes ver en la figura que se muestra a continuación.

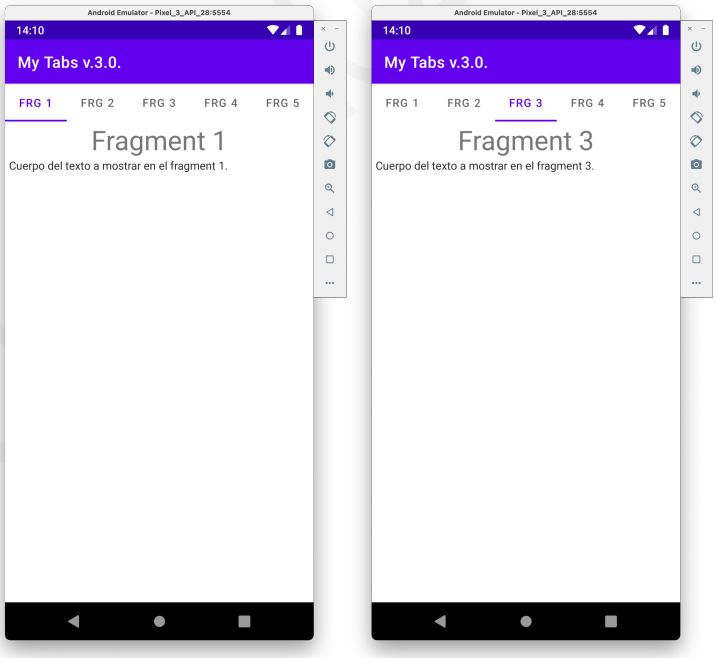

Figura 9

7.6. Navigation

Ahora que ya conoces el uso de *fragments* y *activities*, veamos 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, permitiendo hacer navegaciones sencillas, hasta patrones más complejos como barras 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:2.3.5"
implementation "androidx.navigation:navigation-ui-ktx:2.3.5"

// Feature module Support
implementation "androidx.navigation:navigation-dynamic-features-fragment:2.3.5"

// Testing Navigation
androidTestImplementation "androidx.navigation:navigation-testing:2.3.5"

// Jetpack Compose Integration
implementation "androidx.navigation:navigation-compose:1.0.0-alpha08"
```

Durante la revisión del libro de 2021, la versión del *Jetpack Compose Integration* era la *2.4.0-alpha04*, pero tenía un bug que no permitía el paso de parámetros mediante SafeArgs, de ahí que se muestre el ejemplo con la versión anterior.

Además, se recomienda añadir el *plug-in* ***androidx.navigation.safeargs.kotlin*** en el fichero `build.gradle(Module: app)` para añadir seguridad de tipo al navegar y pasar datos entre los destinos.

Por último, antes de comenzar, para terminar de configurar *Safe Args*[9], se añadirá la siguiente línea en las dependencias del fichero `build.gradle(Project)`.

8 Navigation (https://developer.android.com/guide/navigation)
9 Safe Args (https://developer.android.com/guide/navigation/navigation-pass-data#Safe-args)

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.3.5"
```

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

Antes de poder añadir el componente *fragment*, 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.

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*.

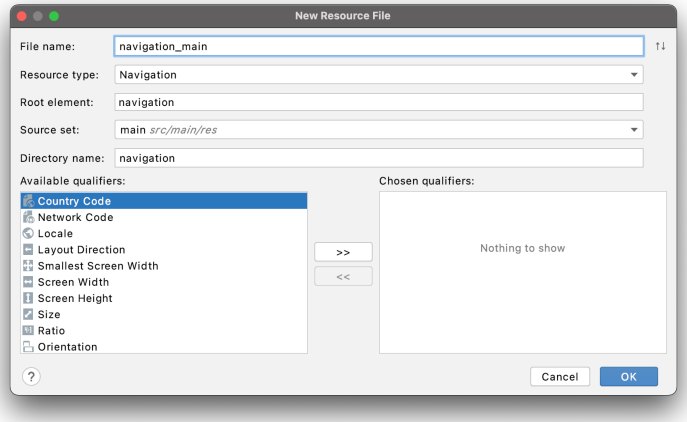

Figura 10

Seguidamente, en la `activity_main.xml` deberás añadir el componente *fragment* (<fragment>) y preguntará que *fragment* añadir, para este caso, selecciona el uno. El resultado de la *activity* puede ser como se muestra a continuación.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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:id="@+id/coordinatorLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <fragment
        android:id="@+id/fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="0dp"
```

```
14          android:layout_height="0dp"
15          app:defaultNavHost="true"
16          app:layout_constraintBottom_toBottomOf="parent"
17          app:layout_constraintEnd_toEndOf="parent"
18          app:layout_constraintStart_toStartOf="parent"
19          app:layout_constraintTop_toTopOf="parent"
20          app:navGraph="@navigation/navigation_main"
21          tools:layout="@layout/fragment_uno" />
22  </androidx.constraintlayout.widget.ConstraintLayout>
```

La propiedad `tools:layout` permite ver una vista previa del *fragment* asociado en la vista diseño, lo que puede facilitar mucho esta tarea.

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.

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.

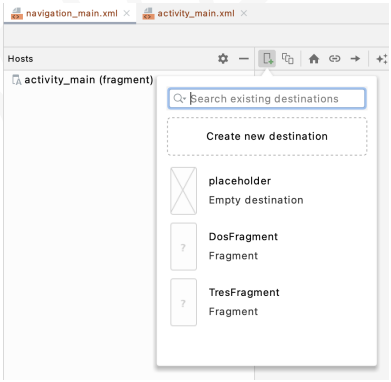

Figura 11

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

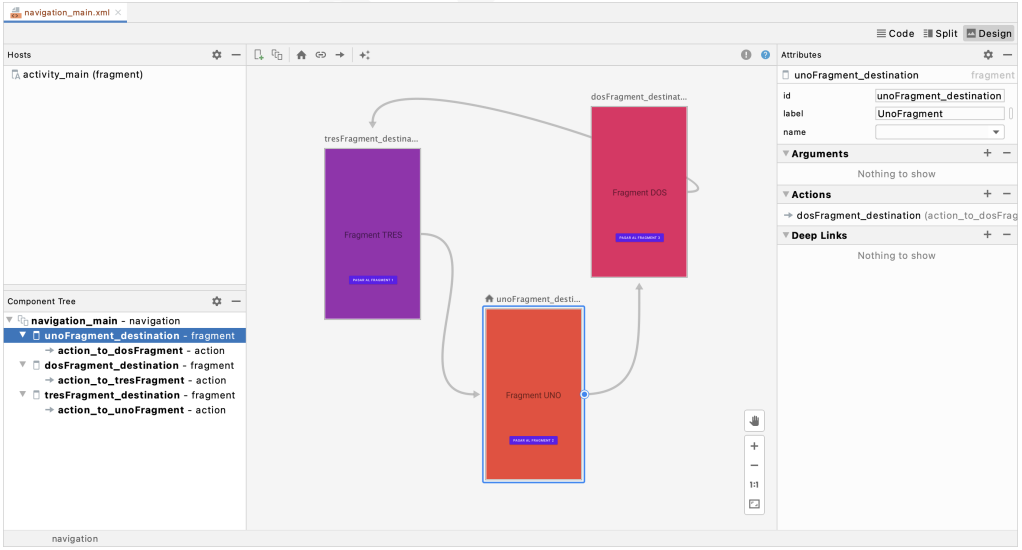

Figura 12

Para crear las acciones fácilmente, 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()` se añadirá el siguiente código después de instanciar la variable `binding`.

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

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 animaciones que vienen por defecto, como *slide in left* y *slide out right*.

Seguidamente, 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.

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.

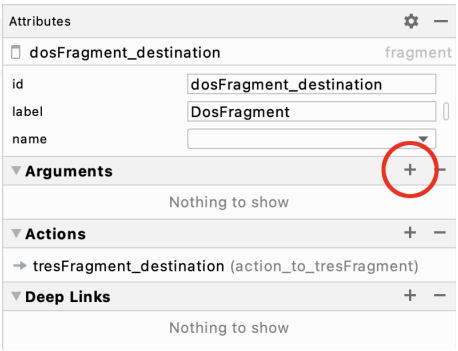

Figura 13

Una vez hecho, si seleccionas las acciones (las flechas), verás como aparece en en atributos, en la sección argumentos, 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.actionToDosFragment(
            getString(
                R.string.txtFragment,
                "UNO"
            )
        )
    )
}
```

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 la forma del sistema para pasar los parámetros de un destino a otro, como verás este sistema, es mucho más sencillo que el visto anteriormente.

```
1  if (UnoFragmentArgs.fromBundle(requireArguments()).numFragment != "nulo")
2      binding.textView.text = UnoFragmentArgs.fromBundle(
3          requireArguments()
4      ).numFragment
5  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.