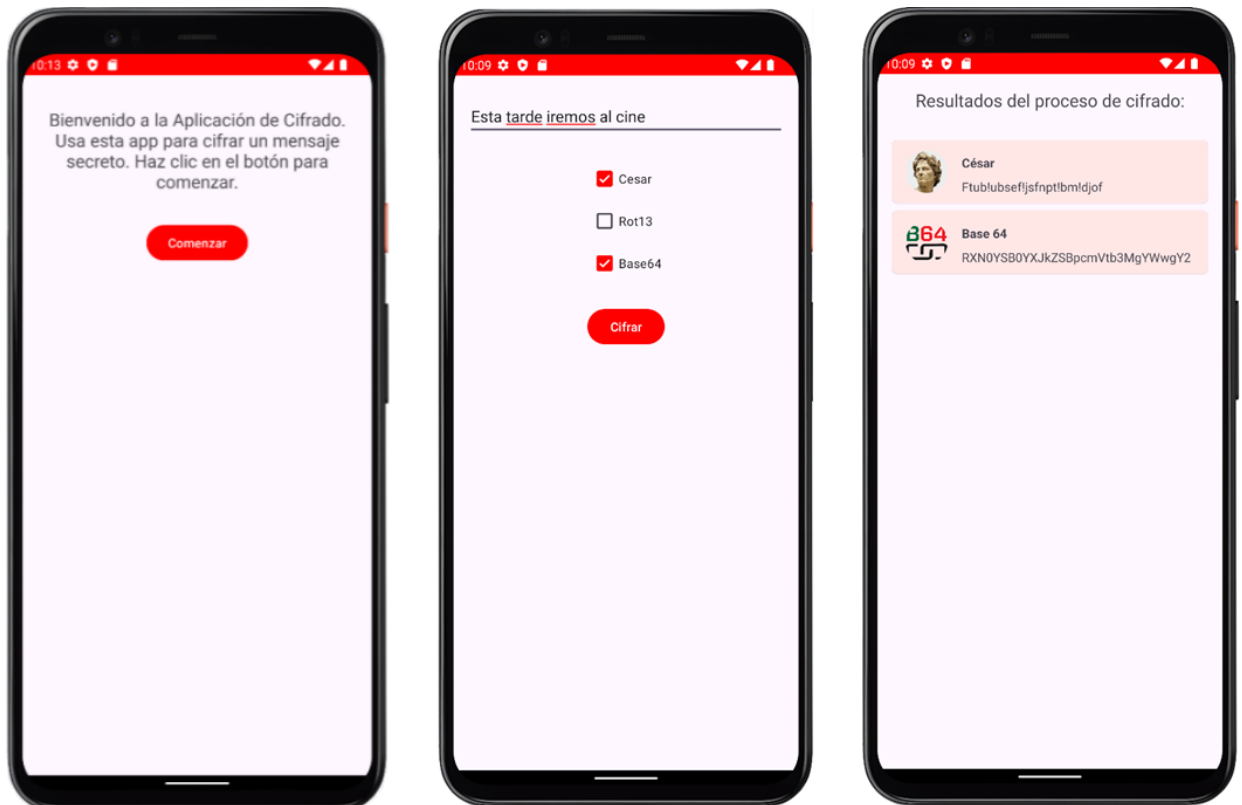


PROYECTO 6

CIFRADOR



En este proyecto haremos una app que mostrará una pantalla de bienvenida al usuario, que dará paso a una nueva pantalla en la que se pedirá al usuario que escriba un texto, que será pasado a una pantalla final donde se mostrará cifrado. El texto cifrado se podrá enviar a otra persona mediante sms, correo electrónico, etc

Durante su desarrollo se tratarán estos conceptos:

- Concepto de navegación
- El Navigation component
- El navigation graph
- Acciones en el navigation graph
- El NavHostFragment
- El NavigationController
- Paso de información entre Fragments
- Arquitecturas MVC y MVVM
- Gestión del botón de atrás
- Animaciones a la navegación
- Implicit Intent

1 – Concepto de navegación

La app que vamos a hacer está formada por tres pantallas que se muestran de forma sucesiva: bienvenida, introducción de datos y resultado de la app. Podríamos hacer tres **Activity**, de forma que se fuesen mostrando una tras otra de esa forma, pero esto está desaconsejado en el desarrollo moderno para Android.

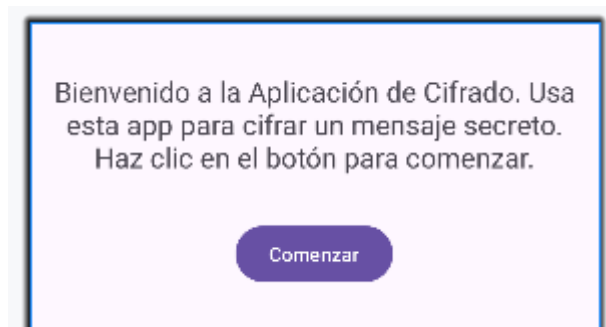
Actualmente se prefiere una sola **Activity** que aloje en su interior un **FragmentManager** que va a ir mostrando el **Fragment** apropiado según se vaya navegando por la app.

- Abre Android Studio y crea un proyecto llamado **Cifrador**, usando la plantilla **Empty Views Activity**
- Crea tres paquetes llamados **modelo**, **vista** y **viewmodel**
- Pulsa el botón derecho del ratón sobre el paquete **vista** y elige **new** → **Fragment** → **Fragment (Blank)**
- En la pantalla que aparece rellena estos datos:
 - o Nombre del fragment: **BienvenidaFragment**
 - o Archivo de layout del fragment: **fragment_bienvenida**
 - o Lenguaje: **Kotlin**
- Se abrirá una ventana con el código fuente del archivo **BienvenidaFragment**, pero el código que nos genera Android Studio está obsoleto. Borra todo lo necesario hasta que el código se quede así:

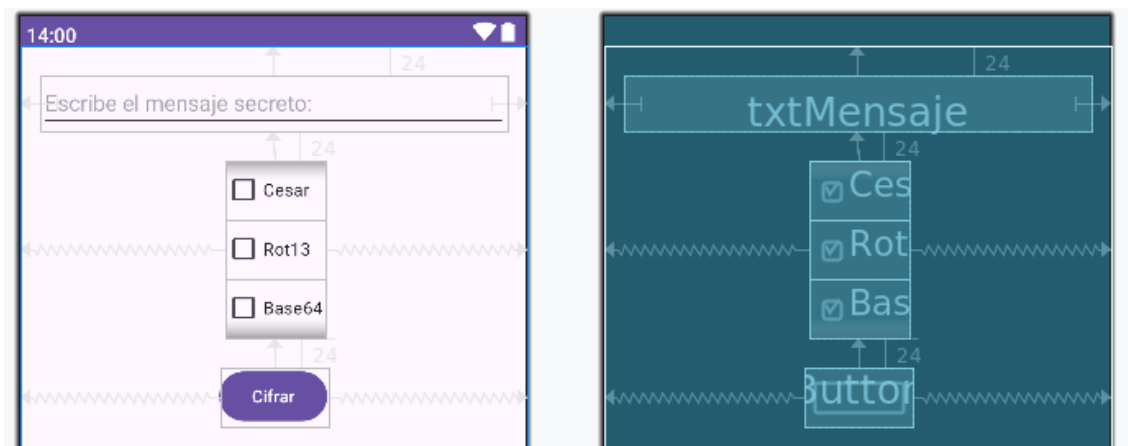
```
1. class BienvenidaFragment : Fragment() {  
2.     override fun onCreateView(  
3.         inflater: LayoutInflater, container: ViewGroup?,  
4.         savedInstanceState: Bundle?  
5.     ): View? {  
6.         // Inflate the layout for this fragment  
7.         return inflater.inflate(R.layout.fragment_bienvenida, container, false)  
8.     }  
9. }
```

- Abre **fragment_bienvenida.xml** con el diseñador de Android Studio y borra el **TextView** que aparece por defecto
- En el **component tree** selecciona el **FrameLayout** que aparece por defecto, pulsa el botón derecho sobre él y **convert view**
- En la ventana que aparece, elige **LinearLayout**
- Selecciona el **LinearLayout** y en sus atributos ponle:
 - o Orientación: **vertical**
 - o Gravity: **center_horizontal**
 - o Padding: **16dp**
- Añade a la interfaz un **TextView** y un **Button** con las características indicadas, para que se quede todo como aparece en la imagen:
 - o **TextView**: su id es **txtBienvenida**, tiene texto con alineación centrada, tamaño de la letra **20sp** y separación superior **20dp**

- **Button:** su id es **btnComenzar**, y tiene separación superior de **32dp**



- Añade al paquete **vista** un nuevo **Fragment** llamado **MensajeFragment** y modifica su código para eliminar la parte obsoleta, tal y como hemos visto
- Abre el archivo **fragment_mensaje.xml** con el diseñador, borra lo que hay y cambia el elemento principal por un **ConstraintLayout**
- Arrastra los siguientes elementos a la interfaz, para que todo quede como se ve en la imagen:
 - **EditText** (aparece como **Multiline Text** en el editor): su id es **txtMensaje**, su distancia al borde superior es de **24dp** y **16dp** a los bordes de la pantalla.
 - **LinearLayout (Vertical)**: su id será **grupoCasillas** y su altura y anchura se ajustará a su contenido. Tendrá una distancia a **txtMensaje** de **24dp** y estará centrado en la pantalla.
 - **CheckBox**: se añadirán 3 **casillas de verificación** a **grupoCasillas** y se les pondrá como id **chkCifrado1**, **chkCifrado2** y **chkCifrado3**
 - **Button**: tendrá como id **btnCifrar** y una separación de **24dp** a **grupoCasillas** y estará centrado en la pantalla.



- A continuación, añade en el paquete **vista** un nuevo **Fragment** llamado **ResultadoFragment** y borra el código fuente obsoleto.
- Pon como elemento principal un **LinearLayout** de orientación vertical y **16dp** de padding.

- Arrastra a la interfaz:
 - Un **TextView** con id **txtResultado**, **20sp** de tamaño de texto y alineación centrada.
 - Un **RecyclerView** con disposición lineal y estas características
 - Id: **lstResultados**
 - Anchura y anchura: la de su contenedor
 - Margen por encima: 32dp
- Abre el archivo **build.gradle.kts**, habilita el **view binding**

```

1. // resto del archivo omitido
2. android {
3.     buildFeatures{
4.         viewBinding=true
5.     }
6. }

```

- Pon una variable de instancia **binding** en cada uno de los **Fragment** y coloca el métodos **inicializarBinding** en ellos, llamándolo en **onCreateView**. Puedes optar por dos opciones:
 - La forma más sencilla, pero propensa a errores en apps grandes:

```

1. class BienvenidaFragment : Fragment() {
2.     private lateinit var binding: FragmentBienvenidaBinding
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarBinding(inflater,container)
8.         return binding.root
9.     }
10.    private fun inicializarBinding(inflater: LayoutInflater,container: ViewGroup?){
11.        binding= FragmentBienvenidaBinding.inflate(inflater,container,false)
12.    }
13. }

```

- Una forma más compleja, que a cambio, nos advertirá si estamos usando los componentes de la interfaz en un momento en el que el **Fragment** o está en la pantalla.

```

1. class MensajeFragment : Fragment() {
2.     private var _binding: FragmentMensajeBinding?=null
3.     private val binding: FragmentMensajeBinding
4.         get() = checkNotNull(_binding){"uso incorrecto del objeto binding"}
5.     override fun onCreateView(
6.         inflater: LayoutInflater, container: ViewGroup?,
7.         savedInstanceState: Bundle?
8.     ): View? {
9.         inicializarBinding(inflater,container)
10.        return binding.root
11.    }
12.    private fun inicializarBinding(inflater:LayoutInflater,container: ViewGroup?){
13.        _binding=FragmentMensajeBinding.inflate(inflater,container,false)
14.    }
15.    override fun onDestroyView() {
16.        super.onDestroyView()
17.        _binding=null
18.    }
19. }

```

- En el paquete **viewModel** crea las clases **MensajeFragmentViewModel** y **ResultadoFragmentViewModel**, haciendo que cada una herede de **ViewModel**

```

1. // archivo MensajeFragmentViewModel.kt
2. class MensajeFragmentViewModel : ViewModel() {
3. }
4. // archivo ResultadoFragmentViewModel.kt
5. class ResultadoFragmentViewModel : ViewModel() {
6. }

```

- En **MensajeFragment** y **ResultadoFragment** añade una variable de instancia **viewModel** del tipo correspondiente, y haz un método **inicializarViewModel** que los inicialice:

```

1. // archivo MensajeFragment.kt
2. class MensajeFragment : Fragment() {
3.     private lateinit var viewModel: MensajeFragmentViewModel
4.     private var _binding: FragmentMensajeBinding?=null
5.     private val binding: FragmentMensajeBinding
6.         get() = checkNotNull(_binding){"uso incorrecto del objeto binding"}
7.     override fun onCreateView(
8.         inflater: LayoutInflater, container: ViewGroup?,
9.         savedInstanceState: Bundle?
10.    ): View? {
11.        inicializarViewModel()
12.        inicializarBinding(inflater,container)
13.        return binding.root
14.    }
15.    private fun inicializarViewModel(){
16.        viewModel = ViewModelProvider(this).get(MensajeFragmentViewModel::class.java)
17.    }
18.    // resto omitido
19. }
20. // archivo ResultadoFragment.kt
21. class ResultadoFragment : Fragment() {
22.     private lateinit var viewModel: ResultadoFragmentViewModel
23.     private var _binding: FragmentResultadoBinding?=null
24.     private val binding: FragmentResultadoBinding
25.         get()= checkNotNull(_binding){"uso incorrecto del objeto binding"}
26.     override fun onCreateView(
27.         inflater: LayoutInflater, container: ViewGroup?,
28.         savedInstanceState: Bundle?
29.    ): View? {
30.        inicializarViewModel()
31.        inicializarBinding(inflater,container)
32.        return binding.root
33.    }
34.    fun inicializarViewModel(){
35.        viewModel = ViewModelProvider(this).get(ResultadoFragmentViewModel::class.java)
36.    }
37.    // resto omitido
38. }

```

2 – El Navigation component

Google ha creado dos librerías muy útiles para trabajar con **Fragments**:

- **Navigation component**: nos permite navegar fácilmente entre **Fragments**.
- **Safe-args plugin**: Nos facilita pasar información de un **Fragment** a otro

Ambos deben ser incorporados al proyecto añadiéndolos en los archivos **build.gradle.kts**

- Abre el archivo **build.gradle.kts** (versión **Project**) y añade a continuación de lo que haya escrito, el siguiente bloque:

```
1. buildscript {
2.     repositories {
3.         google()
4.     }
5.     dependencies {
6.         classpath("androidx.navigation:navigation-safe-args-gradle-plugin:2.8.4")
7.     }
8. }
```

- Abre el archivo **build.gradle.kts** (versión **Module**) y añade, dentro del bloque **plugins**, la siguiente línea:

```
1. plugins {
2.     alias(libs.plugins.android.application)
3.     alias(libs.plugins.kotlin.android)
4.     id("androidx.navigation.safeargs.kotlin")
5. }
```

- En el mismo archivo, añade en la zona **dependencies** la siguiente línea:

```
implementation("androidx.navigation:navigation-fragment-ktx:2.8.4")
```

- Pulsa el botón **Sync Now** para hacer efectivos los cambios en el archivo **build.gradle.kts**, y verás que la librería se descargará automáticamente.


3 – El navigation graph

El primer paso para poder navegar entre **Fragments** consiste en la creación de un **navigation graph**, que es un gráfico en el que aparecen todos los **Fragment** de nuestra app unidos con flechas (llamadas **acciones**) que indican las transiciones que se pueden hacer entre ellos.

- Pulsa el botón derecho en la carpeta **res** y después **new** → **android resource file**
- Rellena la ventana que aparece con esta información:
 - o Nombre del archivo: **nav_graph.xml**
 - o Tipo de recurso: **Navigation**

Se abre una ventana donde vamos a añadir de forma gráfica los **Fragments** de nuestra app y los vamos a conectar por líneas que nos dicen desde dónde se puede ir hacia donde.

El **navigation graph** es tan potente que no solo permite la navegación entre **Fragments**. En general, se permite la navegación entre objetos denominados **Destinations**, entre los que se encuentran los **Fragments**.

- Pulsa el botón  para añadir un **Fragment** al gráfico.
- En la ventana que se muestra, elige **BienvenidaFragment**

El primer **Fragment** que se añade es el que se verá primero en la pantalla. Eso se nota porque tiene el símbolo 🏠 en el gráfico.

Cuando tenemos muchos **Fragment**, es posible cambiar el **Fragment** de inicio abriendo **nav_graph.xml** en la vista **xml** y poniendo el **Fragment** de inicio en el atributo **app:startDestination**

- En **bienvenidaFragment** (o cualquier otro **Fragment**) hay veces que pone *Preview Unavailable*. Si es así, para hacer que se muestre la interfaz del **Fragment**, abre la vista de código fuente y en la etiqueta **fragment** añade el atributo **tools:layout** indicando el archivo **xml** del **Fragment**, así:

```
1. <fragment
2.     android:id="@+id/bienvenidaFragment"
3.     android:name="dam.moviles.proyecto4.BienvenidaFragment"
4.     tools:layout="@layout/fragment_bienvenida"
5.     android:label="BienvenidaFragment" />
```

- Observa que el **Fragment** añadido tiene un **id** en el gráfico

id bienvenidaFragment

El **id** del **Fragment** en este gráfico, se utilizará para navegar a dicho **Fragment** cuando tengamos menús con opciones para navegar (esto no se verá en este proyecto, sino en otros posteriores).

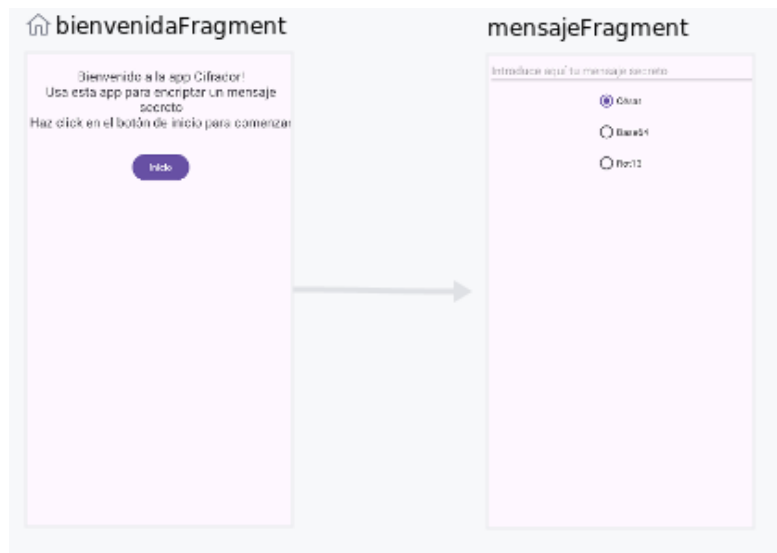
- Añade al gráfico los otros dos fragments y haz que se muestre su previsualización correctamente.

4 – Acciones en el navigation graph

Una **acción (Action)** es una transición de un **Fragment** a otro dentro del **navigation graph**.

Cada **action** tiene un **id**, que nos servirá para lanzar dicha acción en el código fuente y hacer que la interfaz navegue desde un **Fragment** hacia otro

- En el **navigation graph** pasa el ratón por encima de **bienvenidaFragment** y observa que hay un punto gordo a su derecha.
- Pulsa el ratón en dicho punto gordo y arrastra la flecha que aparece hasta el **messageFragment**



La flecha creada es un **action** que permite navegar desde **bienvenidaFragment** hacia **mensajeFragment**.

Tiene un **id** llamado **action_bienvenidaFragment_to_mensajeFragment**

- Crea una acción que permita navegar desde **mensajeFragment** hasta **resultadoFragment** y observa el **id** que Android Studio le ha puesto a esa acción.

5 – El NavHostFragment

Hasta ahora no hemos tocado **MainActivity**. Si recordamos los proyectos anteriores, los **Fragment** se muestran en un **FragmentContainerView** donde ponemos el **Fragment** que queremos ver.

Comenzaremos añadiendo a **MainActivity** un **FragmentContainerView**, pero en esta ocasión, no le pondremos ninguno de los **Fragment** que hemos hecho, sino que le pondremos un **Fragment** especial que viene con Android, llamado **NavHostFragment**.

El **NavHostFragment** es un **Fragment** que usa el **navigation graph** para navegar entre **Fragment** mediante las **actions** que tiene definidas.

- Abre el archivo **activity_main.xml** y borra todo lo que tiene
- Cambia su elemento raíz para que sea un **FragmentContainerView** (*deberás escribir ese nombre en el buscador, ya que no sale en las opciones predefinidas*)
- Abre **activity_main.xml** con la vista de código fuente
- Añade estos atributos al **FragmentContainerView**:
 - o **android:name** → **androidx.navigation.fragment.NavHostFragment**
 - o **app:navGraph** → **@navigation/nav_graph**
 - o **app:defaultNavHost** → **true**


```

1. <androidx.fragment.app.FragmentContainerView
2.     android:id="@+id/fragment_container_view"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent"
5.     android:name="androidx.navigation.fragment.NavHostFragment"
6.     app:navGraph="@navigation/nav_graph"
7.     app:defaultNavHost="true"/>

```

El **NavHostFragment** es un **Fragment** que permite la navegación entre **Fragment**. Los atributos que hay que configurar son:

- **app:name** → Se pondrá **androidx.navigation.fragment.NavHostFragment**
- **app:navGraph** → Es el **navigation graph** que se usará para navegar
- **app:defaultNavHost** → Vale **true** si queremos que el botón de **atrás** del dispositivo permita la vuelta al **Fragment** previo del **navigation graph**

- Si ponemos la app en marcha, veremos que se muestra correctamente el **BienvenidaFragment**, pero al pulsar el botón no pasa nada. Eso se debe a que debemos hacer que al pulsarlo, se active el **action** que nos permite pasar de **BienvenidaFragment** a **MensajeFragment**.

6 – El Navigation Controller

El **Navigation Controller** es el objeto que nos permite pasar de un **Fragment** a otro, de forma que el **FragmentContainerView** muestre el nuevo **Fragment**.

- Abre el archivo **BienvenidaFragment** y crea un método llamado **inicializarEventos**, donde vamos a escribir el código fuente que nos permitirá navegar desde **BienvenidaFragment** hasta **MensajeFragment**. Este método **inicializarEventos** se llamará en **onCreateView**

```

1. class BienvenidaFragment : Fragment() {
2.     // resto omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarBinding(inflater,container)
8.         inicializarEventos()
9.         return binding.root
10.    }
11.    private fun inicializarEventos(){
12.        binding.btnComenzar.setOnClickListener {
13.            // acciones cuando se pulsa btnComenzar
14.        }
15.    }
16. }

```

- Dentro del código que recibe **setOnClickListener**, obtenemos el **navigation controller** llamando al método **findNavController()**

```

1. binding.btnComenzar.setOnClickListener {
2.     val navigationController = findNavController()
3. }

```

El método **findNavController** puede ser usado dentro de un **Fragment** y nos devuelve un **NavController** con que podemos lanzar un **Action** que nos lleve a otro **Fragment**

Recordamos que los **Actions** están definidos gráficamente como flechas que unen los **Fragment** en el **navigation graph**.

- A continuación, obtenemos un objeto **Action** con la acción que nos permite pasar desde **BienvenidaFragment** a **MensajeFragment**

```
1. private fun inicializarEventos(){
2.     binding.btnComenzar.setOnClickListener {
3.         val navigationController = findNavController()
4.         val accion = BienvenidaFragmentDirections.actionBienvenidaFragmentToMensajeFragment()
5.     }
6. }
```

La clase **BienvenidaFragmentDirections** contiene las acciones que “se inician” en **BienvenidaFragment** y nos permiten navegar a otros **Fragment**

- Por último, llamamos al método **navigate** del **NavController**, pasando como parámetro la acción obtenida en el punto anterior

```
1. private fun inicializarEventos(){
2.     binding.btnComenzar.setOnClickListener {
3.         val navigationController = findNavController()
4.         val accion = BienvenidaFragmentDirections.actionBienvenidaFragmentToMensajeFragment()
5.         navigationController.navigate(accion)
6.     }
7. }
```

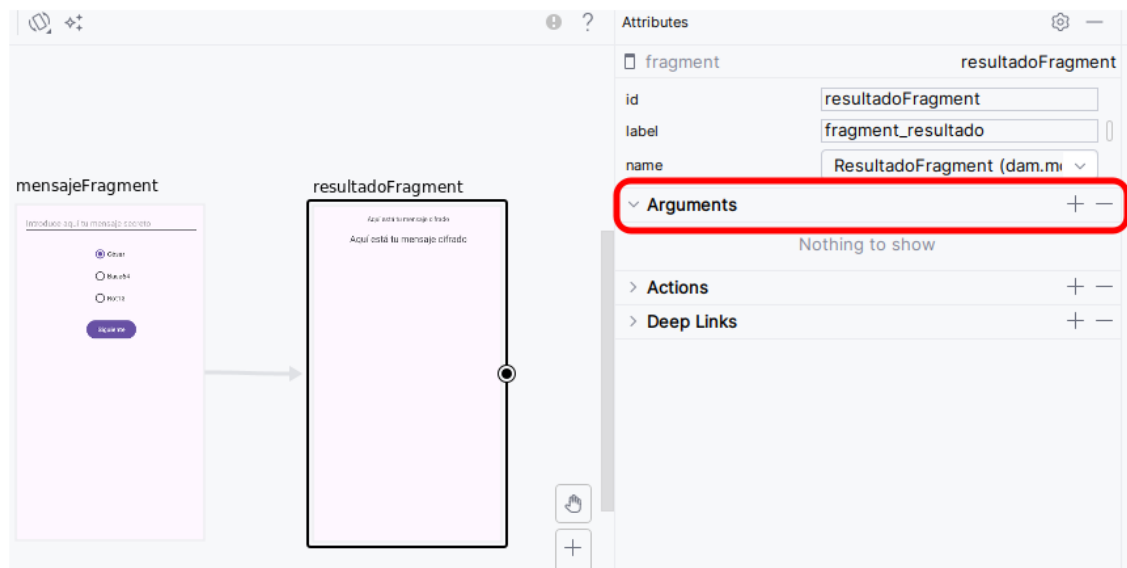
- Pon en marcha la app y comprueba que al pulsar **btnInicio** navegamos a **MensajeFragment**

7 – Paso de información entre Fragments

Al pulsar **btnCifrar** navegamos desde **MensajeFragment** hasta **ResultadoFragment**, pero es necesario pasar el mensaje y los métodos de cifrado que elegimos en **MensajeFragment** a **ResultadoFragment**.

Realizaremos estos pasos de forma gráfica en el **navigation graph**

- Abre el archivo **nav_graph.xml** con el diseñador gráfico
- Selecciona en el gráfico **ResultadoFragment**, que es el **Fragment** que necesita la información que queremos enviarle desde **MensajeFragment**
- Observa que en la derecha, hay un lugar que pone **Arguments**



- Pulsa en el signo + de la zona **Arguments** y aparecerá una ventana que rellenaremos con estos datos:
 - o Nombre del argumento: **mensaje**
 - o Tipo de dato: **String**

Esta ventana nos permite definir todos los datos que necesita **ResultadoFragment** para funcionar

En nuestro caso, son dos datos los que necesita: un String con el mensaje que escribe el usuario, y una lista con los números de los botones de radio seleccionados (0=Cesar, 1 = Base64, 2 = Rot13)

- Añade a **resultadoFragment** un nuevo argumento llamado **cifradosSeleccionados**, su tipo será entero y marcaremos **array**, para indicar que se deberá pasar una lista



Cuando un parámetro **Integer** se marca como **array**, su tipo de dato en Kotlin deberá ser **IntArray**, que es el tipo equivalente al **int[]** de Java

- A continuación, abre el código fuente de **MensajeFragment** y añade un método **inicializarEventos** donde programaremos las acciones que se realizarán al pulsar **btnCifrar**

```

1. class MensajeFragment : Fragment() {
2.     // resto omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarViewModel()
8.         inicializarBinding(inflater, container)
9.         inicializarEventos()
10.        return binding.root
11.    }
12.    private fun inicializarEventos(){
13.        binding.btnCifrar.setOnClickListener {
14.        }
15.    }
16. }

```

- Añadimos estos dos métodos auxiliares que nos van a hacer falta:
 - **getNumero** → Recibe un **Checkbox** y nos devuelve el número **0** si es **chkCifrado1**, **1** si es **chkCifrado2** y **2** si es **chkCifrado3**. Se programa consultando la posición del **Checkbox** en el **LinearLayout** que lo contiene.
 - **getCasillasSeleccionadas** → Nos devuelve una lista de tipo **IntArray** con los números asociados a las casillas de verificación seleccionadas. Se usará programación funcional para programar este método (*también se puede hacer con bucles, aunque es más largo*)

```

1. private fun getNumero(c:CheckBox):Int = binding.linearLayout.indexOfChild(c)
2. private fun getCasillasSeleccionadas():IntArray =
3.     listOf(binding.chkCifrado1,binding.chkCifrado2,binding.chkCifrado3)
4.         .filter { casilla -> casilla.isChecked }
5.         .map { casilla -> getNumero(casilla)}
6.         .toIntArray()

```

- A continuación, dentro del código que se activa al pulsar **btnCifrar**, recuperamos el mensaje escrito en **txtMensaje** y con los dos métodos auxiliares, obtenemos la lista con los números de las casillas de verificación seleccionadas.

```

1. private fun inicializarEventos(){
2.     binding.btnCifrar.setOnClickListener {
3.         val mensaje = binding.txtMensaje.text.toString()
4.         val cifradosSeleccionados = getCasillasSeleccionadas()
5.     }
6. }

```

- Por último, obtenemos el **Navigation Controller**, la **Action** que nos lleva desde **MensajeFragment** a **ResultadoFragment**, y llamaremos al método **navigate** pasándole como parámetros el texto y la lista obtenidos de la interfaz

```

1. private fun inicializarEventos(){
2.     binding.btnCifrar.setOnClickListener {
3.         val mensaje = binding.txtMensaje.text.toString()
4.         val cifradosSeleccionados = getCasillasSeleccionadas()
5.         val navigationController = findNavController()
6.         val accion = MensajeFragmentDirections
7.             .actionMensajeFragmentToResultadoFragment(mensaje,cifradosSeleccionados)
8.         navigationController.navigate(accion)
9.     }
10. }

```

- Para comprobar que todo funciona correctamente, vamos a hacer que el método **onCreateView** de **ResultadoFragment** se llame a un método auxiliar **testParametros**, que imprima en **LogCat** el valor de los argumentos recibidos.

```

1. class ResultadoFragment : Fragment() {
2.     // resto omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarViewModel()
8.         inicializarBinding(inflater,container)
9.         testParametros()
10.        return binding.root
11.    }
12.    fun testParametros(){
13.        val mensajeRecibido = ResultadoFragmentArgs.fromBundle(requireArguments()).mensaje
14.        val lista = ResultadoFragmentArgs.fromBundle(requireArguments()).cifradosSeleccionados
15.        Log.d("test parámetros recibidos",mensajeRecibido)
16.        Log.d("test parámetros recibidos",lista.joinToString(", "))
17.    }
18. }

```

Los argumentos que recibe **ResultadoFragment** se encuentran en la clase **ResultadoFragmentArgs**

Podemos acceder a ellos llamando al método **fromBundle(requireArguments())**

- Ejecuta la app y comprueba que al llegar a **ResultadoFragment** en **LogCat** aparecen mensajes indicando el valor correcto de los argumentos recibidos.

8 – Arquitecturas MVC y MVVM

El siguiente paso de nuestro proyecto consiste en hacer que el mensaje recibido por **ResultadoFragment** sea cifrado usando los algoritmos indicados por el usuario: Cesar, Rot13 o Base64 y mostrar en el **RecyclerView** los resultados.

Para ello, programaremos clases e interfaces que hagan dicha funcionalidad. Todas ellas forman el **modelo** de la app.

La estructura **MVC** de una app consta de tres partes:

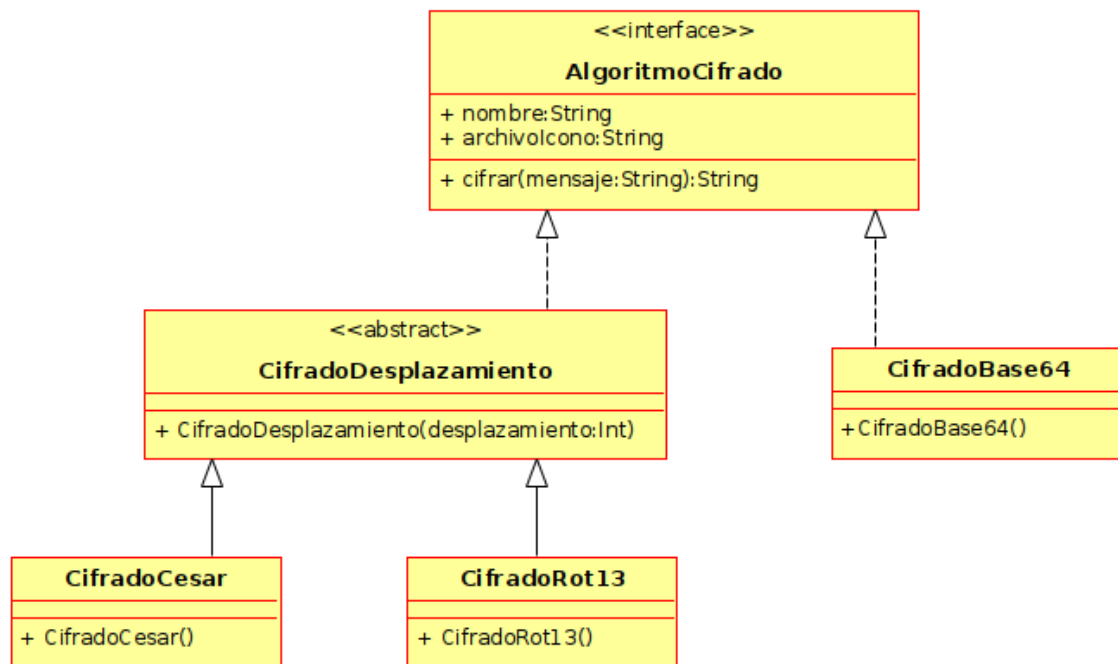
- **Modelo:** Son las clases que programamos nosotros para concentrar en ella la funcionalidad de la app, de forma independiente de la interfaz de usuario
- **Vista:** Son los archivos **xml** de la carpeta **layout**, en la que están los diseños de las pantallas que forman la app

- **Controlador:** Son las clases, como los **Activity** y los **Fragments**, que recuperan los elementos de la interfaz (las vistas) y definen los eventos que se producen cuando el usuario interactúa con la app

Y si se usa un **view model**, la estructura se denomina **MVVM** y será esta:

- **Modelo:** Tiene la misma misión que en la arquitectura MVC
- **Vista:** Se encarga de todo lo relacionado con el diseño de la interfaz de usuario, y está formado por los archivos **xml** de las vistas y también los **Activity** y los **Fragment**. Esto es así ya que al usar un **view model**, la funcionalidad de los controladores se reduce únicamente a inicializar cosas de la interfaz.
- **View model:** Concentra las variables que definen el estado de la interfaz de usuario y todas las acciones que se realizan sobre ellas.

- En nuestra app el modelo será este diagrama de clases



- o **AlgoritmoCifrado:** Representa un algoritmo de cifrado cualquiera
- o **CifradoBase64:** Cifra un mensaje usando el método **Base64.encodeToString**
- o **CifradoDesplazamiento:** Cifra un mensaje sustituyendo cada letra del mensaje por la letra que se obtiene sumándole en el alfabeto, tantas letras como indica el desplazamiento
- o **CifradoCesar:** Es un cifrado cuyo desplazamiento es 1 (sustituye cada letra por su siguiente en el alfabeto)
- o **CifradoRot13:** Es un cifrado cuyo desplazamiento es 13 (sustituye cada letra por la que viene 13 posiciones después en el alfabeto)

- Copia las imágenes adjuntas **cesar.webp**, **base64.webp** y **rot13.webp** a la carpeta **drawable** del proyecto
- Dentro del paquete **dam.moviles.cifrados** haz un paquete llamado **modelo** y añade a él la interfaz **AlgoritmoCifrado**

```
1. interface AlgoritmoCifrado {
2.     val nombre:String
3.     val idIcono:Int
4.     fun cifrar(mensaje:String):String
5. }
```

☞ **¿qué hacen unas propiedades en la interfaz?** En Java las interfaces solo pueden tener métodos y nunca pueden tener propiedades, pero en Kotlin sí. Las clases que implementen la interfaz, están obligadas a incorporar esas propiedades

- En ese mismo paquete, añadimos la clase **CifradoBase64**, que hará esto:
 - Tendrá las propiedades **nombre** y **archivolcono**
 - Su método **cifrar** llamará al método **Base64.encodeToString**

```
1. class CifradoBase64 : AlgoritmoCifrado {
2.     override val nombre = "Base 64"
3.     override val idIcono: Int = R.drawable.base64
4.     override fun cifrar(mensaje: String): String =
5.         Base64.encodeToString(mensaje.toByteArray(),Base64.DEFAULT)
6. }
```

En Kotlin se usa la palabra **override** para indicar que una propiedad o método procede de una interfaz o clase abstracta

- En ese mismo paquete, añadimos la clase **CifradoDesplazamiento**. Usaremos **programación funcional** (el estilo de programación recomendado por Kotlin) para programar los métodos, pero en su lugar, también podemos usar bucles.

```
1. open abstract class CifradoDesplazamiento(val desplazamiento:Int) : AlgoritmoCifrado{
2.     override fun cifrar(mensaje: String): String =
3.         mensaje.toCharArray()
4.             .map { letra -> letra.code }
5.             .map { numero -> numero + desplazamiento}
6.             .map { numero -> numero.toChar()}
7.             .joinToString("")
8. }
```

Para programar esta clase se han tenido en cuenta varias cosas del lenguaje Kotlin:

- Poner **val desplazamiento:Int** en la declaración de la clase hace que la clase automáticamente tenga una propiedad llamada **desplazamiento**, y además, el constructor de la clase recibe un valor para inicializarlo.
- Por motivos de seguridad las clases no pueden tener hijos. Si queremos que una clase pueda tener clases hijas, deberemos ponerles la palabra **open**
- Como **CifradoDesplazamiento** es una clase abstracta, no está obligada a definir las propiedades y métodos abstractos de su interfaz o clase padre.

- Ahora vamos a añadir al paquete las clases **CifradoCesar** y **CifradoRot13**, que simplemente heredan de **CifradoDesplazamiento** con valores de desplazamiento 1 y 13 respectivamente. Ambas definirán las propiedades **nombre** e **idIcono** con sus valores correspondientes.

```

1. // archivo CifradoCesar.kt
2. class CifradoCesar : CifradorDesplazamiento(1) {
3.     override val nombre: String = "César"
4.     override val idIcono: Int = R.drawable.cesar
5. }
6. // archivo Rot13.kt
7. class CifradoRot13 : CifradorDesplazamiento(13) {
8.     override val nombre: String = "Rot 13"
9.     override val idIcono: Int = R.drawable.rot13
10. }

```

Cuando en Kotlin una clase hereda de otra, se ponen entre paréntesis los valores que se pasan al constructor de la clase padre, que en Java se hacía poniendo **super** al principio del constructor de la clase hija.

- Por último, vamos a añadir en el archivo **AlgoritmoCifrado.kt** una **función** que reciba un número entero (0,1,2) y nos devuelva el método de cifrado Cesar, Rot13 o Base64 respectivamente.

```

1. fun getAlgoritmoCifrado(numero:Int):AlgoritmoCifrado =
2.     when(numero){
3.         0 -> CifradoCesar()
4.         1 -> CifradorRot13()
5.         2 -> CifradoBase64()
6.         else -> throw Exception("opción no permitida")
7.     }

```

- Una vez que el modelo está terminado, modificamos el método **testParametros** de **ResultadoFragment** para que en lugar de mostrar en **Logcat** la lista de cifrados, nos muestre el nombre de cada cifrado seleccionado, y el texto cifrado.

```

1. fun testParametros(){
2.     val mensajeRecibido = ResultadoFragmentArgs
3.         .fromBundle(requireArguments()).mensaje
4.     val listaRecibida = ResultadoFragmentArgs
5.         .fromBundle(requireArguments()).cifradosSeleccionados
6.     Log.d("test parámetros recibidos", "el mensaje original es $mensajeRecibido")
7.     listaRecibida
8.         .map { numero -> getAlgoritmoCifrado(numero) }
9.         .forEach { cifrador ->
10.             Log.d("test parámetros recibidos",
11.                 "El cifrado ${cifrador.nombre} produce: ${cifrador.cifrar(mensajeRecibido)}"
12.             )
13.         }
14. }

```

- Ejecuta la app y comprueba que el mensaje enviado desde **MensajeFragment** hacia **ResultadoFragment** aparece en **Logcat** correctamente cifrado según los algoritmos de cifrado elegidos

- Ahora que ya hemos visto que el paso de parámetros funciona correctamente, puedes borrar el método **testParametros** y su llamada en **onCreateView**
- Abre el archivo **ResultadoFragmentViewModel** y coloca en él variables de instancia para la lista de cifrados y la lista de resultados producidos por los cifrados. Como dichas variables de instancia se rellenarán fuera del constructor, llevarán la palabra **lateinit**

```
1. class ResultadoFragmentViewModel : ViewModel() {
2.     lateinit var listaCifrados:List<AlgoritmoCifrado>
3.     lateinit var listaResultados:List<String>
4. }
```

- Para dar el valor inicial a estas variables, programaremos en **ResultadoFragmentViewModel** dos métodos auxiliares: (*aunque se pueden hacer con bucles, usaremos programación funcional*)
 - **inicializarCifrados**: Recibe una lista de números (que es la que procede de **MensajeFragment**) y nos rellena **listaCifrados** con los objetos **AlgoritmoCifrado** correspondientes a dichos números. Para programarlo, nos ayudaremos de la función **getAlgoritmoCifrado** que ya tenemos hecha.
 - **cifrar**: Recibe un mensaje y rellena **listaResultados**, cifrando el mensaje con cada uno de los cifrados que hay en la lista de cifrados.

```
1. class ResultadoFragmentViewModel : ViewModel() {
2.     lateinit var listaCifrados:List<AlgoritmoCifrado>
3.     lateinit var listaResultados:List<String>
4.     fun inicializarCifrados(listaNumeros:IntArray){
5.         listaCifrados = listaNumeros.map { numero -> getAlgoritmoCifrado(numero) }
6.     }
7.     fun cifrar(mensaje:String){
8.         listaResultados = listaCifrados.map { cifrado -> cifrado.cifrar(mensaje)}
9.     }
10. }
```

Observa que cada resultado se encuentra, dentro de **listaResultados**, en la misma posición que el cifrado correspondiente en **listaCifrados**

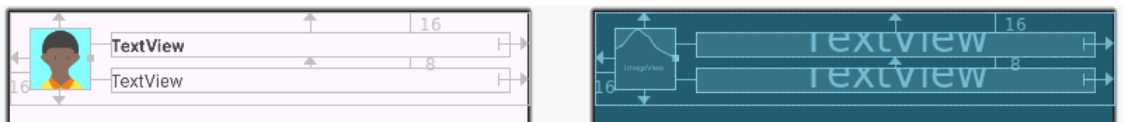
- A continuación, vamos a **ResultadoFragment** y vamos a completar el método **inicializarViewModel**, de manera que se inicialicen las variables de instancia del **view model** que tiene **ResultadoFragment**. Esto se hará recogiendo los valores de los parámetros, y llamando a los métodos auxiliares que acabamos de programar.

```
1. fun inicializarViewModel(){
2.     viewModel = ViewModelProvider(this).get(ResultadoFragmentViewModel::class.java)
3.     val mensajeOriginal = ResultadoFragmentArgs.fromBundle(requireArguments()).mensaje
4.     val listaNumerosCifrados = ResultadoFragmentArgs
5.         .fromBundle(requireArguments()).cifradosSeleccionados
6.     viewModel.inicializarCifrados(listaNumerosCifrados)
7.     viewModel.cifrar(mensajeOriginal)
8. }
```

- En la carpeta **res/layout** haz clic con el botón derecho del ratón y luego elige **new → layout resource file**
- En la pantalla que aparece, pon el nombre **item_cifrado** y pulsa **OK**
- Abre el archivo **item_cifrado.xml** con el diseñador, borra todo lo que haya y cambia su elemento principal para que sea un **ConstraintLayout**
- Selecciona el **ConstraintLayout** y ponle de altura **72dp**

Material Design recomienda que la altura de las vistas de una lista sea de **72dp**

- Usando el diseñador, arrastra los siguientes elementos y colócalos como se indica:
 - **ImageView (imgCifrado)**: Es una imagen de tamaño 48x48 donde se mostrará el icono asociado al método de cifrado usado.
 - **TextView (txtNombreCifrado)**: Es una etiqueta donde se mostrará el nombre del cifrado. Su texto se muestra en negrita (usar el atributo **android:textStyle**)
 - **TextView (txtMensajeCifrado)**: Aquí se mostrará el mensaje cifrado.



- En el paquete de código fuente **vista**, crea la clase **ResultadoViewHolder**, cuyo constructor recibirá como parámetro un objeto **ItemCifradoBinding** (la “mochila” que contiene los objetos de la interfaz definida en **item_resultado.xml**) y pasa al constructor de su clase padre el objeto raíz de dicha interfaz

```
1. class ResultadoViewHolder(val binding:ItemCifradoBinding)
2.     : RecyclerView.ViewHolder(binding.root) {
3. }
```

La clase **ResultadoViewHolder** es hija de **RecyclerView.ViewHolder** y su misión es portar (aquí lo hacemos mediante la variable de instancia **binding**) los elementos de la interfaz asociada a un ítem de la lista que se muestra en el **RecyclerView** y programar los eventos asociados a dichos componentes.

- Añade a **ResultadoViewHolder** variables de instancia para el método de cifrado y el mensaje cifrado:

```
1. class ResultadoViewHolder(val binding:ItemCifradoBinding):RecyclerView.ViewHolder(binding.root){
2.     var cifrado:Cifrado=""
3.     var mensaje:String=""
4. }
```

- Añade a **ResultadoViewHolder** un método **mostrarResultado** que reciba un cifrado y el mensaje que produce, y rellene los datos de la vista con ellos:

```

1. class ResultadoViewHolder(val binding:CifradoBinding): RecyclerView.ViewHolder(binding.root) {
2.     var cifrado:String=""
3.     var mensaje:String=""
4.     fun mostrarResultado(c:AlgoritmoCifrado, m:String){
5.         cifrado=c.nombre
6.         mensaje=m
7.         binding.txtNombreCifrado.text=c.nombre
8.         binding.imgCifrado.setImageResource(c.idIcono)
9.         binding.txtMensajeCifrado.text=mensaje
10.    }
11. }

```

El método **setImageResource** pone a un **ImageView** la imagen cuyo id (en la carpeta **res/drawable**) se pasa como parámetro.

- A continuación, añade al paquete **vista** una clase llamada **CifradoAdapter**, que heredará de **RecyclerView.Adapter<ResultadoViewHolder>**. El constructor de **CifradoAdapter**, recibirá la listas de cifrados, la de resultados y el código fuente que se activará al pulsar un **ResultadoViewHolder**.

```

1. class CifradoAdapter(
2.     val listaCifrados:List<AlgoritmoCifrado>,
3.     val listaResultados:List<String>,
4.     val lambda:(ResultadoViewHolder) -> Unit
5. ) : RecyclerView.Adapter<ResultadoViewHolder>() {
6.     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ResultadoViewHolder {
7.         }
8.     override fun getItemCount(): Int {
9.         }
10.    override fun onBindViewHolder(holder: ResultadoViewHolder, position: Int) {
11.        }
12. }

```

La clase **RecyclerView.Adapter** es abstracta, y eso hace que sus clases hijas (como **CifradoAdapter**) estén obligadas a sobrescribir sus métodos abstractos, que son:

- **onCreateViewHolder**: Lo utiliza el **RecyclerView** para crear un objeto **ResultadoViewHolder** cada vez que necesita uno.
- **onBindViewHolder**: Lo utiliza el **RecyclerView** cada vez que debe poner un ítem en un **ResultadoViewHolder**. Como parámetros recibe la posición del ítem que va a ser colocado y el **ResultadoViewHolder** donde se deberá mostrar el ítem.
- **getItemCount**: Devuelve el número total de ítems que hay en la lista de objetos que se rotan entre las vistas del **RecyclerView**.

- Programa el método **onCreateViewHolder**, cuya misión es crear y devolver un objeto **ResultadoViewHolder**. Para ello, obtiene un objeto **inflater** (que sabe convertir documentos xml en objetos), y lo pasa al método **ItemCifradoBinding.inflate**, de forma que se cree un objeto **binding** con todos los objetos de la interfaz. Por último, crea un **ResultadoViewHolder** con ese objeto **binding**

```

1. override fun onCreateView(parent: ViewGroup, viewType: Int): ResultadoViewHolder {
2.     val inflater = LayoutInflater.from(parent.context)
3.     val binding = ItemCifradoBinding.inflate(inflater, parent, false)
4.     return ResultadoViewHolder(binding)
5. }

```

La misión del método **onCreateViewHolder** es crear y devolver un objeto **ResultadoViewHolder**.

- Programa el método **onBindViewHolder**, consultando primero el cifrado y resultado que en sus correspondientes listas ocupan la posición pasada como parámetro, y rellenando con sus datos los elementos de la interfaz de **ResultadoViewHolder**. También hacemos que al pulsar el elemento principal del **ResultadoViewHolder** se ejecute la expresión lambda

```

1. override fun onBindViewHolder(holder: ResultadoViewHolder, position: Int) {
2.     val cifrado = listaCifrados[position]
3.     val mensajeCifrado = listaResultados.get(position)
4.     holder.mostrarResultado(cifrado, mensajeCifrado)
5.     holder.binding.root.setOnClickListener {
6.         lambda(holder)
7.     }
8. }

```

La misión del método **onBindViewHolder** es modificar la interfaz de la vista que porta **ResultadoViewHolder**, para que muestre los datos del ítem cuya posición en la lista de datos se pasa como parámetro.

- Por último, programa el método **getItemCount**, que simplemente devuelve el tamaño de la lista de cifrados (o la de resultados)

```

1. override fun getItemCount(): Int = listaCifrados.size

```

La misión del método **getItemCount** es devolver el número total de ítems que hay en la lista de datos que se mostrará en el **RecyclerView**

- Por último, abre el archivo **ResultadoFragment** y añade un método auxiliar **configurarRecyclerView**, que llamarás en **onCreateView**.

```

1. class ResultadoFragment : Fragment() {
2.     // resto omitido
3.     override fun onCreateView(
4.         inflater: LayoutInflater, container: ViewGroup?,
5.         savedInstanceState: Bundle?
6.     ): View? {
7.         inicializarViewModel()
8.         inicializarBinding(inflater, container)
9.         inicializarRecyclerView()
10.        return binding.root
11.    }
12.    fun inicializarRecyclerView(){
13.        // inicializamos aquí el RecyclerView
14.    }
15. }

```

- En **inicializarRecyclerView**, crea un objeto **CifradoAdapter** y pónselo a la propiedad **adapter** del **RecyclerView**. *El constructor de **CifradoAdapter** nos pide las listas de cifrados y resultados, que tenemos en el **view model***

```

1. fun inicializarRecyclerView(){
2.     binding.recResultados.adapter = CifradoAdapter(
3.         viewModel.listaCifrados,
4.         viewModel.listaResultados
5.     ){
6.         // aquí compartiremos a otra app el mensaje cifrado
7.     }
8. }

```

- Si ejecutas la app, verás que todo funciona bien, aunque los iconos aparecen cuadrados.
- Abre el archivo **themes.xml**, que está en la carpeta **res/values/themes** y observa que el nombre del tema de nuestra app es **Base.Theme.Cifrador**, y como “tema padre” usa el tema **Theme.Material3.DayNight.NoActionBar**
- Añade dentro de la etiqueta **style** la siguiente etiqueta **item**, para cambiar el color principal de la app a rojo:

```

1. <resources xmlns:tools="http://schemas.android.com/tools">
2.     <style name="Base.Theme.Cifrador" parent="Theme.Material3.DayNight.NoActionBar">
3.         <item name="colorPrimary">#FF0000</item>
4.     </style>
5. </resources>

```

Quando añadimos una clave a un tema, estamos cambiando su valor respecto al valor que tiene esa clave en el tema padre

- Ejecuta la app y comprueba que solo por cambiar el color principal de la app, cambia el color de un montón de cosas que dependen de ese valor, como el color de los botones o el de las casillas de verificación.
- Abre el archivo **item_cifrado.xml** con la vista de código fuente
- Encierra todo el contenido del archivo en una etiqueta **MaterialCardView**, para que todo se quede así:

```

1. <com.google.android.material.card.MaterialCardView
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     xmlns:app="http://schemas.android.com/apk/res-auto"
4.     xmlns:tools="http://schemas.android.com/tools"
5.     android:layout_marginBottom="8dp"
6.     android:layout_width="match_parent"
7.     android:layout_height="72dp">
8.     <!-- Aquí viene el ConstraintLayout -->
9. </com.google.android.material.card.MaterialCardView>

```

- Ejecuta la app y verás que los cifrados aparecen cada uno dentro de una carta, y que al pinchar en ellos, se ilumina la zona correspondiente.
- Navega hacia **ResultadoFragment** y comprueba que los **MaterialCardView** no tienen fondo.

Algunos elementos, como los **MaterialCardView** poseen su propio tema (en los **MaterialCardView** es **Widget.MaterialComponents.CardView**), que es necesario configurar si queremos cambiar su apariencia

- Abre el archivo **themes.xml** y añade una etiqueta **style** que sobrescribirá el atributo **cardBackgroundColor** del estilo de las **MaterialCardView**

```
1. <style name="resultadoCardView" parent="Widget.MaterialComponents.CardView">
2.     <item name="cardBackgroundColor">#FE7E6</item>
3. </style>
```

Para elegir el color más adecuado, pulsa el ratón en el cuadrito de color que aparece junto al número de línea y aparecerá un selector de color donde podremos elegir el color que más nos guste.

- Por último, vamos a indicar que todas las **MaterialCardView** tengan el estilo que hemos llamado **resultadoCardView**. Eso lo haremos añadiendo la clave **materialCardViewStyle** al estilo del tema de la app

```
1. <style name="Base.Theme.Cifrador" parent="Theme.Material3.DayNight.NoActionBar">
2.     <item name="colorPrimary">#FF0000</item>
3.     <item name="materialCardViewStyle">@style/resultadoCardView</item>
4. </style>
```

- Ejecuta la app y comprueba que las **MaterialCardView** tienen ahora un color de fondo.
- Abre el archivo **themes.xml**
- Añade un estilo llamado **bordeResultado**, que incluye la clave **cornerSize** para definir el tamaño de doblado de los bordes (esto es similar al border-radius de css), y **cornerFamily** para indicar el tipo de operación que se realiza en el borde

```
1. <style name="bordeResultado">
2.     <item name="cornerSizeTopRight">25dp</item>
3.     <item name="cornerFamilyTopRight">cut</item>
4.     <item name="cornerSizeBottomLeft">25dp</item>
5.     <item name="cornerFamilyBottomLeft">cut</item>
6. </style>
```

- Vete al código fuente de **item_cifrado.xml** y cambia la etiqueta **ImageView** por un **ShapeableImageView** de esta forma:

```
1. <com.google.android.material.imageview.ShapeableImageView
2.     android:id="@+id/imgAlgoritmo"
3.     android:layout_width="48dp"
4.     android:layout_height="48dp"
5.     android:layout_marginStart="16dp"
6.     app:layout_constraintBottom_toBottomOf="parent"
7.     app:layout_constraintStart_toStartOf="parent"
8.     app:layout_constraintTop_toTopOf="parent"
9.     app:shapeAppearanceOverlay="@style/circular"
10.     tools:srcCompat="@tools:sample/avatars" />
```

9 – Gestión del botón de atrás

Si ejecutamos el proyecto veremos que al pulsar el botón **atrás**, si estamos en **ResultadoFragment**, volveremos a **MensajeFragment**. Podría ocurrir que en su lugar, quisiéramos volver a **BienvenidaFragment**, para así poder comenzar desde cero.

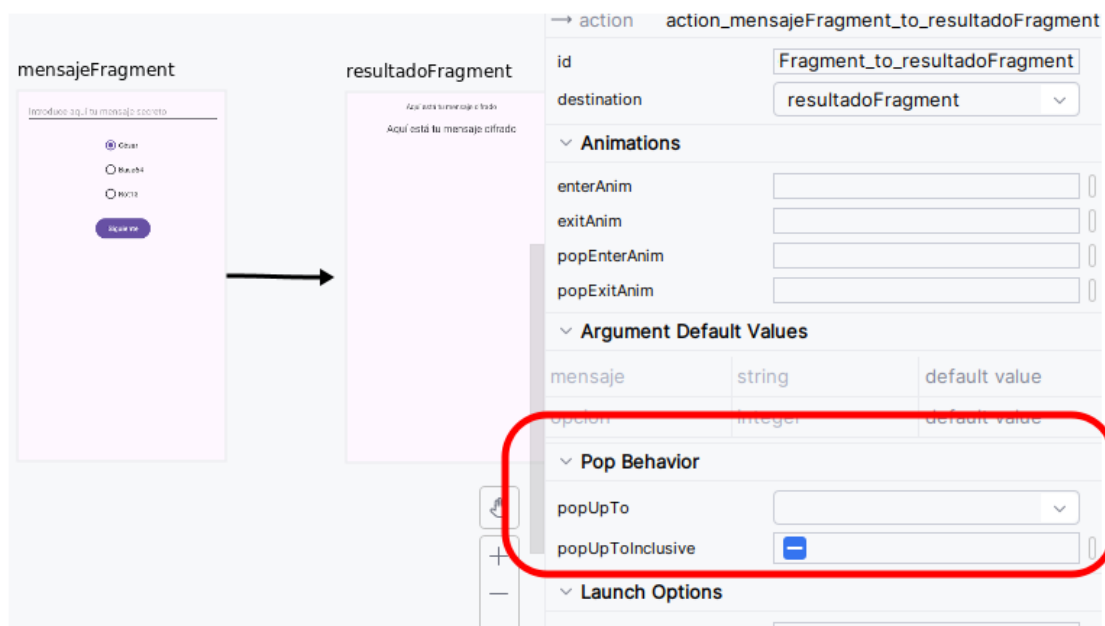
Para poder hacer esto, debemos acceder al **back stack**.

Cuando se navega entre **Fragment**, Android va almacenando en memoria la secuencia de **Fragments** visitados. Dicha memoria se denomina **back stack**.

Si queremos que al pulsar atrás en **ResultadoFragment**, volvamos a **BienvenidaFragment**, deberemos eliminar del **back stack** a **MensajeFragment**.

Realizaremos gráficamente esta acción en el **navigation graph**

- Abre con el diseñador el archivo **nav_graph.xml**
- Pulsa el ratón en la **Action** que lleva desde **mensajeFragment** hasta **resultadoFragment**
- Observa que hay un lugar que pone **pop behavior**



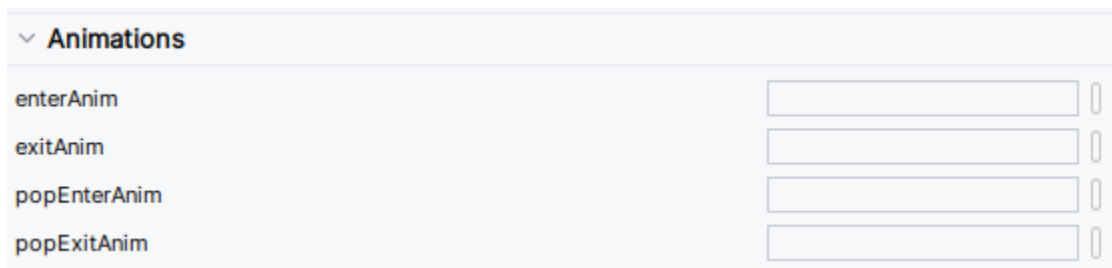
- En dicha zona, pon estas opciones:
 - o **popUpTo** → **bienvenidaFragment**
 - o **popUpToInclusive** → **false**

Con esto, estamos diciendo que cuando suceda la acción que navega desde **MensajeFragment** hasta **ResultadoFragment**, se eliminará del **back stack** todo lo que haya visitado previamente hasta llegar a **BienvenidaFragment**, y que dicho **Fragment** no debe ser eliminado (valor **false** en **popUpInclusive**)

10 – Animaciones a la navegación

En nuestro proyecto podemos navegar de un **Fragment** a otro, pero podemos también configurar la animación que queramos que se produzca cuando se activa una **Action**. Realizaremos esto de forma gráfica en el **navigation graph**

- Abre el archivo **nav_graph.xml** con el diseñador de Android Studio
- Pulsa la **Action** que nos lleva desde **bienvenidaFragment** hacia **mensajeFragment**
- Observa que en la derecha hay una zona que pone **Animations**



La zona **Animations** permite indicar el tipo de animación que se pone a los **Fragment** que entran (**enter**) y salen (**salen**) de la pantalla.

- **enterAnim**: Animación que se pone al **Fragment** que entra en pantalla
 - **exitAnim**: Animación que se pone al **Fragment** que sale de la pantalla
 - **popEnterAnim**: Animación que se pone al **Fragment** que entra en pantalla, cuando se pulsa el botón de atrás
 - **exitEnterAnim**: Animación que se pone al **Fragment** que sale de la pantalla, cuando se pulsa el botón de atrás
- Con la **Action** que lleva desde **bienvenidaFragment** hacia **mensajeFragment** seleccionada, pulsa el pequeño botón que hay junto a **enterAnim** y aparecerá una ventana con animaciones predefinidas, donde seleccionarás **slide_in_left**
 - Repite lo anterior y en **exitAnim** elige **fade_out**
 - Ejecuta la app y observa que al pasar desde **bienvenidaFragment** hacia **mensajeFragment** el **BienvenidaFragment** se desvanece y el **MensajeFragment** aparece de izquierda a derecha.
 - Repite los mismos pasos para pasar desde **mensajeFragment** hacia **resultadoFragment**
 - Observa que si estás en **ResultadoFragment** y pulsas el botón de atrás, vuelves bruscamente a **BienvenidaFragment**

Android viene con unas animaciones predefinidas, como **slide_in_left** o **fade_out**, pero podemos crear nuestras propias animaciones en el archivo **anim.xml**

- Haz clic con el botón derecho del ratón en la carpeta **res** y en la ventana que aparece escribe:
 - Nombre del archivo: **hacia_derecha.xml**
 - Tipo de recurso: **Animation**
- Abre el archivo **entrada_por_derecha.xml** y copia el siguiente contenido:

```

1. <set
2.     xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:shareInterpolator="false">
4.     <translate
5.         android:duration = "500"
6.         android:fromXDelta="100%"
7.         android:fromYDelta="0%"
8.         android:toXDelta="0%"
9.         android:toYDelta="0%" />
10. </set>

```

*☞ **¿Qué estamos indicando con estas etiquetas?** En este archivo estamos programando una **traslación**, que consiste en mover algo de unas coordenadas a otras. La traslación queda definida por su duración (en ms) y 4 parámetros que la posición donde comienza la animación y la posición en la que termina. Estas posiciones se expresan respecto a la posición final del elemento (valor 0)*

- Cambia las animaciones que has puesto para que en lugar de **slide_in_left** tengas **hacia_derecha**
- Ejecuta la app y comprueba que al ir avanzando por la app los **Fragment** entran de derecha a izquierda.
- Abre el archivo **nav_graph.xml**, selecciona la **Action** que navega desde **mensajeFragment** hacia **resultadoFragment** y coloca estos valores:
 - **popEnterAnim: slide_in_left**
 - **popExitAnim: fade_out**
- Ejecuta la app y comprueba que al volver atrás en **ResultadoFragment**, aparece **BienvenidaFragment** entrando de izquierda a derecha

11 – Implicit Intent

Vamos a hacer que al pulsar uno de los cifrados, podamos enviar el mensaje con alguna app que tengamos instalada que permita el envío de mensajes (sms, correo, etc).

Para abrir una app y pasarle información, es necesario utilizar un **implicit intent**, que es un objeto que describe al sistema Android una acción que se quiere realizar sobre el dispositivo, como por ejemplo, enviar un dato a otra app.

El sistema Android busca la app más adecuada para recibir el dato, y la pone en marcha. Si hay varias, permite al usuario elegir cuál quiere utilizar.

- Vete a **ResultadoFragment** al método **inicializarRecyclerView** y escribe el siguiente código fuente, para crear un **Intent** que envíe el mensaje cifrado a una app externa

```

1. private fun inicializarRecyclerView(){
2.     binding.lstResultados.adapter = CifradoAdapter(
3.         viewModel.listaCifrados,
4.         viewModel.listaResultados
5.     ){ holder ->
6.         val intent=Intent() // creamos el Intent
7.         intent.action= Intent.ACTION_SEND // lo rellenamos con la información a enviar
8.         intent.type="text/plain"
9.         intent.putExtra(Intent.EXTRA_TEXT,holder.mensaje)
10.        binding.root.context.startActivity(intent) // iniciamos el Intent
11.    }

```

Una vez creado el **Intent**, es necesario configurarle estas características:

- o **action** → Es la acción que queremos realizar. En este caso, **ACTION_SEND** significa que queremos enviar algo a otra app
- o **type** → indica el tipo (mime) que tiene el dato que se desea enviar. En nuestro caso, **text/plain** indica que se enviará texto plano
- o **putExtra** → Con este método adjuntamos al **Intent** el mensaje de texto que se va a enviar

El método **startActivity** se encuentra en el objeto **context**, que como ya vimos, representa la app y se puede obtener en el elemento raíz de la interfaz.

- Pon en marcha la app y comprueba que al pulsar en uno de los resultados, se abre una ventana para elegir la app con la que enviar el mensaje
- Existe una ventana alternativa llamada **Android Sharesheet**, que en ocasiones permite elegir más cómodamente el destinatario de la información. Podemos abrir dicha ventana mediante el método **Intent.createChooser**, así:

```

1. private fun inicializarRecyclerView(){
2.     binding.lstResultados.adapter = CifradoAdapter(
3.         viewModel.listaCifrados,
4.         viewModel.listaResultados
5.     ){ holder ->
6.         val intent=Intent() // creamos el Intent
7.         intent.action= Intent.ACTION_SEND // lo rellenamos con la información a enviar
8.         intent.type="text/plain"
9.         intent.putExtra(Intent.EXTRA_TEXT,holder.mensaje)
10.        binding.root.context.startActivity(
11.            Intent.createChooser(intent,null)
12.        ) // iniciamos el Intent
13.    }
14. }

```

- Ejecuta la app y comprueba que al pulsar en uno de los resultados, la ventana que se abre permite elegir una persona y una app.