IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Programación multimedia y dispositivos móviles

UD06. Fragments. Ciclo de vida de una Actividad y un Fragment. Notificaciones.

Desarrollo de Aplicaciones Multiplataforma

Anna Sanchis Perales

ÍNDICE

1. OBJETIVOS

Los objetivos a conocer y comprender son:

- Describir cómo la interfaz de usuario en una aplicación Android estará formada por un conjunto de actividades y de fragments.
- Ver la diferencia entre los fragments estáticos y los fragments dinámicos.
- Entender el ciclo de vida de los fragments.
- Utilizar diferentes tipos de notificaciones en nuestras aplicaciones.

2. FRAGMENTS

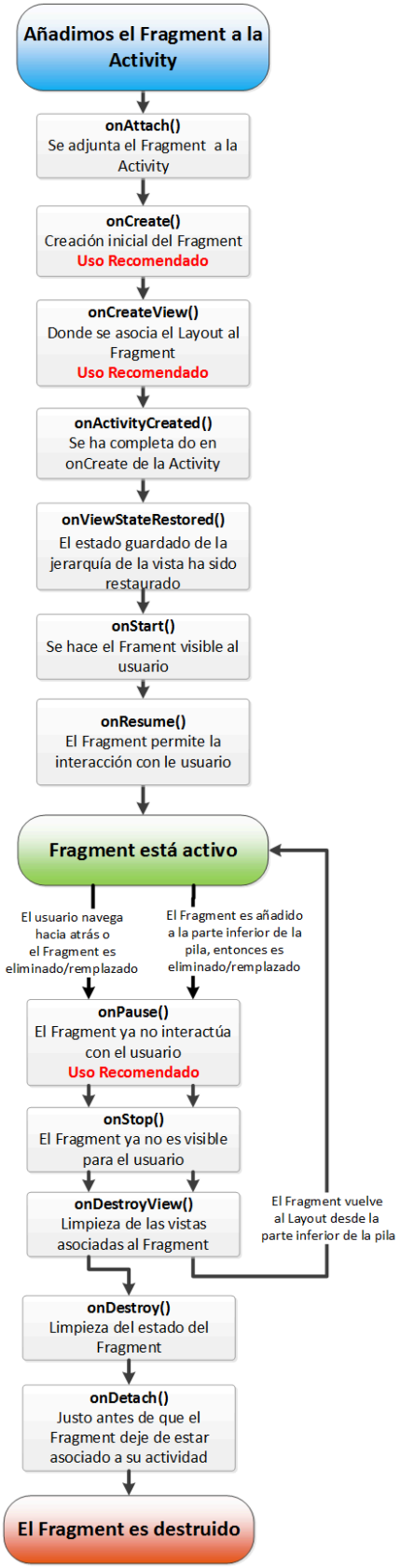
Cuando empezaron a aparecer dispositivos de gran tamaño tipo tablet, el equipo de Android tuvo que solucionar el problema de la adaptación de la interfaz gráfica de las aplicaciones a ese nuevo tipo de pantallas. Una interfaz de usuario diseñada para un teléfono móvil no se adapta fácilmente a una pantalla varias pulgadas mayores. La solución a esto vino en forma de un nuevo tipo de componente llamado Fragment.

Un Fragment no puede considerarse ni un control ni un contenedor, aunque se parecería más a lo segundo. **Un Fragment podría definirse como una porción de la interfaz de usuario que puede añadirse o eliminarse de la interfaz de forma independiente al resto de elementos de la actividad, y que por supuesto puede reutilizarse en otras actividades.** Esto, aunque en principio puede parecer algo trivial, nos va a permitir poder dividir nuestra interfaz en varias porciones de forma que podamos diseñar diversas configuraciones de pantalla, dependiendo de su tamaño y orientación, sin tener que duplicar código en ningún momento, sino tan solo utilizando o no los distintos fragmentos para cada una de las posibles configuraciones.

Por lo tanto, **un Fragment es un trozo (o un fragmento) de una actividad.** Un fragment tiene su propio layout y su propio ciclo de vida. También, podemos observar que cada fragment trabaja su lógica de forma independiente.

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

En cuanto al ciclo de vida de un Fragment es el siguiente:



3. TIPOS DE FRAGMENTS

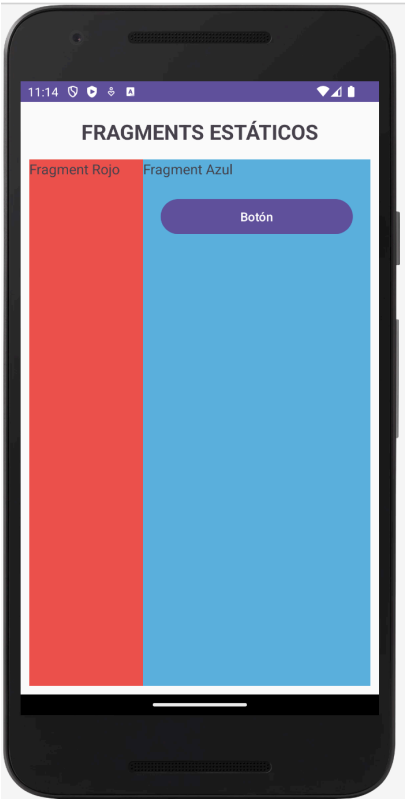
Podemos implementar los fragments de manera estática o dinámica:

- Un **fragment estático o final** es aquel que se declara en el fichero XML de la carpeta /layout directamente. Este fragment tendrá la cualidad de no ser eliminado o sustituido por nada en tiempo de ejecución. De lo contrario tendremos errores.
- Un **fragment dinámico** es el que se crea desde código Kotlin y se asocia a un FrameLayout. Este sí que se podrá eliminar o sustituir por otro fragment u otro contenido.

Más información sobre las principales diferencias.

3.1. Fragments estáticos

Vamos a realizar un pequeño ejemplo para entender mejor los fragments estáticos. En este ejemplo tendremos una actividad (MainActivity.kt) y dentro del layout de la actividad vamos a poner de forma estática los dos fragments (RedFragment.kt y BlueFragment.kt).



1. Creamos un nuevo proyecto denominada **Tema6App1**.
2. Creamos un paquete (package) denominado fragments dentro del proyecto. Este paquete no es obligatorio, pero lo vamos a crear para tener todos los fragments dentro de este paquete.
3. Creamos dos nuevos fragments con el botón derecho sobre el paquete creado anteriormente y seleccionamos **New > New Fragment > Fragment (Blank)**. Al primero le llamaremos RedFragment y al segundo BlueFragment.

   Nos creará tanto la lógica (RedFragment.java) como su layout (fragment_red.xml). La parte lógica no la modificaremos por ahora y solamente modificaremos los layouts creados.

   NOTA: Indicar que el layout de los fragments puede ser de cualquier tipo no hace falta contener dentro ninguna etiqueta <fragment> o <FrameLayout> como en los ejemplos que nos crea Android Studio por defecto. En los ejemplos dejaremos el RedFragment por defecto con algunos pequeños cambios y el BlueFragment le cambiaremos el FrameLayout por un LinearLayout en vertical como podemos observar en el código de cada uno de los layouts que tenemos a continuación:

   fragment_red.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="@android:color/holo_red_light"
   tools:context=".fragments.RedFragment">

   <!-- TODO: Update blank fragment layout -->
   <TextView
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="Fragment Rojo"
       android:textSize="16sp" />

</FrameLayout>
```

fragment_blue.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:background="@android:color/holo_blue_light"
   tools:context=".fragments.BlueFragment">

   <!-- TODO: Update blank fragment layout -->
   <TextView
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="Fragment Azul"
       android:textSize="16sp" />

</LinearLayout>
```

4. Modificamos el layout de la actividad principal (activity_main.xml). Para tener dentro los dos contenedores que serán los fragments dentro de un LinearLayout.

```
<?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"
   android:padding="10dp"
   tools:context=".MainActivity">

   <TextView
       android:id="@+id/textView"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="8dp"
       android:text="FRAGMENTS ESTÁTICOS"
       android:textSize="24sp"
       android:textStyle="bold"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintHorizontal_bias="0.5"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent" />

   <LinearLayout
```

```
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginTop="16dp"
        android:orientation="horizontal"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"
        android:baselineAligned="false">

        <androidx.fragment.app.FragmentContainerView
            android:id="@+id/fragRojo"
            android:name="com.example.tema6app1.fragments.RedFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_weight="2"
            tools:layout="@layout/fragment_red" />

        <androidx.fragment.app.FragmentContainerView
            android:id="@+id/fragAzulo"
            android:name="com.example.tema6app1.fragments.BlueFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_weight="1"
            tools:layout="@layout/fragment_blue" />

    </LinearLayout>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Comentar algunas de las etiquetas utilizadas:

- **androidx.fragment.app.FragmentContainerView**

  Etiqueta para poder insertar un fragment dentro de un layout (se puede poner la etiqueta <fragment> pero se recomienda utilizar la etiqueta anterior).

- **android:name="com.example.tema6app1.fragments.RedFragment"**

  Asocia el FragmentContainerView a su lógica.

- **tools:layout="@layout/fragment_red"**

  Asocia el FragmentContainerView a su layout.

5. Vamos a agregar un evento en el fragment azul. Para ello modificamos el layout para que tenga un botón que será al cual le agregamos el evento onClick().

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="@android:color/holo_blue_light"
    tools:context=".fragments.BlueFragment">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Fragment Azul"
        android:textSize="16sp" />

    <Button
        android:id="@+id/btnFragmentAzul"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="20dp"
        android:text="Botón" />

</LinearLayout>
```

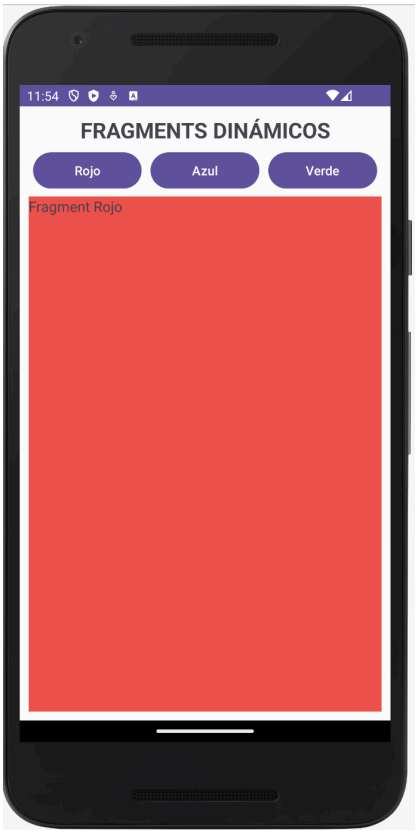
6. Modificamos el archivo BlueFragment.java para agregar la funcionalidad del botón.

```
class BlueFragment : Fragment() {

private lateinit var mBinding : FragmentBlueBinding
   override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?,
       savedInstanceState: Bundle?
   ): View? {
       mBinding = FragmentBlueBinding.inflate(inflater, container, false)

       //Gestionamos el evento del botón
       mBinding.btnFragmentAzul.setOnClickListener {
           Toast.makeText(context, "Fragment Azul presionado",
Toast.LENGTH_LONG).show()
       }
       return mBinding.root
   }
}
```

3.2. Fragments dinámicos

Como hemos comentado anteriormente, un **fragment dinámico** es el que se crea desde código Kotlin y se asocia a un FrameLayout. Este sí se puede eliminar o sustituir por otro fragment u otro contenido.

Vamos a realizar un pequeño ejemplo para entender mejor este tipo de fragments. En este ejemplo tendremos una actividad (MainActivity.kt) y cargaremos por código Java cada uno de los fragments según el botón pulsado.



1. Creamos un nuevo proyecto denominado **Tema6App2**.
2. Creamos un paquete (package) denominado fragments dentro del proyecto. Este paquete no es obligatorio, pero lo vamos a crear para tener todos los fragments dentro de este paquete.
3. Creamos tres nuevos fragments con el botón derecho sobre el paquete creado anteriormente y seleccionamos **New > New Fragment > Fragment (Blank)**. Con los siguientes nombres: RedFragment, BlueFragment y GreenFragment.

   No modificaremos la lógica de los fragments (RedFragment.java) pero si su layout (fragment_red.xml). Vemos el ejemplo del primer fragment, los otros sería realizar el mismo proceso con diferentes colores:

   fragment_red.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="@android:color/holo_red_light"
   tools:context=".fragments.RedFragment">

   <!-- TODO: Update blank fragment layout -->
   <TextView
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:text="Fragment Rojo"
       android:textSize="16sp" />

</FrameLayout>
```

4. El layout de la actividad principal (activity_main.xml) contiene lo siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:padding="10dp"
   tools:context=".MainActivity">

   <TextView
```

```
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:text="FRAGMENTS DINÁMICOS"
        android:textSize="24sp"
        android:textStyle="bold"
        tools:layout_editor_absoluteX="67dp"
        tools:layout_editor_absoluteY="18dp" />

    <LinearLayout
        android:id="@+id/linearLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:baselineAligned="false"
        android:orientation="horizontal"
        tools:layout_editor_absoluteX="10dp"
        tools:layout_editor_absoluteY="66dp">

        <Button
            android:id="@+id/btnRojo"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="5dp"
            android:layout_weight="1"
            android:text="Rojo" />

        <Button
            android:id="@+id/btnAzul"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="5dp"
            android:layout_weight="1"
            android:text="Azul" />

        <Button
            android:id="@+id/btnVerde"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="5dp"
            android:layout_weight="1"
            android:text="Verde" />

    </LinearLayout>

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/contenedorFragments"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
```

```
</LinearLayout>
```

5. La lógica de la actividad (MainActivity.kt):

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

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

        ViewCompat.setOnApplyWindowInsetsListener(mBinding.main) { v, insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right,
systemBars.bottom)
            insets
        }
        mBinding.btnRojo.setOnClickListener {
            loadFragment(RedFragment())
        }
        mBinding.btnAzul.setOnClickListener {
            loadFragment(BlueFragment())
        }
        mBinding.btnVerde.setOnClickListener {
            loadFragment(GreenFragment())
        }
    }

    private fun loadFragment(fragment: Fragment) {
        // Iniciar una transacción de fragment
        val transaction = supportFragmentManager.beginTransaction()
        transaction.replace(R.id.contenedorFragments, fragment)
        transaction.addToBackStack(null) // Agregar a la pila de retroceso
        transaction.commit()
    }
}
```

Explicación del Código en MainActivity

1. **loadFragment**: Esta función recibe un Fragment como argumento y realiza una transacción para reemplazar el fragmento que esté en contenedorFragments con el nuevo fragmento.
2. **addToBackStack(null)**: Esto permite que el fragmento actual se agregue a la pila de retroceso. Al hacer clic en el botón de retroceso, el fragmento anterior se muestra en lugar de cerrarse la actividad.

Más información:

- https://developer.android.com/guide/components/fragments?hl=es

4. FRAGMENTS LISTA-DETALLE

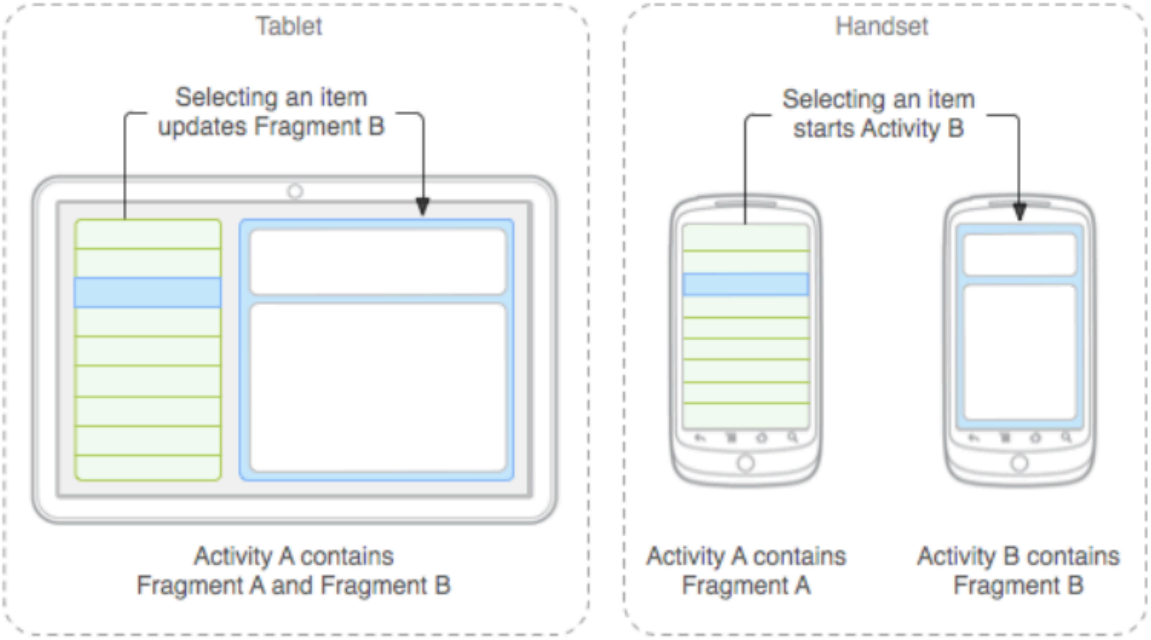
Supongamos una **aplicación de correo electrónico**, en la que, por un lado, debemos mostrar la lista de correos disponibles, con sus campos clásicos De y Asunto, y, por otro lado, debemos mostrar el contenido completo del correo seleccionado. En un teléfono móvil lo habitual será tener una primera actividad que muestre el listado de correos, y cuando el usuario seleccione uno de ellos se navegue a una nueva actividad que muestre el contenido de dicho correo. Sin embargo, en una tablet puede existir espacio suficiente para tener ambas partes de la interfaz en la misma pantalla, por ejemplo en una tablet en posición horizontal podríamos tener una columna a la izquierda con el listado de correos y dedicar la zona derecha a mostrar el detalle del correo seleccionado, todo ello sin tener que cambiar de actividad.

Antes de existir los fragments podríamos haber hecho esto implementando diferentes actividades con diferentes layouts para cada configuración de pantalla, pero esto nos habría obligado a duplicar gran parte del código en cada actividad. Tras la aparición de los fragments, colocaremos el listado de correos en un fragment y la vista de detalle en otro, cada uno de ellos acompañado de su lógica de negocio asociada, y tan sólo nos quedaría definir varios layouts para cada configuración de pantalla donde se incluyeran [o no] cada uno de estos fragments.

A modo de ejemplo, nosotros vamos a simular la aplicación de correo que hemos comentado antes, adaptándola a tres configuraciones distintas:

- Pantalla “normal” (p.e. un teléfono).
- Pantalla grande horizontal (p.e. tablet).
- Pantalla grande vertical (p.e. tablet).

Para el primer caso colocaremos el listado de correos en una actividad y el detalle en otra (o sea tendremos dos actividades, que coincide con la parte izquierda de la imagen que hay abajo), mientras que para el segundo y el tercero ambos elementos estarán en la misma actividad, a derecha/izquierda para el caso horizontal, y arriba/abajo en el caso vertical (coincide con la imagen de la izquierda).



Por lo que tendremos los siguientes casos:

- Una actividad que muestra una lista y que para ver el detalle de algún elemento lleva a otra actividad. Cada actividad tiene un fragment asociado. Podríamos haber tenido una solución con solamente una actividad que reemplaza los fragments en la misma actividad pero por motivos académicos lo vamos a realizar en dos actividades diferentes.
- Una única actividad que contiene 2 fragmentos, de forma que en la parte izquierda tenemos la lista y en la derecha el detalle del elemento seleccionado a la izquierda.

4.1. Aplicación de ejemplo: Correo electrónico

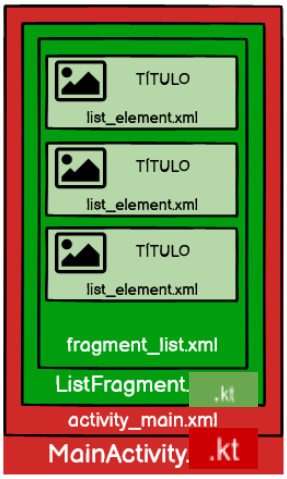
Como ya hemos comentado tendremos 3 casos:

- La visualización normal.
- La visualización en una tablet diferente para horizontal o vertical.

Vamos a tratar cada uno de ellos. En los siguientes diagramas distinguiremos las imágenes con los siguientes colores: **verde como Fragment** y **rojo como Activity**.

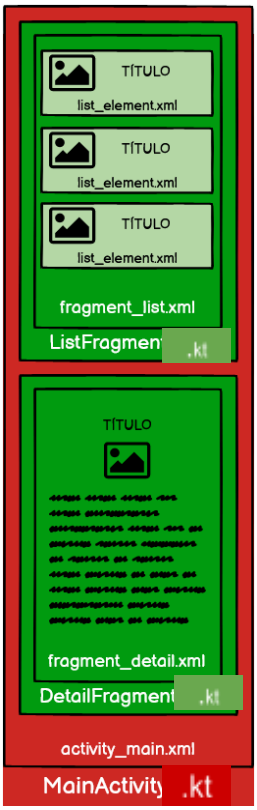
En el caso de **la visualización normal (que lo vamos a llamar caso 1)** tendremos lo siguiente:

- La **actividad principal**, la cual mostrará la lista de correos, contiene un Fragment con el listado. Aquí participan 5 elementos:
    - Actividades: MainActivity.kt y ListFragment.kt.
    - Layouts: activity_main.xml, fragment_list.xml y list_element.xml.

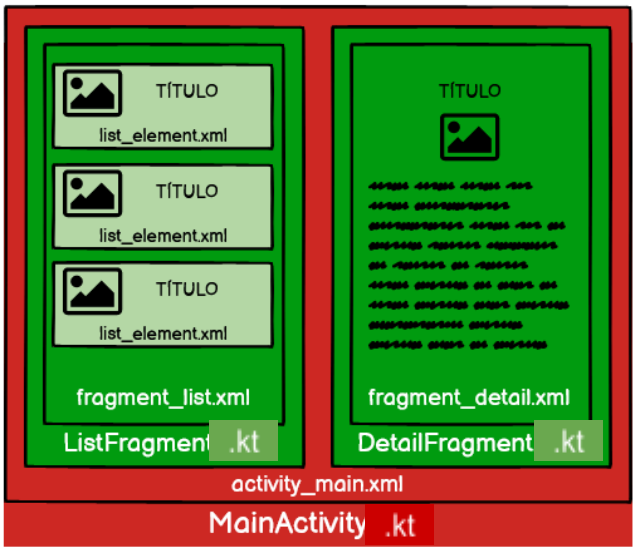


- La **actividad del detalle** del correo, que mostrará el detalle o contenido de cada correo, contiene un Fragment con dicho detalle. Aquí participan 4 elementos:
    - Actividades: DetailActivity.kt y DetailFragment.kt.
    - Layouts: activity_detail.xml y fragment_detail.xml.

En el caso de **la visualización en tablet de 10” vertical (lo llamamos caso 2)** tendremos lo siguiente:



Y en el caso de **la visualización en una tablet de 10” en horizontal (que lo vamos a llamar caso 3)** tendremos lo siguiente:



Definiremos por tanto dos fragments: uno para el listado (que se llamará ListFragment con su layout, que será fragment_list) y otro fragment para el detalle (que se llamará DetailFragment con su layout, que será fragment_detail). Ambos serán muy sencillos, por lo que, al igual que una actividad, cada fragment se compondrá de un fichero de layout XML para la interfaz (colocado en alguna carpeta /res/layout) y una clase java para la lógica asociada en el paquete fragment.

1. Creamos un nuevo proyecto denominada **Tema6App3**.
2. Creamos los siguientes paquetes: **activities**, **adapters**, **fragments**, **pojos**.
3. Movemos la actividad principal (**MainActivity.kt**) dentro del paquete activities y nos pregunta si queremos refactorizar y le decimos que si.
4. Creamos una nueva clase denominada **Correo**, que representará el POJO (Plain Old Java Object) del mismo, dentro del paquete pojos.

```
class Correo (private var de: String, private var asunto: String, private var texto:
String){

    fun getDe(): String{
        return de
    }

    fun getAsunto(): String{
        return asunto
    }

    fun getTexto(): String{
        return texto
    }
...
```

5. Creamos una clase denominada **CorreoDatos**, que representará el conjunto de correos de ejemplos del mismo, dentro del paquete pojos.

```
class CorreoDatos{
    companion object{
        val CORREOS = arrayListOf<Correo>(
            Correo("Persona 1", "Asunto del correo 1", "Texto del correo 1"),
            Correo("Persona 2", "Asunto del correo 2", "Texto del correo 2"),
            Correo("Persona 3", "Asunto del correo 3", "Texto del correo 3"),
            Correo("Persona 4", "Asunto del correo 4", "Texto del correo 4"),
            Correo("Persona 5", "Asunto del correo 5", "Texto del correo 5"),
```

```
            Correo("Persona 6", "Asunto del correo 6", "Texto del correo 6")
        )
    }
}
```

6. Crearemos la segunda actividad dentro del paquete activities con el nombre **DetailActivity**. Botón derecho sobre el paquete activities New > Activity > Empty Activity.
7. Creamos los dos fragments dentro del paquete fragments: **ListFragment** y **DetailFragment**. Botón derecho sobre el paquete fragments New > Fragment > Fragment (Blank).
8. Pasemos a hablar de la pantalla principal del caso 1: dicha pantalla solo ha de mostrar la lista de correos que se tienen. Hasta ahora metíamos la lista directamente en el layout principal y ya la teníamos, pero ahora esto no va a ser posible, ya que tenemos fragments.

   El layout principal (que se llama **activity_main.xml**) va a ser un simple contenedor del fragment, que en este caso será ListFragment.kt.

   Por ello, para indicarle al layout principal que es un simple contenedor, y que ha de cargar el fragment que le corresponde, haremos uso de la etiqueta <FragmentContainerView> con un atributo class que indique la ruta completa de la clase kotlin fragment, en este primer caso "com.example.tema6app3.fragments.ListFragment" y el layout por defecto "fragment_list". Los demás atributos utilizados son los que ya conocemos de id, layout_width y layout_height.

```
<?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=".activities.MainActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/frgListado"
        android:name="com.example.tema6app3.fragments.ListFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
```

```
       tools:layout="@layout/fragment_list" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

9. Hemos definido el primer layout que cargará un fragment, pero no hemos modificado el layout del **fragment_list.xml**. El primero de los fragment a definir contendrá solamente un control RecyclerView, para el que definiremos un adaptador personalizado para mostrar dos campos por fila (“De” y “Asunto”). Quedaría por tanto de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:padding="10dp"
   tools:context=".fragments.ListFragment">

   <androidx.recyclerview.widget.RecyclerView
       android:id="@+id/recyclerIdList"
       android:layout_width="match_parent"
       android:layout_height="match_parent" />

</LinearLayout>
```

10. En cada elemento de la lista queremos mostrar ambos datos (“De” y “Asunto”), por lo que el siguiente paso será crear un layout XML con la estructura que deseemos. En este caso vamos a mostrarlos en dos etiquetas de texto (TextView), la primera de ellas en negrita y con un tamaño de letra un poco mayor. Llamaremos a este layout: **list_item_correo.xml**, y quedaría así:

```
<?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="wrap_content"
   android:orientation="vertical">
```

```
    <TextView android:id="@+id/tvDe"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="20sp"
        tools:text="De" />

    <TextView android:id="@+id/tvAsunto"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textStyle="normal"
        android:textSize="12sp"
        tools:text="Asunto" />

</LinearLayout>
```

11. En este caso de pantalla normal (caso 1), la vista de detalle se mostrará en la segunda actividad (**activity_detail.xml**). El contenido de su layout será el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".activities.DetailActivity">

    <fragment class="com.example.tema6app3.fragments.DetailFragment"
        android:id="@+id/frgDetalle"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:layout="@layout/fragment_detail"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

12. El segundo fragment se encargará de mostrar la vista de detalle. La definición de este fragment será aún más sencilla que el anterior fragment. Su layout (**fragment_detail.xml**) se compondrá de un cuadro de texto:

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

```
    <TextView
        android:id="@+id/tvDetalle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="20sp"
        tools:text="Detalle"/>

</LinearLayout>
```

Ya tenemos creados todos los layouts.

13. Creamos una nueva clase que se llamará **CorreoAdapter.kt** dentro del paquete adapters, que será el adaptador que necesita el RecyclerView, el cual será personalizado para mostrar dos campos por fila (“De” y “Asunto”). La clase contendrá el siguiente código:

```
class CorreoAdapter (private val correos: List<Correo>, private val listener:
OnClickListener): RecyclerView.Adapter<CorreoAdapter.ViewHolder>(){

    private lateinit var context: Context

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        context = parent.context
        val view = LayoutInflater.from(context).inflate(R.layout.list_item_correo,
parent, false)
        return ViewHolder(view)
    }

    override fun getItemCount(): Int = correos.size


    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
       val correo = correos.get(position)
        with(holder){
            setListener(correo)
            binding.tvDe.text = correo.getDe()
            binding.tvAsunto.text = correo.getAsunto()
        }
    }


    inner class ViewHolder(view: View):RecyclerView.ViewHolder(view){
        val binding = ListItemCorreoBinding.bind(view)

        fun setListener(correo:Correo){
            binding.root.setOnClickListener {
                listener.onClick(correo)
```

```
            }
        }
    }
}
```

14. Ahora modificamos el código del fragment del listado (**ListFragment.kt**) que estará en el paquete fragments y contendrá el siguiente código:

```
class ListFragment : Fragment(), OnClickListener {

    private lateinit var correoAdapter: CorreoAdapter
    private lateinit var linearLayoutManager: LinearLayoutManager
    private lateinit var itemDecoration: DividerItemDecoration

    private lateinit var binding: FragmentListBinding

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

        binding = FragmentListBinding.inflate(inflater, container, false)

        correoAdapter = CorreoAdapter(Correo.CorreoDatos.CORREOS, this)
        linearLayoutManager = LinearLayoutManager(context)
        itemDecoration = DividerItemDecoration(context, DividerItemDecoration.VERTICAL)
        binding.recyclerIdList.apply{
            layoutManager = linearLayoutManager
            adapter = correoAdapter
            addItemDecoration(itemDecoration)
        }


        return binding.root

    }

    override fun onClick(c: Correo) {
        TODO("Not yet implemented")
    }
}
```

El método onCreateView() de la clase ListFragment, es el “equivalente” al onCreate() de las actividades, y dentro de él es donde normalmente asignaremos un layout determinado al fragment. En este caso tendremos que “inflarlo” (convertir el XML en la estructura de

objetos equivalente) mediante el método inflate() pasándole como parámetro el ID del layout correspondiente, en nuestro caso fragment_list.

15. Ahora trabajamos sobre la lógica del fragment de detalle (**DetailFragment.kt**) y contendrá el siguiente código:

```
class DetailFragment : Fragment() {

   private lateinit var binding: FragmentDetailBinding

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

   // Creamos un método público que nos ayude a asignar el contenido del correo
   fun mostrarDetalle(texto: String) {
      binding.tvDetalle.text = texto
   }
}
```

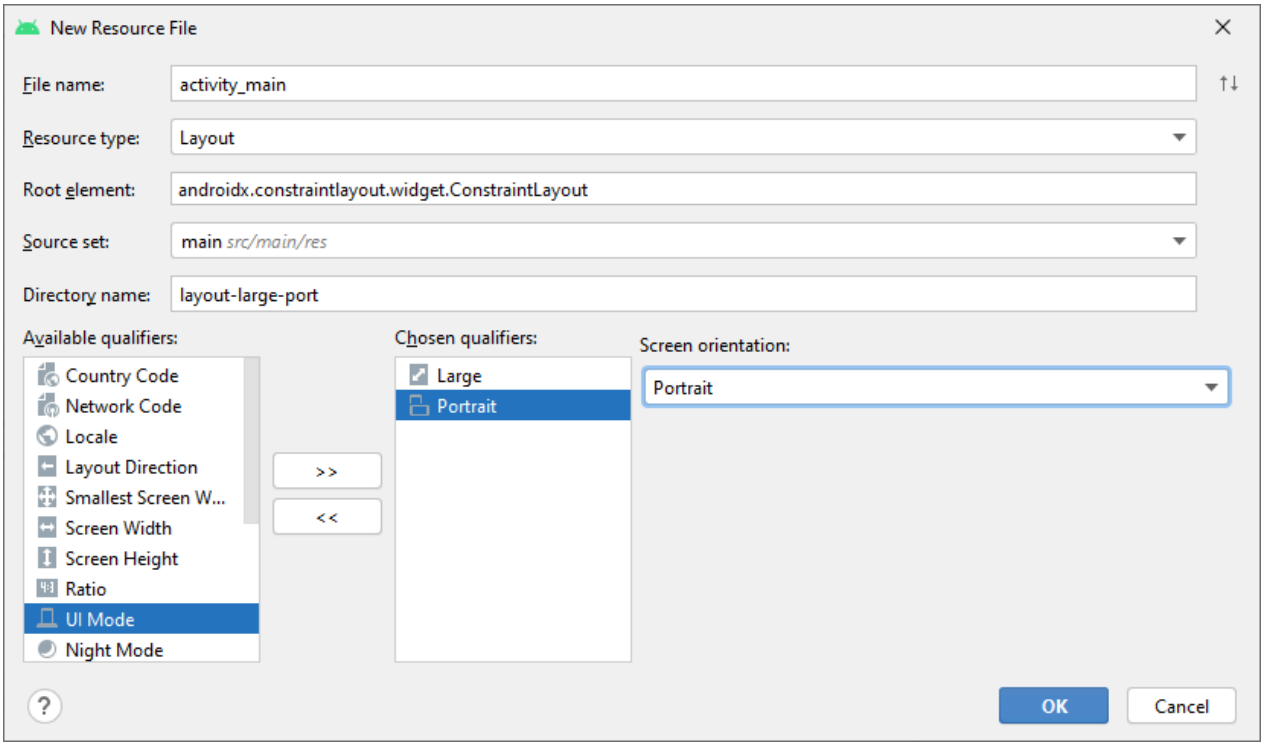
Se limitará a inflar el layout del detalle. Adicionalmente añadiremos un método, llamado mostrarDetalle(), que nos ayude posteriormente a asignar el contenido a mostrar en el cuadro de texto.

16. En el caso de la actividad **MainActivity.kt** se quedará como está por defecto.
17. Y en el caso de la actividad **DetailActivity.kt** pasa exáctamente lo mismo, que se queda como está.
18. Pero para poder adaptarse a los 3 casos es necesario que para la actividad principal existan **3 layouts diferentes**:
    - Caso 1: cuando la aplicación se ejecute en una pantalla normal (un teléfono móvil).
    - Caso 2: para pantalla grande y orientación vertical (una tablet).
    - Caso 3: para pantalla grande y orientación horizontal (una tablet).

    Todos se llamarán **activity_main.xml** (de hecho ya tenemos uno creado en la carpeta /res/layout/), y lo que marcará la diferencia será la carpeta en la que colocaremos cada uno.

Así, el primero de ellos lo colocaremos en la carpeta por defecto **/res/layout**, y los otros dos en las carpetas **/res/layout-large** (pantalla grande) y **/res/layout-large-land** (pantalla grande con orientación horizontal) respectivamente, por lo que es necesario crear estas dos carpetas. De esta forma, según el tamaño y orientación de la pantalla Android utilizará un layout u otro de forma automática sin que nosotros tengamos que hacer nada.

Para crear estas nuevas carpetas, nos situamos sobre la carpeta /res y con el botón derecho New > Android Resource File:



19. Por su parte, el layout para **/res/layout-large-land**, que es el caso de pantalla grande horizontal, será de la siguiente forma (recordar que el fichero se llamará también 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:orientation="horizontal"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
```

```
    <androidx.fragment.app.FragmentContainerView
        class="com.example.tema6app3.fragments.ListFragment"
        android:id="@+id/frgListado"
        android:layout_weight="30"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        tools:layout="@layout/fragment_list" />

    <androidx.fragment.app.FragmentContainerView
        class="com.example.tema6app3.fragments.DetailFragment"
        android:id="@+id/frgDetalle"
        android:layout_weight="70"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        tools:layout="@layout/fragment_detail" />

</LinearLayout>
```

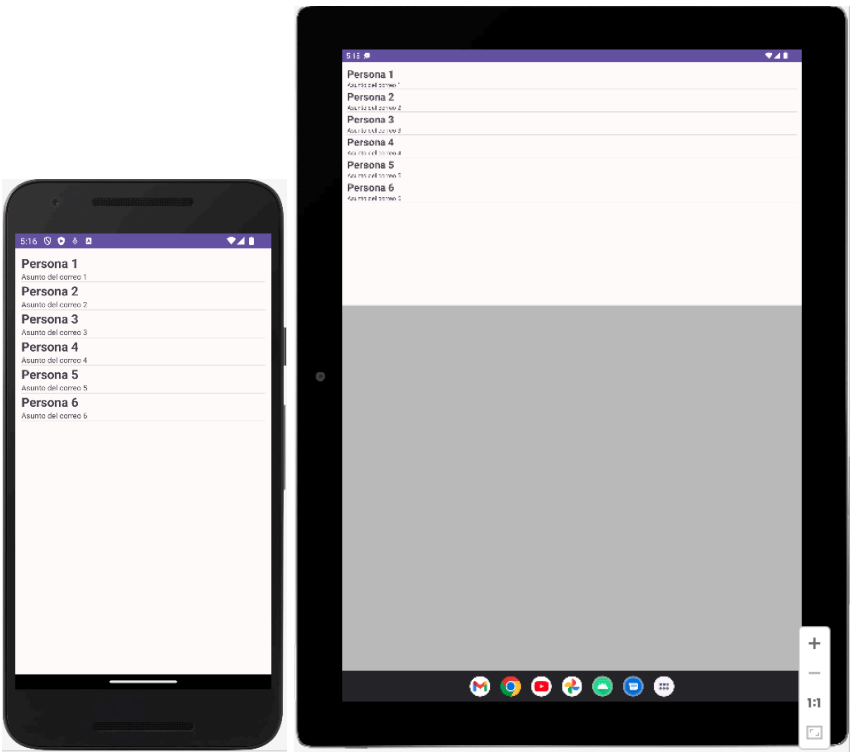
Como veis en este caso incluimos los dos fragment en la misma pantalla, ya que tendremos espacio de sobra, ambos dentro de un LinearLayout horizontal, asignando al primero de ellos un peso (propiedad layout_weight) de 30 y al segundo de 70 para que la columna de listado ocupe un 30% de la pantalla a la izquierda y la de detalle ocupe el resto.

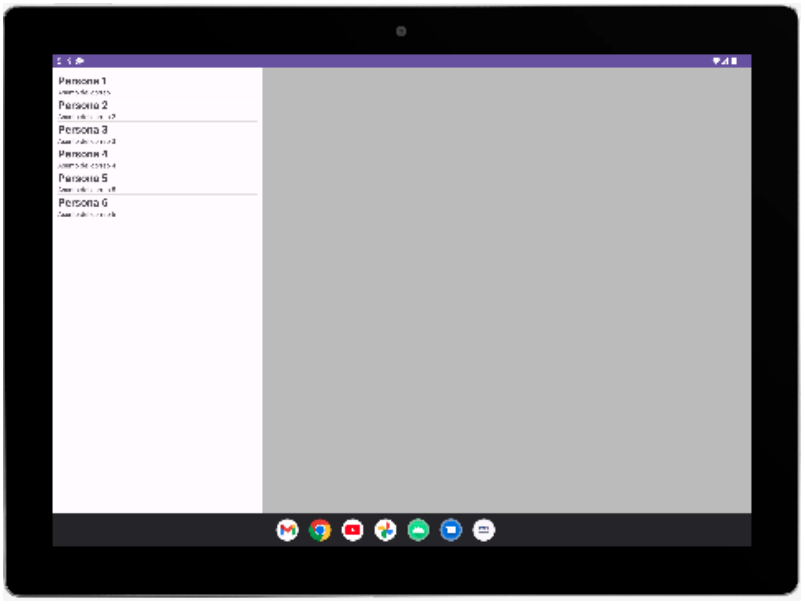
20. Por último, para el caso de pantalla grande vertical, el layout para **/res/layout-large** será prácticamente igual, sólo que usaremos un LinearLayout vertical.

```
<?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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.fragment.app.FragmentContainerView
        class="com.example.tema6app3.fragments.ListFragment"
        android:id="@+id/frgListado"
        android:layout_weight="40"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        tools:layout="@layout/fragment_list" />

    <androidx.fragment.app.FragmentContainerView
        class="com.example.tema6app3.fragments.DetailFragment"
        android:id="@+id/frgDetalle"
```

```
        android:layout_weight="60"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        tools:layout="@layout/fragment_detail" />

</LinearLayout>
```

21. Hecho esto ejecutamos la aplicación en dos AVD, uno que será nuestro móvil de 5” y otro que será una tablet de 10”. La primera imagen es del móvil, mientras que la segunda es de la tablet de 10” en vertical y la tercera de la tablet de 10” en horizontal.





Como vemos en las imágenes anteriores, la interfaz se ha adaptado perfectamente a la pantalla en cada caso, mostrándose uno o ambos fragments, y en caso de mostrarse ambos distribuyéndose horizontal o verticalmente.

4.1.1. Agregar evento onClick()

Lo que aún no hemos implementado en la lógica de la aplicación es lo que debe ocurrir al pulsarse un elemento de la lista de correos. Para ello, hemos preparado nuestro adaptador (CorreoAdapter) para que podamos agregar el evento onClick() cuando presionamos un ítem de nuestro listado. Ahora tenemos que modificar el evento onClick() dentro de la lógica ListFragment.kt. Lo que hagamos al capturar este evento dependerá de si en la pantalla se está viendo el fragment de detalle o no:

1. Si existe el fragment de detalle, habría que obtener una referencia a él y llamar a su método mostrarDetalle() con el texto del correo seleccionado.
2. En caso contrario, tendríamos que navegar a la actividad secundaria DetalleActivity para mostrar el detalle.

Sin embargo, existe un problema, un fragment no tiene por qué conocer la existencia de ningún otro, es más, deberían diseñarse de tal forma que fueran lo más independientes posible, de forma que puedan reutilizarse en distintas situaciones sin problemas de dependencias con otros elementos de la interfaz. Por este motivo, el patrón utilizado normalmente en estas circunstancias

no será tratar el evento en el propio fragment, sino definir y lanzar un evento personalizado al pulsarse el ítem de la lista y delegar a la actividad contenedora la lógica del evento, ya que ella sí debe conocer qué fragments componen su interfaz. ¿Cómo hacemos esto? Pues de forma análoga a cuando definimos eventos personalizados para un control, definiendo una interfaz con el método asociado al evento, en este caso llamada CorreosListener con un único método llamado onCorreoSeleccionado(), declaramos un atributo de la clase con esta interfaz y definimos un método setXXXListener() para poder asignar el evento desde fuera de la clase. Veamos cómo sería:

1. Creamos una interfaz dentro del paquete fragment denominada **CorreosListener.kt**, que definirá el método asociado al evento de seleccionar un correo de la lista.

```
interface CorreosListener {
   fun onCorreoSeleccionado(c: Correo)
}
```

2. Ahora nos vamos a la clase **ListFragment.kt** donde realizaremos los siguientes cambios:

   Declararemos un atributo de la interfaz:

```
private lateinit var listener: CorreosListener
```

   Y ahora definiremos el método setCorreosListener() para poder asignar el evento desde fuera de la clase:

```
fun setCorreosListener(listener: CorreosListener) {
   this.listener = listener
}
```

   Ahora lo único que deberemos hacer en el evento onClick() de la lista será lanzar nuestro evento personalizado onCorreoSeleccionado() pasándole como parámetro el contenido del correo. Por lo que modificamos el onClick() de la clase ListFragment:

```
// Para realizar onClick
override fun onClick(correo: Correo) {
   Toast.makeText(context, "Seleccion: ${correo.getDe()}", Toast.LENGTH_LONG).show()
```

```
    if (listener != null) {
        listener.onCorreoSeleccionado(correo)
    }


}
```

3. Hecho esto, el siguiente paso será tratar este evento en la clase Kotlin de nuestra actividad principal. Para ello, en el onCreate() de nuestra actividad principal, que es **MainActivity.kt**, obtendremos una referencia al fragment de la lista mediante el método findFragmentById() del Fragment Manager (componente encargado de gestionar los fragments) y asignaremos el evento que escucha el método setCorreosListener() que acabamos de definir. Así que nos vamos a MainActivity.kt y modificamos el método onCreate() de la siguiente forma (recordar que hay que poner en la clase el implements CorreosListener, y al hacerlo nos saldrá un error indicando que tenemos métodos que no han sido implementados, por lo que debemos solucionar esto declarando dichos métodos):

```
class MainActivity : AppCompatActivity(), CorreosListener{

    private lateinit var binding: ActivityMainBinding


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

        val frgListado: ListFragment =
supportFragmentManager.findFragmentById(binding.frgListado.id) as ListFragment
        frgListado.setCorreosListener(this)

    }

    override fun onCorreoSeleccionado(correo: Correo) {

    }
}
```

4. La mayor parte del interés de la clase anterior está en el método **onCorreoSeleccionado()**.

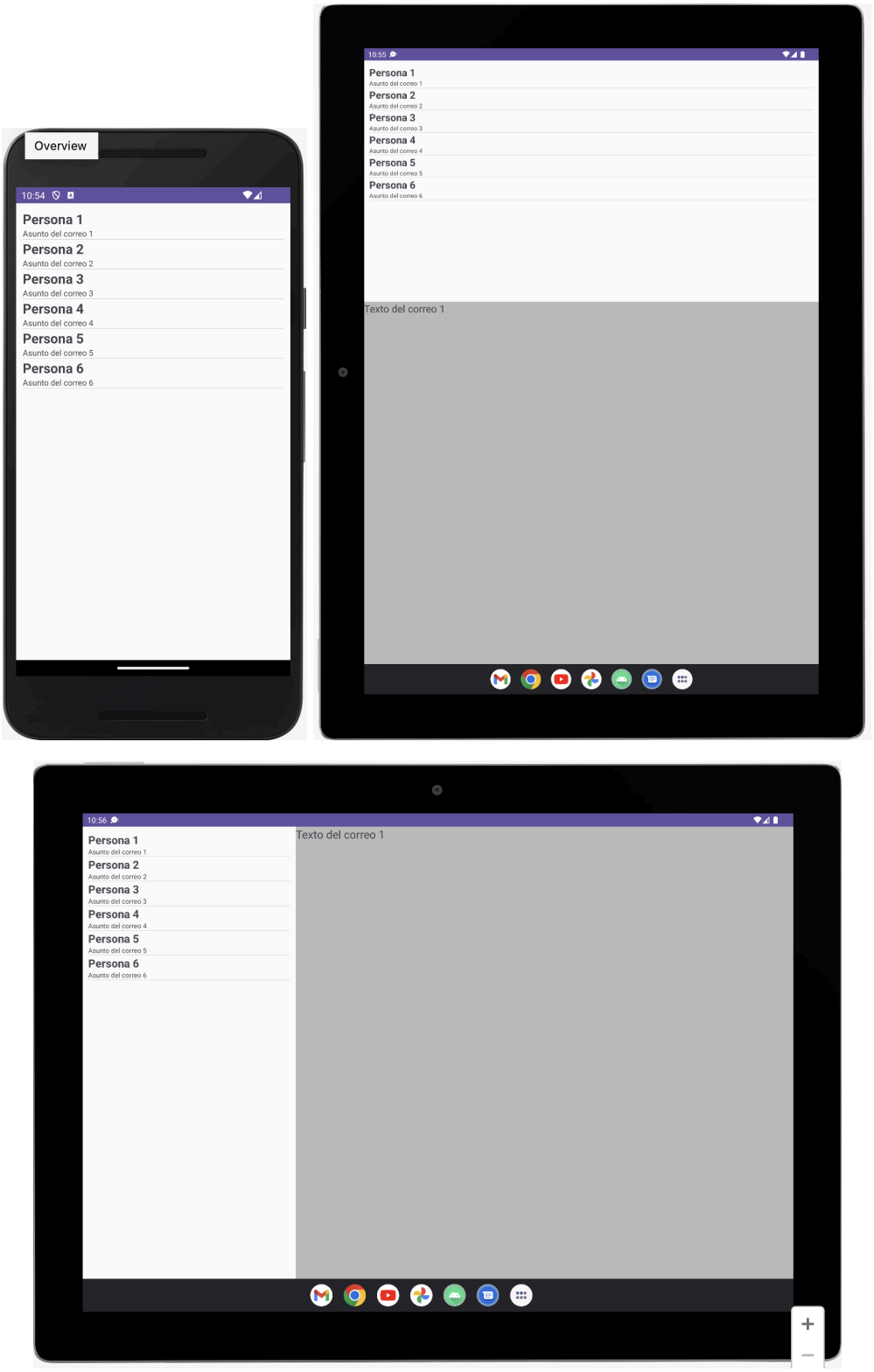
Este es el método que se ejecutará cuando el fragment de listado nos avise de que se ha seleccionado un determinado ítem de la lista. Esta vez sí, la lógica será la ya mencionada, es decir, si en la pantalla existe el fragment de detalle, simplemente lo actualizaremos mediante mostrarDetalle() y en caso contrario navegaremos a la actividad DetailActivity.kt. Para este segundo caso, crearemos un nuevo Intent con la referencia a dicha clase, y le añadiremos como parámetro extra un campo de texto con el contenido del correo seleccionado. Finalmente llamamos a startActivity() para iniciar la nueva actividad. El código del método será el siguiente:

```
override fun onCorreoSeleccionado(correo: Correo) {

/*val screenSize = resources.configuration.screenLayout and
Configuration.SCREENLAYOUT_SIZE_MASK
val isTablet = screenSize == Configuration.SCREENLAYOUT_SIZE_LARGE || screenSize ==
Configuration.SCREENLAYOUT_SIZE_XLARGE

val isPhone = resources.configuration.uiMode and Configuration.UI_MODE_TYPE_MASK ==
Configuration.SCREENLAYOUT_SIZE_NORMAL

var hayDetalle = isTablet && !isPhone*/


   if (correo != null) {
      var hayDetalle = supportFragmentManager.findFragmentById(R.id.frgDetalle) !=
null


       if(hayDetalle){//Se muestra el contenido en la misma Activity

           val detailFragment = DetailFragment()
           val transaction = supportFragmentManager.beginTransaction()
           transaction.replace(R.id.frgDetalle, detailFragment)
           transaction.commitNow()
           detailFragment.mostrarDetalle(correo.getTexto())

       }else{
           val i = Intent(this, DetailActivity::class.java)
           i.putExtra("TextoDetalle", correo.getTexto())
           startActivity(i)
       }
   }
}
```

5. Y ya sólo nos queda comentar la implementación de esta segunda actividad, **DetailActivity.kt**, por lo que crearemos la clase. El código será muy sencillo, y se limitará a recuperar el parámetro extra pasado desde la actividad anterior y mostrarlo en el fragment de detalle mediante su método mostrarDetalle(), todo ello dentro de su método onCreate().

```
class DetailActivity : AppCompatActivity() {

   private lateinit var binding : ActivityDetailBinding


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

       val detalle = supportFragmentManager
           .findFragmentById(R.id.frgDetalle) as DetailFragment?

       intent.getStringExtra("TextoDetalle")?.let { detalle?.mostrarDetalle(it) }

   }
}
```

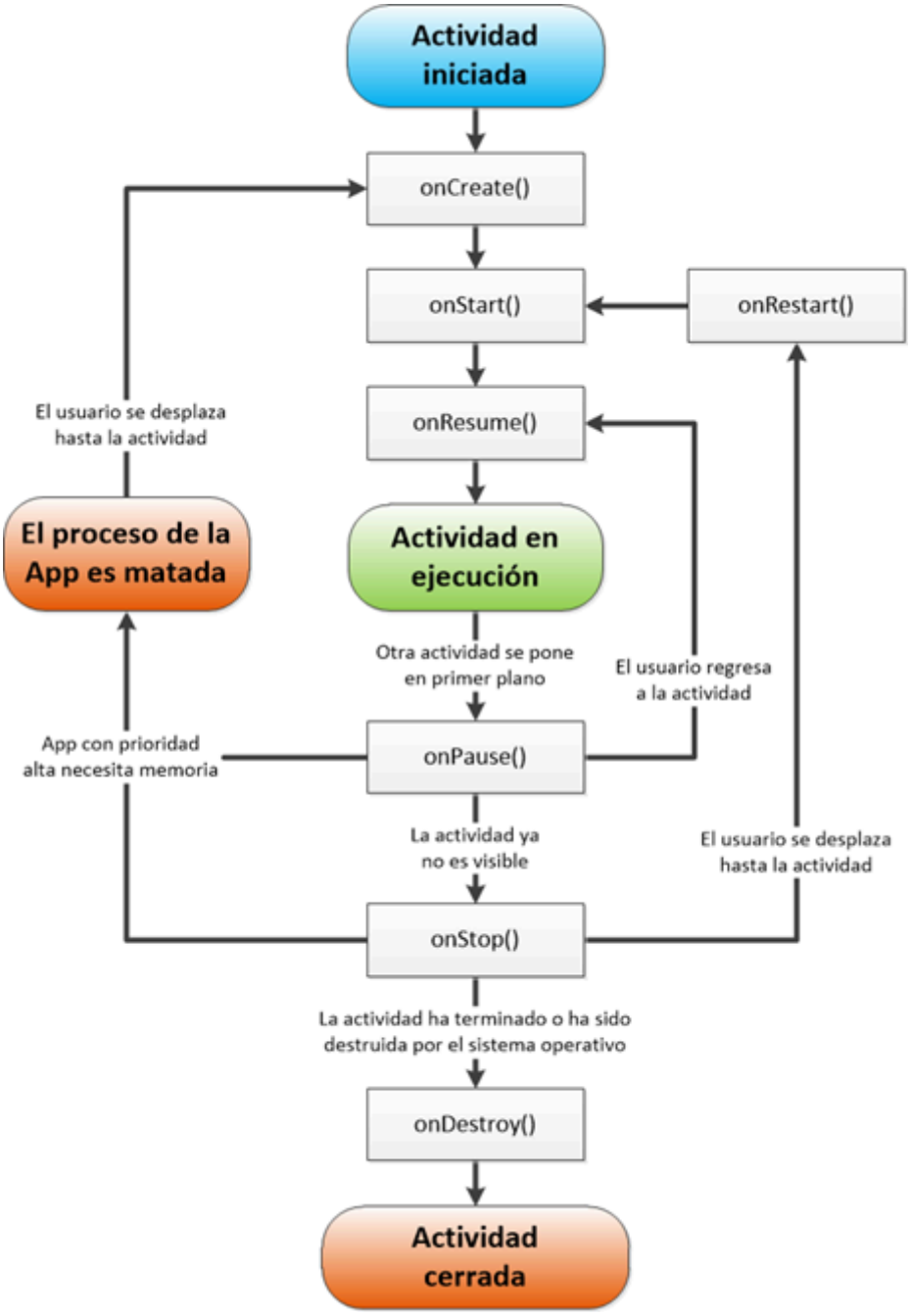
6. Si ejecutamos de nuevo la aplicación, comprobaremos el funcionamiento de la selección en las distintas configuraciones de la pantalla:



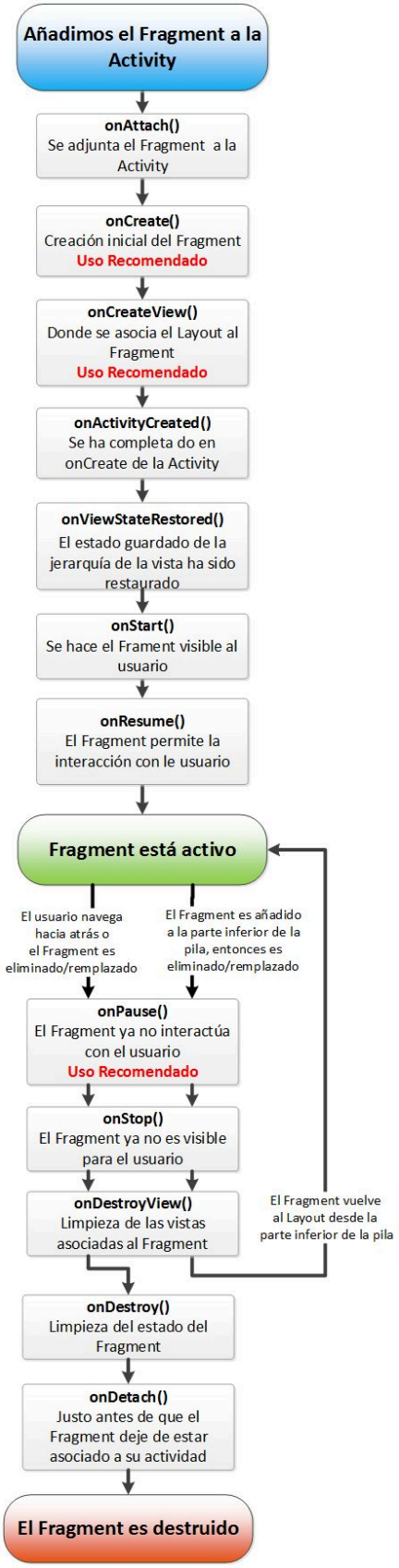
5. CICLO DE VIDA DE UNA ACTIVIDAD Y UN FRAGMENT

Vamos a recordar el ciclo de vida de una Actividad, y la relacionamos con el ciclo de vida de un Fragment, con el objetivo de entender las diferencias de los mismos.

Recordamos que el ciclo de vida de una actividad es el siguiente:



En cuanto al ciclo de vida del fragment es el siguiente:



Vamos a crear un proyecto nuevo con un fragment que se cargará en la actividad principal y lo vamos a utilizar para mostrar los mensajes del ciclo de vida. Por ello hacemos lo siguiente:

1. Abrimos el proyecto **Tema6App4**.
2. Creamos un paquete nuevo llamado **fragments**.
3. Creamos dentro del paquete fragments un nuevo fragment con el nombre **BlueFragment**.
4. Modificamos el layout del fragment (**fragment_blue.xml**) para que el fondo sea azul.

```
android:background="@android:color/holo_blue_light"
```

5. Modificamos el layout de la actividad principal (**activity_main.xml**) para contener al fragment.

```
<?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=".MainActivity">

   <androidx.fragment.app.FragmentContainerView
       android:id="@+id/fragAzulo"
       android:name="com.example.tema6app4.fragments.BlueFragment"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:layout_weight="1"
       tools:layout="@layout/fragment_blue" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

6. Abrimos la clase **BlueFragment.kt** que se encuentra dentro del paquete fragments.

   Modificamos la clase añadiendo la orden Log.v() en todos los métodos existentes del Fragment, de forma que el código de la clase será el siguiente:

```
class BlueFragment : Fragment() {


   override fun onCreate(savedInstanceState: Bundle?) {
       Log.v(BlueFragment::class.java.simpleName, "onCreate")
```

```
        super.onCreate(savedInstanceState)
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        Log.v(BlueFragment::class.java.simpleName, "onCreateView")
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_blue, container, false)
    }

    override fun onActivityCreated(savedInstanceState: Bundle?) {
        Log.v(BlueFragment::class.java.simpleName, "onActivityCreated")
        super.onActivityCreated(savedInstanceState)
    }

    override fun onAttach(context: Context) {
        Log.v(BlueFragment::class.java.simpleName, "onAttach")
        super.onAttach(context)
    }

    override fun onViewStateRestored(savedInstanceState: Bundle?) {
        Log.v(BlueFragment::class.java.simpleName, "onViewStateRestored")
        super.onViewStateRestored(savedInstanceState)
    }

    override fun onStart() {
        Log.v(BlueFragment::class.java.simpleName, "onStart")
        super.onStart()
    }

    override fun onResume() {
        Log.v(BlueFragment::class.java.simpleName, "onResume")
        super.onResume()
    }

    override fun onPause() {
        Log.v(BlueFragment::class.java.simpleName, "onPause")
        super.onPause()
    }

    override fun onStop() {
        Log.v(BlueFragment::class.java.simpleName, "onStop")
        super.onStop()
    }

    override fun onDestroyView() {
        Log.v(BlueFragment::class.java.simpleName, "onDestroyView")
        super.onDestroyView()
```

```
    }

    override fun onDestroy() {
        Log.v(BlueFragment::class.java.simpleName, "onDestroy")
        super.onDestroy()
    }

    override fun onDetach() {
        Log.v(BlueFragment::class.java.simpleName, "onDetach")
        super.onDetach()
    }

}
```

7. Si lo ejecutamos y comprobamos el Android Monitor podemos ver como en el Logcat va pasando por los distintos estados de la aplicación:

```
2023-11-07 23:14:15.658  2557-2557  BlueFragment            com.example.tema6app4  V  onAttach
2023-11-07 23:14:15.714  2557-2557  BlueFragment            com.example.tema6app4  V  onCreate
2023-11-07 23:14:15.789  2557-2557  BlueFragment            com.example.tema6app4  V  onCreateView
2023-11-07 23:14:15.841  2557-2557  Compatibil...geReporter com.example.tema6app4  D  Compat change id reported: 210923482; UID 10192; state: ENABLED
2023-11-07 23:14:15.877  2557-2557  BlueFragment            com.example.tema6app4  V  onActivityCreated
2023-11-07 23:14:15.877  2557-2557  BlueFragment            com.example.tema6app4  V  onViewStateRestored
2023-11-07 23:14:15.877  2557-2557  BlueFragment            com.example.tema6app4  V  onStart
2023-11-07 23:14:15.920  2557-2557  BlueFragment            com.example.tema6app4  V  onResume
2023-11-07 23:14:15.984  2557-2557  Compatibil...geReporter com.example.tema6app4  D  Compat change id reported: 237531167; UID 10192; state: DISABLED
```

6. NOTIFICACIONES

Las notificaciones se usan para mostrar algún mensaje de interés al usuario. Existen varios tipos de notificaciones que podremos implementar en Android, que son: las llamadas Toast, notificaciones en forma de cuadros de diálogos y Snackbar.

Explicaremos estos tres tipos de notificaciones a continuación, dando ejemplos de uso para cada uno de ellos.

6.1. Notificaciones Toast

Son las notificaciones más sencillas que podremos usar en Android. Básicamente son mensajes de texto que se muestran en la pantalla durante un periodo de tiempo corto o largo y que desaparecen automáticamente transcurrido dicho periodo.

Se usan simplemente para mostrar mensajes informativos rápidos y sencillos que no requieren intervención por parte del usuario para continuar con la ejecución de la aplicación. Al no requerir confirmación del usuario, no se aconseja su uso para notificaciones importantes.

Para crear un mensaje de estos, usaremos la clase **Toast**. Esta clase dispone de un método estático que nos facilitará la creación de un mensaje, éste es makeText(). Este método recibe como parámetros el contexto de la activity, el mensaje a mostrar y la duración del periodo de tiempo que permanecerá este mensaje visible al usuario, que puede tomar dos valores Toast.LENGTH_SHORT o Toast.LENGTH_LONG, si lo que queremos es que sea durante un tiempo corto o largo, respectivamente. Una vez creado el texto, ya solo queda mostrarlo al usuario, para ello llamamos al método show() de la clase Toast.

Por defecto este mensaje aparecerá en la parte inferior de la pantalla, centrado y con el texto de color blanco sobre un fondo grisáceo.

Veamos un ejemplo del funcionamiento de este tipo de notificación. En este ejemplo tendremos un botón que al ser pulsado mostrará el mensaje “Toast por defecto”.

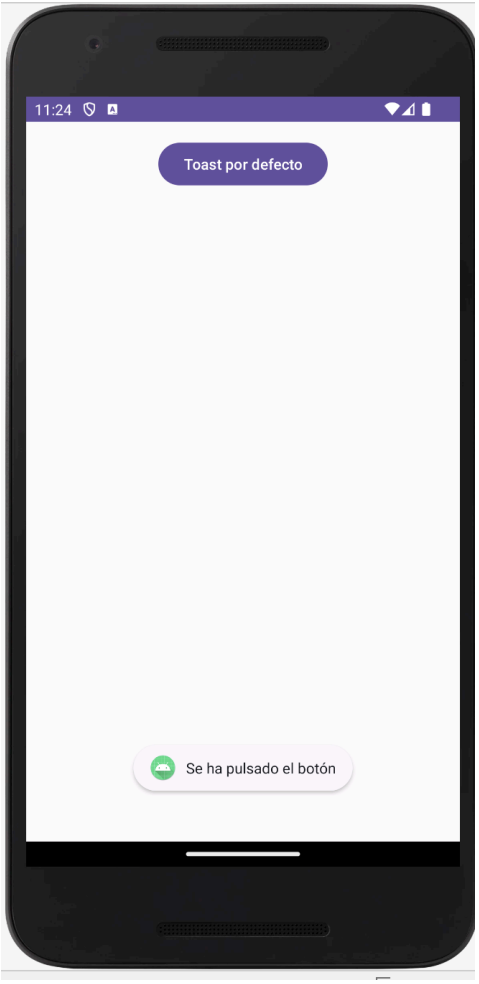
1. Creamos una nueva aplicación denominada **Tema6App5**.
2. Nos vamos al layout activity_main.xml y borramos el TextView existente. Le añadimos un botón al layout. El código quedará de la siguiente forma:

```
<?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=".MainActivity">

   <Button
       android:id="@+id/btnDefecto"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="16dp"
       android:text="Toast por defecto"
       app:layout_constraintEnd_toEndOf="parent"
       app:layout_constraintStart_toStartOf="parent"
       app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

3. Nos vamos a la clase MainActivity.java y modificamos el método onCreate():

```
class MainActivity : AppCompatActivity() {
   private lateinit var binding: ActivityMainBinding
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       binding = ActivityMainBinding.inflate(layoutInflater)
       setContentView(binding.root)

       binding.btnDefecto.setOnClickListener {
           Toast.makeText(this, "Se ha pulsado el botón", Toast.LENGTH_LONG).show()
       }
   }
```

4. El resultado de este código se puede observar en la siguiente captura.



Toast con gravity

Éste es el comportamiento por defecto de las notificaciones toast, sin embargo también podemos personalizarlo un poco cambiando su posición en la pantalla. Para esto utilizaremos el método setGravity(), al que podremos indicar en qué zona deseamos que aparezca la notificación. La zona deberemos indicar con alguna de las constantes definidas en la clase Gravity: CENTER, LEFT, BOTTOM, … o con alguna combinación de éstas.

Para nuestro ejemplo vamos a colocar la notificación en la zona central izquierda de la pantalla. Para ello, añadimos un segundo botón a la aplicación de ejemplo que muestre un toast con estas características:

1. Seguimos trabajando con el proyecto Tema6App5.
2. Añadimos un nuevo botón al layout activity_main.xml para ver el funcionamiento del Gravity. El layout le añadiremos el siguiente código:

```
...
<Button
   android:id="@+id/btnGravity"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginTop="16dp"
   android:text="Toast con gravity"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/btnDefecto" />
...
```

3. Ahora, en la clase MainActivity, en el método onCreate() modificamos el código:

```
binding.btnGravity.setOnClickListener {
   var toastGravity = Toast.makeText(this, "Toast con gravity", Toast.LENGTH_LONG)
   toastGravity.setGravity(Gravity.CENTER_HORIZONTAL, 10, 0)
   toastGravity.show()
}
```

Toast personalizado

Si esto no es suficiente y necesitamos personalizar por completo el aspecto de la notificación, Android nos ofrece la posibilidad de definir un layout XML propio para toast, donde podremos incluir todos los elementos necesarios para adaptar la notificación a nuestras necesidades. Vamos a crear un ejemplo con un layout sencillo, con una imagen y una etiqueta de texto sobre un rectángulo gris:

1. Seguimos trabajando sobre el proyecto **Tema6App5**.
2. Añadimos un nuevo botón al layout activity_main.xml para ver el funcionamiento del Toast personalizado. El layout le añadiremos el siguiente código:

```
...

<Button
```

```
   android:id="@+id/btnPersonalizado"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_marginTop="16dp"
   android:text="Toast Personalizado"
   app:layout_constraintEnd_toEndOf="parent"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintTop_toBottomOf="@+id/btnGravity" />

...
```

3. Crear el diseño del mensaje personalizado y crear su layout. En nuestro caso hemos creado el fichero toast_layout.xml con el siguiente código:

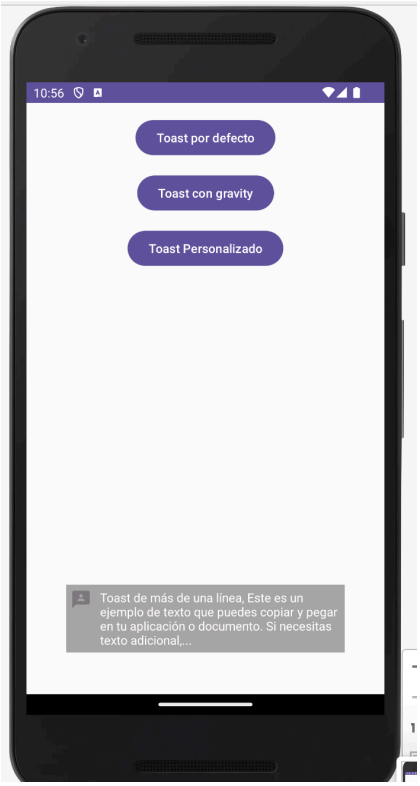
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@+id/lytLayout"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="horizontal"
   android:background="#555555"
   android:padding="5dip" >

   <ImageView android:id="@+id/imgIcono"
       android:layout_height="wrap_content"
       android:layout_width="wrap_content"
       android:src="@drawable/marcador" />

   <TextView android:id="@+id/txtMensaje"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_gravity="center_vertical"
       android:textColor="#FFFFFF"
       android:paddingLeft="10dip" />

</LinearLayout>
```

4. Para asignar este layout a nuestro toast tendremos que actuar de una forma algo diferente a las anteriores. En primer lugar deberemos inflar el layout mediante un objeto LayoutInflater. Una vez construido el layout modificaremos los valores de los distintos controles para mostrar la información que queramos. En nuestro caso, tan sólo modificaremos el mensaje de la etiqueta de texto, ya que la imagen ya la asignamos de forma estática en el layout XML

mediante el atributo android:src. Tras esto, sólo nos quedará establecer la duración de la notificación con setDuration() y asignar el layout personalizado al toast mediante el método setView(). El código sería el siguiente:

```
binding.btnPersonalizado.setOnClickListener {
   var toastPersonalizado = Toast(applicationContext)

   var inflater = layoutInflater
   var layout = inflater.inflate(R.layout.toast_layout, findViewById(R.id.lytLayout))

   val txtMsg = layout.findViewById<TextView>(R.id.txtMensaje)
   txtMsg.text = "Toast de más de una línea, Este es un ejemplo de texto que puedes
copiar y pegar en tu aplicación o documento. Si necesitas texto adicional,... "

   toastPersonalizado.duration = Toast.LENGTH_SHORT
   toastPersonalizado.view = layout
   toastPersonalizado.show()

}
```

5. Ejecutamos el proyecto y obtenemos el siguiente resultado:



6.2. Snackbar

Puedes usar un Snackbar para mostrar un mensaje breve al usuario. El mensaje desaparece automáticamente después de poco tiempo. Un Snackbar es ideal para mensajes breves en los que no es necesario que actúe el usuario. Por ejemplo, una app de correo electrónico podría usar un Snackbar para indicarle al usuario que la app envió correctamente un correo electrónico.

Para poder emplearlo, tendrás que tener una dependencia, la de material design, en el build.gradle (Module:) dentro del bloque dependencies{}. Por defecto ya está agregada, si no estuviera la agregas y luego sincroniza el proyecto (File -> Sync Proyect).

```
implementation 'com.google.android.material:material:1.4.0'
```

El uso es bastante sencillo, el formato es:

```
Snackbar.make(vista, "Mensaje", Snackbar.LENGTH_LONG|Snackbar.LENGTH_SHORT).show();
```

El problema, o lo nuevo comparado con Toast, es el primer parámetro, Toast utilizaba el contexto de la aplicación, sin embargo, Snackbar usa como objetivo la vista o View. Es decir, requiere una referencia a una vista que permita al snackbar descubrir un «contenedor adecuado» donde alojarse, este contenedor será normalmente el content view o vista raíz de la actividad.

Vemos seguidamente un ejemplo, teniendo una referencia a un Button llamada btnSnackbar, y una referencia a una vista llamada view.

```
btnSnackbar.setOnClickListener(new View.OnClickListener() {

   @Override
   public void onClick(View view) {
      Snackbar.make(view, "Texto a mostrar", Snackbar.LENGTH_LONG).show();
   }

});
```

Es evidente, que si el Snackbar en vez de estar situado en el manejador del evento, estuviera en una función externa, por ejemplo debido a usarse en múltiples lugares, como necesita saber sobre el

View, deberemos mandarle a ese método la referencia de la vista en la llamada, por lo tanto, en este caso esa llamada, desde la misma situación anterior, sería:

```
llamadaFuncion(view, "Esto es una prueba")
```

La ventaja del Snackbar frente al Toast, es que podemos añadirle al Snackbar un botón de acción, básicamente se consigue encadenando al mismo setAction(), seguido de un escuchador del evento onClick, cuya función anónima ejecuta el código deseado. Ejemplo.

```
binding.btnSnack.setOnClickListener {
   Snackbar.make(it, "Texto a mostrar", Snackbar.LENGTH_LONG)
       .setAction("Pulse para ejecutar", View.OnClickListener {
           //Código a ejecutar
       }).show()
}
```

Por otro lado, en el Snackbar podemos tunear el color del botón, en este ejemplo, usamos un color predefinido, concretamente el Color.CYAN (admite #RGB).

```
binding.btnSnack.setOnClickListener {
   Snackbar.make(it, "Texto a mostrar", Snackbar.LENGTH_LONG)
       .setActionTextColor(Color.CYAN)
       .setAction("Pulse para ejecutar", View.OnClickListener {
           //Código a ejecutar
       }).show()
}
```

NOTA: Otros colores predefinidos podrían ser Color.BLACK ,Color.BLUE, Color.DKGRAY, Color.GRAY, Color.GREEN, Color.LTGRAY, Color.MAGENTA, Color.RED, Color.TRANSPARENT, Color.WHITE, Color.YELLOW.

Más información sobre la creación de mensajes emergentes de tipo Snackbar:

- https://developer.android.com/training/snackbar/showing?hl=es

6.3. Notificaciones de tipo diálogo

Otra forma de generar notificaciones en Android es usando cuadros de diálogos. Estos cuadros de diálogos en la mayoría de los casos requieren intervención por parte del usuario para poder continuar con la ejecución de la aplicación. Se usan por lo tanto para mostrar mensajes en los que se exige una contestación por parte del usuario o para mostrar opciones en las cuales el usuario tendrá que seleccionar una o varias de ellas.

En principio, los diálogos de Android los podremos utilizar con distintos fines, en general:

- Mostrar un mensaje.
- Pedir una confirmación rápida.
- Solicitar al usuario una elección (simple o múltiple) entre varias alternativas.

De cualquier forma, veremos también cómo personalizar completamente un diálogo para adaptarlo a cualquier otra necesidad.

El uso actual de los diálogos en Android se basa en fragmets. En este caso nos vamos a basar en la clase DialogFragment. Para crear un diálogo lo primero que haremos será crear una nueva clase que herede de DialogFragment y sobreescribimos uno de sus métodos onCreateDialog(), que será el encargado de construir el diálogo con las opciones que necesitemos.

La forma de construir cada diálogo dependerá de la información y funcionalidad que necesitemos. A continuación mostraré algunas de las formas más habituales.

6.3.1. Diálogo de Alerta

MaterialAlertDialog es un componente de diseño de Material Design que se utiliza en aplicaciones Android para mostrar diálogos de alerta con un aspecto y comportamiento coherentes con las pautas de diseño de Material Design. Este componente proporciona una apariencia moderna y consistente a tus diálogos de alerta en Android.

A diferencia de los diálogos de alerta estándar proporcionados por AlertDialog en Android,

MaterialAlertDialog sigue las directrices de diseño de Material Design y ofrece una experiencia de usuario más agradable y atractiva. Los diálogos de alerta Material Design incluyen animaciones suaves, sombras y elementos visuales que hacen que la interfaz de usuario sea más atractiva.

Para utilizar MaterialAlertDialog, generalmente necesitas agregar la biblioteca de Material Design a tu proyecto y luego usar las clases y métodos proporcionados por esa biblioteca para crear tus diálogos de alerta.

El código es el que se muestra a continuación:

```
binding.btnDefecto.setOnClickListener {
   MaterialAlertDialogBuilder(this)
       .setTitle("Esto es un mensaje de alerta.")
       .setPositiveButton("Aceptar", DialogInterface.OnClickListener { dialog, i ->
           //Código a ejecutar en caso de Aceptar
           dialog.cancel()
       })
       .setNegativeButton("Cancelar", null)//Evento a null porque no va a hacer nada
       .show()
}
```

El código completo está en el proyecto **Tema6App6** del repositorio de clase.

Tras ejecutar la aplicación tenemos:

6.3.2. Diálogos de Selección

Cuando las opciones a seleccionar por el usuario no son sólo dos, como en los diálogos de confirmación, sino que el conjunto es mayor podemos utilizar los diálogos de selección para mostrar una lista de opciones entre las que el usuario pueda elegir.

Un ejemplo de uso es el que se muestra a continuación:

```
binding.btnAlertaSelec.setOnClickListener {
       val items = arrayOf("Inglés", "Español", "Francés")

       MaterialAlertDialogBuilder(this)
           .setTitle("¿Seleccione un idioma?")
           .setItems(items, DialogInterface.OnClickListener { dialog, i ->
               when(i){
                   0 -> confirmSelection()
                   1 -> Toast.makeText(this, "Spanish", Toast.LENGTH_SHORT).show()
                   2 ->Toast.makeText(this, "Francés", Toast.LENGTH_SHORT).show()
               }
           }).show()
   }
}
```

```
fun confirmSelection(){
   MaterialAlertDialogBuilder(this)
       .setTitle("Esto es un mensaje de alerta.")
       .setPositiveButton("Aceptar", DialogInterface.OnClickListener { dialog, i ->
           //Código a ejecutar en caso de Aceptar
           dialog.cancel()
       })
       .setNegativeButton("Cancelar", null)
       .show()
}
```

Tras la ejecución tenemos:

También podemos ver el código completo en el repositorio **Tema6App6**.

6.3.3. Diálogos Personalizados

Por último, vamos a comentar cómo podemos establecer completamente el aspecto de un cuadro de diálogo. Para esto vamos a actuar como si estuviéramos definiendo la interfaz de una actividad, es decir, definiremos un layout XML con los elementos a mostrar en el diálogo. En este caso vamos a definir un layout de ejemplo llamado dialog_personal.xml que colocaremos como siempre en la carpeta res/layout. Contendrá por ejemplo una imagen a la izquierda y dos líneas de texto a la derecha.

1. Abrimos el proyecto **Tema6App6**.
2. Creamos un nuevo layout dialog_personal.xml dentro de res/layout.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:orientation="horizontal"
       android:padding="3dp" >

       <ImageView
          android:id="@+id/imageView1"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:src="@mipmap/ic_launcher" />

       <LinearLayout
          android:layout_width="wrap_content"
          android:layout_height="match_parent"
          android:orientation="vertical"
          android:padding="3dp">

          <TextView
              android:id="@+id/textView1"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/dialogo_linea_1" />

          <TextView
              android:id="@+id/textView2"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="@string/dialogo_linea_2" />

       </LinearLayout>
```

```
</LinearLayout>
```

3. Nos vamos al fichero res/values/strings.xml y añadimos dos nuevos valores:

```
<string name="dialogo_linea_1"> Diálogo Línea 1 </string>
<string name="dialogo_linea_2"> Diálogo Línea 2 </string>
```

4. Vemos que el código será muy similar al explicado en el MaterialAlert, lo único que hacemos es inflar la vista utilizando el layoutInflater. Finalmente podremos incluir botones tal como vimos para los diálogos de alerta. En este caso de ejemplo incluimos un botón de Aceptar. El código de la clase será el siguiente:

```
binding.btnAlertaPerso.setOnClickListener {
   //Inflamos la vista personalizada
   val dialogView = layoutInflater.inflate(R.layout.dialog_perso, null)

   //Con el setView le pasamos la vista
   MaterialAlertDialogBuilder(this)
       .setTitle("Esto es un diálogo personalizado")
       .setView(dialogView)
       .setPositiveButton("Aceptar", DialogInterface.OnClickListener { dialog, i ->
           //Código a ejecutar en caso de Aceptar
           dialog.cancel()
       })
       .setCancelable(false)//No podrá desaparecer el diálogo por ningún motivo
       .show()
}
```

5. Si ejecutamos y pulsamos el botón obtenemos lo siguiente: