PROYECTO 2

INFORME ASIGNATURAS

Este proyecto haremos una app muy básica en la que elegiremos en una lista desplegable el nombre de un curso y tras pulsar un botón, se mostrará la lista de asignaturas de ese curso. También hay una zona para escribir observaciones y un botón que de momento, simula el envío de los datos a un servidor.

Durante su desarrollo se tratarán estos conceptos:

- Concepto de LinearLayout
- Elementos básicos de la interfaz: Spinner, Button, TextView y EditText
- Mejora básica de la apariencia: margin, padding, gravity y weight
- Acceder a los objetos de la interfaz en el código fuente de la app
- Responder a pulsaciones del botón
- Toast
- Concepto de FrameLayout
- ImageView
- Introducción al View Binding

1 – Concepto de LinearLayout

Para diseñar la interfaz de la ventana usaremos un **LinearLayout**. Todos los elementos de la interfaz que estén dentro del LinearLayout se verán uno a continuación de otro, en sentido horizontal o vertical, según como queramos

- Abre Android Studio y crea un proyecto llamado **InformeAsignaturas,** usando la plantilla **Empty Views Activity**
- Abre el archivo **res/layout/activity_main.xml** en la vista de código fuente
- Borra todo el código fuente que nos ha escrito Android Studio, de forma que comencemos a escribir desde cero en el archivo
- Añade el siguiente elemento **LinearLayout**

```
1.  <LinearLayout>
2.
3.  </LinearLayout>
```

Cuando diseñamos la interfaz de usuario, siempre hay un **elemento raíz**, que es el contenedor de todos los demás elementos de la interfaz. Dicho elemento raíz debe poseer las definiciones de los espacios de nombres **android, app** y **tools**, que son estos atributos:

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"

- Añade a la etiqueta de inicio de **LinearLayout** los atributos que definen los espacios de nombres android, app y tools, de esta forma:

```
1.  <LinearLayout
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.
6.  </LinearLayout>
```

- Añade al **LinearLayout** los atributos para que su alto y ancho sean los de la pantalla (**match_parent**)

```
<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">

</LinearLayout>
```

- Por último, configuraremos el **LinearLayout** para que todos los elementos que contengan se muestren uno tras otro en vertical. Eso se hace añadiendo el atributo **android:orientation**

```
<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">

</LinearLayout>
```

A partir de ahora, todos los elementos (textos, botones, etc) que pongamos dentro de las etiquetas <LinearLayout> y </LinearLayout>, se verán uno debajo de otro

2 – Elementos básicos de la interfaz: Spinner, Button, TextView y EditText

Vamos a añadir a la interfaz tres elementos básicos:

- **TextView:** Es un texto fijo que aparece en la pantalla
- **EditText:** Es un cuadro de texto en el que podemos escribir
- **Button:** Es un botón que puede ser pulsado.
- **Spinner:** Es una lista desplegable de valores. El usuario puede elegir uno de ellos.

Estos elementos tienen un ancho y un alto, que configuraremos con los atributos **android:with** y **android:height** de la forma que ya conocemos

- Coloca, entre las etiquetas **<LinearLayout>** y **</LinearLayout>** un elemento **Spinner**, de esta forma:

```
<Spinner
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

- A continuación, vamos a añadir un **identificador** llamado **spnCurso** al **Spinner**,

```
<Spinner
    android:id="@+id/spnCurso"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

> El identificador es muy importante porque sirve para poder acceder al **Spinner** en el archivo de código fuente **MainActivity**. El nombre del identificador puede ser el que queramos, pero se suele usar **notación húngara**[1] para los componentes de la interfaz.

- Pon en marcha la app y comprueba que aparece una lista desplegable vacía.
- Para añadir elementos a la lista desplegable, abre el archivo **res/values/strings.xml** y añade un elemento **string-array** llamado **cursos,** de esta forma:

```
<resources>
    <string name="app_name">Informe Asignaturas</string>
    <string-array name="cursos">
        <item>1º DAM</item>
        <item>2º DAM</item>
    </string-array>
</resources>
```

- Vete al **Spinner** y añádele el atributo **android:entries** para referenciar al array **cursos** añadido en el apartado anterior, así:

```
<Spinner
    android:id="@+id/spnCurso"
    android:entries="@array/cursos"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

- Ejecuta la app y comprueba que ahora podemos elegir en la lista desplegable los datos **1º DAM** y **2º DAM**
- Abre el archivo **strings.xml** y añade estos String:

Nombre del String	Texto del String
seleccionarCurso	Seleccionar curso
sinCurso	No hay ningún curso seleccionado
observaciones	Introduce observaciones
enviar	Enviar

[1] La notación húngara consiste en poner un prefijo al nombre que nos indique el tipo de lo que estamos identificando. Por ejemplo en **spnCurso** el perfijo **spn** nos indica que estamos identificando un Spinner.

- A continuación, vamos a añadir un **Button** justo a continuación del **Spinner** y lo configuraremos con estas características:

    - Identificador: **btnSeleccionarCurso**
    - Anchura y altura: la que tenga su contenido
    - Texto: el del string **seleccionarCurso**

```
<Button
    android:id="@+id/btnSeleccionarCurso"
    android:text="@string/seleccionarCurso"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Como el **Spinner** y el **Button** están dentro de un **LinearLayout** vertical, se verá primero la lista desplegable y justo debajo el botón.

- Ahora añadimos un **TextView** con las siguientes características:
    - Identificador: **txtAsignaturas**
    - Anchura: La que tenga su padre
    - Altura: La que tenga su contenido
    - Texto: El del string **sinCurso**

```
<TextView
    android:id="@+id/txtAsignaturas"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/sinCurso"/>
```

- Por último, añadimos un **EditText** con las siguientes características

    - Identificador: **txtObservaciones**
    - Anchura: La que tenga su padre
    - Altura: La que tenga su contenido

```
<EditText
    android:id="@+id/observaciones"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

- Añade al **EditText** el atributo **android:inputType** con el valor **textMultiline** para que el usuario pueda escribir múltiples líneas de texto

```
<EditText
    android:id="@+id/observaciones"
    android:inputType="textMultiLine"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

El atributo **android:inputType** sirve para indicar lo que se puede escribir en el cuadro de texto, y en consecuencia el móvil mostrará el tipo de teclado más apropiado para ello. Entre los valores que se admiten destacan:

- **text**: Una sola línea de texto
- **textMultiline:** Múltiples líneas de texto
- **textPassword:** Contraseña en la que se ocultan los símbolos que se escriben
- **number:** Número

- A continuación, pon el atributo **android:hint** al **EditText** y haz que su mensaje sea el string identificado como **observaciones**.

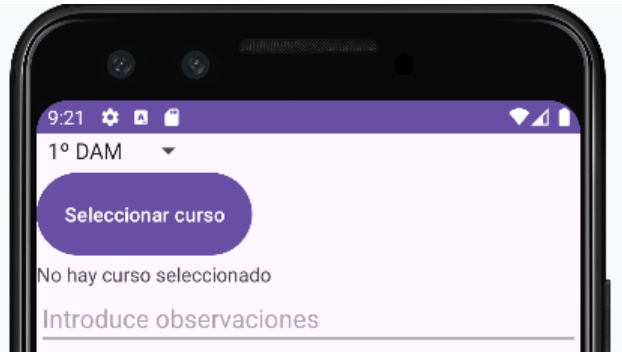
El atributo **android:hint** muestra un mensaje de texto dentro del **EditText** que sugiere al usuario lo que tiene que escribir. Dicho mensaje se quita cuando el usuario escribe algo en el **EditText**

```
1.  <EditText
2.      android:id="@+id/observaciones"
3.      android:hint="@string/observaciones"
4.      android:inputType="textMultiLine"
5.      android:layout_width="match_parent"
6.      android:layout_height="wrap_content"/>
```

- Añade por último un nuevo **Button** con estas características:
  - Identificador: **btnEnviar**
  - Texto: El referenciado en el String **enviar**
  - Anchura y altura: la de su contenido

```
1.  <Button
2.      android:id="@+id/btnEnviar"
3.      android:text="@string/enviar"
4.      android:layout_width="wrap_content"
5.      android:layout_height="wrap_content"/>
```

- Ejecuta la app y comprueba que aparecen los elementos que se han añadido al **LinearLayout** uno tras otro, pero ahora mismo tienen feo aspecto porque está todo muy pegado al borde y no hay márgenes que los separen entre ellos. Además, el botón se puede pulsar pero no hace nada



3 – Mejora básica de la apariencia: margin, padding, gravity y weight

Podemos mejorar la apariencia de la interfaz añadiendo a los elementos los siguientes atributos:

- **android:layout_margin:** Permite definir la separación del elemento con respecto a los elementos que lo rodean. De forma similar al margen de CSS, es posible indicar margen superior, inferior, izquierdo y derecho.
- **android:padding:** Permite definir la separación del borde del elemento con respecto a su contenido interior. De forma similar al padding de CSS, es posible indicar padding superior, inferior, izquierdo y derecho.
- **android:layout_gravity:** Nos indica hacia donde queremos centrar el elemento, dentro de la anchura o altura del layout que lo contiene.
- **android:gravity:** Nos indica hacia donde queremos centrar el contenido que hay dentro de un elemento, dentro de la anchura o altura de dicho elemento.
- **android:text_alignment:** Se utiliza en los elementos que contienen texto, y sirve para indicar la alineación del mismo.
- **android:weight:** Los elementos que llevan este atributo se estiran hasta repartirse de forma proporcional el espacio libre de su contenedor.

- Añade al **Spinner** y al **Button btnSeleccionarCurso** el atributo **android:layout_gravity** para centrarlo horizontalmente dentro del **LinearLayout** que lo contiene

```
<Spinner
    android:id="@+id/spnCurso"
    android:layout_gravity="center"
    android:entries="@array/cursos"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
<Button
    android:id="@+id/btnSeleccionarCurso"
    android:layout_gravity="center"
    android:padding="20dp"
    android:text="@string/seleccionarCurso"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

- Añade al **Button btnEnviar** el atributo **android:layout_gravity** para centrarlo a la derecha (valor **end**) dentro del layout que lo contiene

```
<Button
    android:id="@+id/btnEnviar"
    android:text="@string/enviar"
    android:layout_gravity="end"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

- Como el **TextView** tiene el mismo ancho que su padre (el **LinearLayout**) no es posible usar el **android:gravity** para centrarlo, pero en su lugar usaremos el atributo **android:textAlignment** de esta forma:

```
<TextView
    android:id="@+id/txtAsignaturas"
    android:textAlignment="center"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/sinCurso"/>
```

- A continuación vamos a separar los elementos metiéndoles un poco de margen, usando el atributo **android:layout_margin** así:

```
<Spinner
    android:id="@+id/spnCurso"
    android:layout_margin="16dp"
    android:layout_gravity="center"
    android:entries="@array/cursos"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
<Button
    android:id="@+id/btnSeleccionarCurso"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:text="@string/seleccionarCurso"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
<TextView
    android:id="@+id/txtAsignaturas"
    android:textAlignment="center"
    android:layout_margin="16dp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/sinCurso"/>
```

La unidad **dp** que acompaña al valor del margen indica que estamos usando **píxeles independientes de la densidad de la pantalla**. Hay móviles que tienen más resolución que otros y eso hace que su tamaño de píxel sea más pequeño. Para que la apariencia sea la misma en todos los móviles, se usa la unidad **dp** (*density independent pixel*). Hay otra unidad llamada **sp** (*scalable independent pixel)* que es lo mismo, pero para el texto. También se pueden usar píxeles brutos usando **px**

- Por último, añadimos un poco de padding para separar el texto del borde del botón, usando el atributo **android:padding**

```
<Button
    android:id="@+id/btnSeleccionarCurso"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    android:padding="20dp"
    android:text="@string/seleccionarCurso"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

- Vamos a añadir ahora el atributo **android:weight** para hacer que el **EditText** se estire hasta ocupar todo el espacio vacío que queda en su layout. *Observa que tras escribir dicho atributo, en el diseñador se ve cómo el **EditText** se ha estirado hasta ocupar toda la pantalla y el texto del **android:hint** aparece centrado en el **EditText**.*

```
<EditText
    android:id="@+id/observaciones"
    android:hint="@string/observaciones"
    android:inputType="textMultiLine"
    android:layout_weight="1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

Todos los elementos de la interfaz que llevan el atributo **android:layout_weight** se "estiran" hasta repartirse todo el espacio libre que queda en su contenedor. El valor de dicho atributo es un número, que indica la proporción de espacio que debe ocupar cada elemento. Por ejemplo, un elemento con valor **2** ocupará el doble de espacio libre que un elemento de valor **1**.

- El texto para introducir observaciones está centrado en el **EditText** y queda muy feo. Vamos a subirlo arriba poniendo al **EditText** el atributo **android:gravity**

```
<EditText
    android:id="@+id/observaciones"
    android:hint="@string/observaciones"
    android:inputType="textMultiLine"
    android:gravity="top"
    android:layout_weight="1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

El atributo **android:gravity** indica el lugar al que debe desplazarse el contenido de un elemento. Se admiten valores como:

- **top,bottom,start,end** → Alinean hacia arriba, abajo, izquierda o derecha[2]
- **center_vertical, center_horizontal, center** → Centra en el eje vertical, horizontal o en ambos
- **fill_vertical, fill_horizontal, fill** → Hace que el contenido no solo se alinee, sino que se expanda verticalmente, horizontalmente o en ambos.

Es posible poner varios de los valores anteriores separándolos con el signo **|**

- Ejecuta la app y comprueba que las separaciones y alineaciones se aplican correctamente

[2] En los dispositivos cuyo idioma se lee de derecha a izquierda, **start** sería la derecha y **end** la izquierda

4 – Acceder a los objetos de la interfaz en el código fuente de la app

Ahora mismo el botón de nuestra app no hace nada, y eso es normal porque en el **activity_main.xml** hemos definido la apariencia de la app, y no su comportamiento.

El comportamiento de la app se define en el archivo **MainActivity.kt** y se programa en lenguaje Kotlin.

Lo primero que tenemos que hacer en dicho archivo, es recuperar en **variables** los objetos que forman la interfaz[3], y para eso usaremos los **identificadores** que les hemos puesto y el método **findContentById**

- Abre el archivo **MainActivity.kt**
- Crea **variables de instancia (o propiedades)** para todos los elementos de la interfaz que queramos manejar en el código fuente. *En nuestro caso, todos los elementos que hemos añadido.*

```
1.  class MainActivity : AppCompatActivity() {
2.      private lateinit var spnCursos: Spinner
3.      private lateinit var btnSeleccionarCursor: Button
4.      private lateinit var txtAsignaturas:TextView
5.      private lateinit var txtObservaciones:EditView
6.      private lateinit var btnEnviar:Button
7.      override fun onCreate(savedInstanceState: Bundle?) {
8.          super.onCreate(savedInstanceState)
9.          setContentView(R.layout.activity_main)
10.     }
11. }
```

En Kotlin las variables de instancia que no son inicializadas en el constructor deben llevar la palabra **lateinit**. En nuestro caso, los elementos de la interfaz llevan **lateinit** porque son inicializados dentro del método **onCreate**, como veremos a continuación

- Inicializa dichas variables de instancia dentro del método **onCreate**, llamando al método **findViewById**, que recibe el identificador que hemos puesto al elemento en el **activity_main.xml**, pero añadiéndole como prefijo **R.id.**

```
1.  class MainActivity : AppCompatActivity() {
2.      private lateinit var spnCursos: Spinner
3.      private lateinit var btnSeleccionarCurso: Button
4.      private lateinit var txtAsignaturas:TextView
5.      private lateinit var txtObservaciones:EditText
6.      override fun onCreate(savedInstanceState: Bundle?) {
7.          super.onCreate(savedInstanceState)
8.          setContentView(R.layout.activity_main)
9.          spnCursos=findViewById(R.id.spnCurso)
10.         btnSeleccionarCurso=findViewById(R.id.btnSeleccionarCurso)
11.         txtAsignaturas=findViewById(R.id.txtAsignaturas)
12.         txtObservaciones=findViewById(R.id.txtObservaciones)
13.         btnEnviar=findViewById(R.id.btnEnviar)
14.     }
15. }
```

[3] El método **setContentView** no solo sirve para indicar qué diseño usará MainActivity, sino que además se encarga de crear objetos (de Java/Kotlin) a partir de las definiciones escritas en el xml del layout (dicho proceso se llama **layout inflation**)

Cuando se diseña **activity_main.xml** y ponemos identificadores, Android Studio crea una clase llamada **R** en la que coloca como constantes dichos identificadores. De esa forma, podemos escribir **R.id.txtAsignaturas** para referirnos al identificador **txtAsignaturas** que hemos puesto al **TextView** en el **activity_main.xml**

- Es buena costumbre que los métodos sean pequeños y que hagan tareas concretas. Para simplificar el método **onCreate**, vamos a crear un método auxiliar llamado **inicializarControles**, de forma que sea allí donde se recuperen los elementos de la interfaz. Así, **onCreate** llamará a **inicializarControles**, y allí se inicializarán las variables de instancia.

```
1.  class MainActivity : AppCompatActivity() {
2.      private lateinit var spnCursos: Spinner
3.      private lateinit var btnSeleccionarCurso: Button
4.      private lateinit var txtAsignaturas:TextView
5.      override fun onCreate(savedInstanceState: Bundle?) {
6.          super.onCreate(savedInstanceState)
7.          setContentView(R.layout.activity_main)
8.          this.inicializarControles()
9.      }
10.     private fun inicializarControles(){
11.         spnCursos=findViewById(R.id.spnCurso)
12.         btnSeleccionarCurso=findViewById(R.id.btnSeleccionarCurso)
13.         txtAsignaturas=findViewById(R.id.txtAsignaturas)
14.         txtObservaciones=findViewById(R.id.txtObservaciones)
15.         btnEnviar=findViewById(R.id.btnEnviar)
16.     }
17. }
```

5 – Responder a pulsaciones del botón

Ahora mismo el botón de nuestra app no hace nada. Si queremos que responda, debemos añadir en el archivo **MainActivity** código fuente con las acciones que queremos que se ejecuten al pulsar el botón.

Este código fuente se pasa como parámetro[4] al método **setOnClickListener** del botón.

- Abre el archivo **MainActivity** y al final del método **onCreate** añade la siguiente llamada al método **setOnClickListener** del objeto **btnSeleccionarCurso**

```
1.  override fun onCreate(savedInstanceState: Bundle?) {
2.      super.onCreate(savedInstanceState)
3.      setContentView(R.layout.activity_main)
4.      this.inicializarControles()
5.      btnSeleccionarCurso.setOnClickListener{
6.          // aquí se escribe el código que se ejecuta cada vez que se pulsa el botón
7.      }
8.  }
```

[4] En Kotlin (y otros lenguajes como Java o JavaScript) hay métodos que pueden recibir como parámetro un bloque de código fuente (también llamado **expresión lambda**)

Observa que el método **setOnClickListener** usa llaves **{ }** y no paréntesis **( )** como estamos acostumbrados. Eso se debe a que **setOnClickListener** recibe como parámetro un bloque de código fuente, que se pone en marcha cada vez que el usuario pulsa el botón

- Dentro del código fuente que recibe **setOnClickListener**, recuperamos el elemento seleccionado en **spnCursos** y lo pasamos a **String** con su método **toString.** Guardaremos dicho String en una variable[5], así:

```
1.  btnSeleccionarCurso.setOnClickListener{
2.      val curso: String = spnCursos.selectedItem.toString()
3.  }
```

- A continuación, vamos a obtener la lista de asignaturas de ese curso usando el bloque **when** del lenguaje Kotlin

```
1.  override fun onCreate(savedInstanceState: Bundle?) {
2.      super.onCreate(savedInstanceState)
3.      setContentView(R.layout.activity_main)
4.      this.inicializarControles()
5.      btnSeleccionarCurso.setOnClickListener{
6.          val curso: String = spnCursos.selectedItem.toString()
7.          val asignaturas: List<String> = when(curso){
8.              "1º DAM" -> listOf("Programación","Bases de datos","Sistemas","Entornos","Fol")
9.              "2º DAM" -> listOf("Móviles","Interfaces","Acceso a datos","PSP","Empresa")
10.             else -> throw Exception("Curso no admitido")
11.         }
12.     }
13. }
```

- Al igual que hemos hecho antes, vamos a facilitar la lectura del código haciéndonos un método llamado **getListaAsignaturas** que reciba el nombre del curso y nos devuelva la lista de asignaturas. De esa forma, el código fuente que pasamos a **setOnClickListener** quedará más sencillo porque se limitará a llamar al método **getListaAsignaturas**

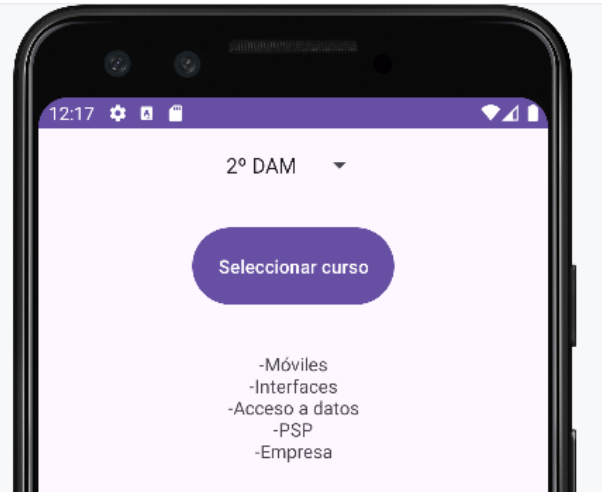
```
1.  override fun onCreate(savedInstanceState: Bundle?) {
2.      super.onCreate(savedInstanceState)
3.      setContentView(R.layout.activity_main)
4.      this.inicializarControles()
5.      btnSeleccionarCurso.setOnClickListener{
6.          val curso: String = spnCursos.selectedItem.toString()
7.          val asignaturas: List<String> = this.getListaAsignaturas(curso)
8.      }
9.  }
10. private fun getListaAsignaturas(curso:String):List<String>{
11.     return when(curso){
12.         "1º DAM" -> listOf("Programación","Bases de datos","Sistemas","Entornos","Fol")
13.         "2º DAM" -> listOf("Móviles","Interfaces","Acceso a datos","PSP","Empresa")
14.         else -> throw Exception("Curso no admitido")
15.     }
16. }
```

[5] En Kotlin normalmente no es necesario indicar el tipo de dato cuando hacemos una variable. Sin embargo, en estos apuntes lo usaremos siempre para que haya mayor claridad en las explicaciones.

- Por último, vamos a colocar en **txtAsignaturas** la lista de asignaturas, separando con guiones cada una de la siguiente. Para ello, formamos un String vacio y después hacemos un recorrido de la lista guardada en la variable **asignaturas**, añadiendo cada asignatura recorrida a dicho String y separándola de la siguiente con el carácter de nueva línea **\n.** Una vez que tenemos hecho eso, asignamos **texto** a la propiedad **text** del objeto **txtAsignaturas**.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    this.inicializarControles()
    btnSeleccionarCurso.setOnClickListener{
        val curso: String = spnCursos.selectedItem.toString()
        val asignaturas: List<String> = this.getListaAsignaturas(curso)
        // creamos una variable para guardar el texto
        var texto:String = ""
        for(a in asignaturas){  // recorremos la lista de asignaturas
            texto += "-"+a+"\n" // pegamos la asignatura al texto
        }
        txtAsignaturas.text=texto // mostramos el texto en txtAsignaturas
    }
}
```

- Ejecuta la app y comprueba que todo funciona correctamente



- Al igual que hemos hecho en ocasiones anteriores, el código que recibe **setOnClickListener** está quedando muy grande y complejo. Lo simplificamos haciendo un método privado llamado **getAsignaturas** que recibe la lista de asignaturas y nos devuelve el String donde aparece cada una separada de la siguiente.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    this.inicializarControles()
    btnSeleccionarCurso.setOnClickListener{
        val curso: String = spnCursos.selectedItem.toString()
        val asignaturas: List<String> = this.getListaAsignaturas(curso)
        val texto:String = this.getAsignaturas(asignaturas)
        txtAsignaturas.text=texto
    }
}
```

```
12. private fun getAsignaturas(asignaturas:List<String>):String{
13.     // creamos una variable para guardar el texto
14.     var texto:String = ""
15.     for(a in asignaturas){  // recorremos la lista de asignaturas
16.         texto += "-"+a+"\n" // pegamos la asignatura al texto
17.     }
18.     return texto
19. }
```

Como última mejora, vamos a usar **programación funcional** para dar una versión alternativa de la programación del método **getAsignaturas**.

Tal y como ha sido programado, el método **getAsignaturas** hace uso de bucles para obtener el **String** que hay que mostrar en el cuadro de texto. Esa forma de trabajar se denomina **programación imperativa** y es propia de los lenguajes clásicos.

Kotlin favorece otro enfoque diferente llamado **programación funcional**, que se basa en realizar transformaciones sobre los datos, evitando así el uso de bucles.

- Borra el código fuente que hay en el interior del método **getAsignaturas**
- Escribe el siguiente código, que toma cada asignatura de la lista, la transforma **(map)** en un String en el que aparece un guión, el nombre de la asignatura, y el salto de línea, y por último, pega todos esos String.

```
1.  private fun getAsignaturas(asignaturas:List<String>):String{
2.      return asignaturas  // toma todas las asignaturas de la lista
3.          .map{ a -> "- $a \n"}  // transforma cada una en "- nombre \n"
4.          .joinToString()  // une todas las asignaturas transformadas en un solo String
5.  }
```

- Y para terminar, hacemos uso de una característica de Kotlin llamada **single line method** que nos dice que cuando un método solo está formado por un return (como ocurre en nuestro **getAsignatura**), podemos quitar la palabra **return** si a cambio ponemos en la declaración del método un signo **=** ,que hace de return:

```
1.  private fun getAsignatura(asignaturas:List<String>):String =
2.          asignaturas
3.              .map{ a -> "- $a \n"}
4.              .joinToString()
```

6 – Toast

Un **Toast** es un mensaje emergente que se muestra dentro de una burbuja. Se muestra para mostrar información breve que desaparece pasado un tiempo, y no es posible interactuar con su contenido.

Los **Toast** se crean con el método estático **Toast.makeText** y se hacen visibles con su método **show**, tal como se verá a continuación

- Dentro del método **onCreate**, vamos a llamar al método **setOnClickListener** de **btnEnviar** para indicar el bloque de código que se ejecutará cuando se pulse dicho botón

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    this.inicializarControles()
    btnSeleccionarCurso.setOnClickListener{
        val curso: String = spnCursos.selectedItem.toString()
        val asignaturas: List<String> = getAsignaturas(curso)
        val texto:String = getAsignatura(asignaturas)
        txtAsignaturas.text=texto // mostramos el texto en txtAsignaturas
    }
    btnEnviar.setOnClickListener{
        // código que se ejecuta al pulsar btnEnviar
    }
}
```

- Dentro del bloque de código que se ejecutará cuando se pulse **btnEnviar**, guardaremos en una variable el contenido del cuadro de texto y lo pasaremos a **String**

```
btnEnviar.setOnClickListener{
        val observaciones:String = txtObservaciones.text.toString()
    }
```

- Dentro del bloque de código que se ejecutará cuando se pulse **btnEnviar**, llamamos al método **Toast.makeText**, encadenando la llamada a dicho método con otra llamada al método **show** para hacer visible el **Toast**. Comenzaremos haciendo que el **Toast** muestre el contenido del cuadro de texto **txtObservaciones**.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    this.inicializarControles()
    btnSeleccionarCurso.setOnClickListener{
        val curso: String = spnCursos.selectedItem.toString()
        val asignaturas: List<String> = getAsignaturas(curso)
        val texto:String = getAsignatura(asignaturas)
        txtAsignaturas.text=texto // mostramos el texto en txtAsignaturas
    }
    btnEnviar.setOnClickListener{
        val observaciones: String = txtObservaciones.text.toString()
        Toast.makeText(this,"Las observaciones son $observaciones",Toast.LENGTH_SHORT).show()
    }
}
```

El método **Toast.makeText** recibe estos parámetros:

- **Contexto:** Es el lugar sobre el que se muestra el **Toast.** Se pasa **this,** para indicar que el **Toast** se muestre encima de la **MainActivity** que estamos programando.
- **Mensaje:** Es un String con el mensaje que se va a mostrar en el **Toast**
- **Duración:** Es una constante **Toast.LENGTH_SHORT** o **Toast.LENGTH_LONG** que define la duración del **Toast** en la pantalla.

- Lo habitual es que el mensaje del **Toast** no sea un String escrito a mano, sino que esté en el archivo **/res/values/strings.xml**. Añade al **strings.xml** un nuevo mensaje llamado **envio** con el texto **"Las observaciones son %1$d"**, así:

```
<resources>
    <string name="app_name">My Application</string>
    <string name="seleccionarCurso">Seleccionar curso</string>
    <string name="sinCurso">No hay curso seleccionado</string>
    <string name="observaciones">Introduce observaciones</string>
    <string name="enviar">Enviar</string>
    <string name="envio">Las observaciones son %1$d</string>
    <string-array name="cursos">
        <item>1º DAM</item>
        <item>2º DAM</item>
    </string-array>
</resources>
```

El mensaje que hemos añadido posee una variable en su interior. Por ese motivo, en el lugar donde debe ir la variable, aparece **%1$s.** El **1** hace referencia al número de la variable (puede ser que el mensaje contenga muchas variables) y **$s** indica que la variable que va ahí es un String. También se puede usar **$d** si la variable es un número entero o **$f** si es un número con decimales.

- Cambia la llamada a **Toast.makeText** para que el mensaje se obtenga con el método **getString**, de esta forma:

```
btnEnviar.setOnClickListener{
    val observaciones = txtAsignaturas.text.toString()
    Toast.makeText(this,getString(R.string.envio,observaciones),Toast.LENGTH_SHORT).show()
}
```

El método **getString** recibe como primer parámetro el identificador del mensaje que se va a mostrar y luego, tantas variables como se deseen, que serán colocadas en los lugares del mensaje donde aparecen **%1$s, %2$s,** etc.

7 – Concepto de FrameLayout

Hemos visto que el **LinearLayout** dispone los elementos de la interfaz uno a continuación de otro, pero hay muchos más layouts que estudiaremos durante el curso.

Uno de ellos es el **FrameLayout**, que dispone los elementos uno encima de otro. Esto puede ser interesante en casos como estos:

- Podemos hacer "capas" que se sitúan unas encima de otras. Por ejemplo, podemos tener una imagen y justo encima, un LinearLayout con varios elementos. Este será el uso que haremos en este proyecto.
- Podemos hacer una interfaz dinámica en la que las pantallas se activan o desactivan según lo que suceda. O sea, tenemos varias "capas" de elementos y solo una es visible. Este enfoque, aunque interesante, es superado por los **Fragments**, que estudiaremos más adelante en el curso.

- Abre el archivo **activity_main.xml** y coloca un **FrameLayout** como elemento principal de la interfaz, de forma que englobe al **LinearLayout** que ya tenías de antes, así:

```
<FrameLayout
    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_height="match_parent"
    android:layout_width="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <!-- aquí viene todo el contenido de antes: Spinner, Button, EditText, TextView -->
    </LinearLayout>
</FrameLayout>
```

Como el **FrameLayout** pasa a ser el elemento padre de la interfaz, deberán definirse en él los espacios de nombres **android, app** y **tools**, retirándolos del **LinearLayout**, que pasa ahora a ser un elemento secundario en la interfaz.

- Si ejecutas la app, no verás ningún cambio apreciable. Pero ahora el **LinearLayout** está dentro del **FrameLayout**. Esto significa que cualquier otro elemento que pongamos en el **FrameLayout** se colocará encima o debajo del **LinearLayout**, según lo escribamos antes o después en el xml.

8 – ImageView

Para mostrar una imagen se utiliza el elemento **ImageView**. Sus atributos más importantes (además de la anchura y altura) son:

- **android:src:** Es la ruta de la imagen que se mostrará en el **ImageView**. Debe ser una imagen (preferiblemente en formato **webp**[6]) que se encontrará en la carpeta **/res/drawable**
- **android:scaleType:** Es el tipo de escalado que se hará sobre la imagen. Por ejemplo, para que se escale en ambas dimensiones se usará **fitXY**
- **android:contentDescription:** Es una descripción que se pone en la imagen y que será leía en voz alta en los dispositivos configurados para personas con discapacidad visual.

- Descarga una imagen de fondo para la aplicación
- En el explorador del sistema operativo, busca dicha imagen, pulsa sobre ella el botón derecho del ratón y dale a **copiar**
- Vete a Android Studio y sobre la carpeta **/res/drawable** pulsa el botón derecho del ratón y dale a **pegar**. Si lo haces bien, la imagen se verá en dicha carpeta.

[6] Android fomenta el uso de este formato debido a su gran calidad y pequeño tamaño de la imagen

- Si quieres convertir la imagen al formato **webp**, pulsa el botón derecho del ratón y selecciona la opción **Convert to webp**. *Se abrirá una ventana donde puedes configurar la calidad de la conversión.*

- Dentro del **FrameLayout**, y antes del **LinearLayout**, coloca el siguiente elemento **ImageView.** *Se supone que la imagen de fondo se llama **fondo.webp**)*

```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/fondo"
    android:contentDescription="Imagen de fondo azul"
    android:scaleType="fitXY"/>
```

- Si ejecutas la app, verás que todo se muestra correctamente, pero se ha perdido la flechita del **Spinner** y además, no se ve claramente cuál es la zona de escritura del **EditText**
- Para mostrar la flechita del **Spinner**, configura el canal alpha (o grado de transparencia) del **ImageView** para que sea de **0.6**, así:

```
<ImageView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/fondo"
    android:contentDescription="Imagen de fondo azul"
    android:alpha="0.6"
    android:scaleType="fitXY"/>
```

- Para destacar la zona de escritura del **EditText**, le pondremos el atributo **android:background** para indicar que su color de fondo sea blanco. *Observa que dicho color ya se encuentra predefinido en **/res/values/colors.xml***

```
<EditText
    android:id="@+id/observaciones"
    android:hint="@string/observaciones"
    android:gravity="top"
    android:layout_weight="1"
    android:layout_margin="16dp"
    android:inputType="textMultiLine"
    android:background="@color/white"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

9 – Introducción al View Binding

En el proyecto que hemos hecho hemos usado muchísimas veces **findElementById** para guardar en variables los elementos de la interfaz, pero esto es muy pesado y además es ineficiente y propenso a errores.

El **View Binding** es una herramienta que nos proporciona un objeto que contiene en su interior a los elementos de la interfaz, ya construidos.

- Para habilitar el **view binding**, nos vamos al archivo **build.gradle.kts** (versión **Module**) y añadimos el siguiente bloque: (*no olvidar sincronizar el proyecto de Gradle para aplicar los cambios de configuración*)

```
android {
    // resto omitido
    buildFeatures {
        viewBinding = true
    }
}
```

Al hacer esto, Android Studio nos genera para **activity_main.xml** una clase llamada **ActivityMainBinding**, cuyas variables de instancia (o atributos) son los elementos de la interfaz ya construidos.

- A continuación, abrimos **MainActivity** y eliminamos todas las variables de instancia en las que guardamos los elementos de la interfaz.

```
class MainActivity : AppCompatActivity() {
    private lateinit var spnCursos: Spinner
    private lateinit var btnSeleccionarCurso: Button
    private lateinit var txtAsignaturas:TextView
    private lateinit var btnEnviar: Button
    // resto omitido
}
```

- En su lugar, añadimos una variable de instancia que será el objeto **controles**, cuyo tipo es **ActivityMainBinding**, y que contendrá todos los elementos de la interfaz

```
class MainActivity : AppCompatActivity() {
    private lateinit var controles: ActivityMainBinding
    // resto omitido
}
```

- A continuación, tenemos que cambiar la programación del método **onCreate** para inicializar el objeto **controles**

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        controles = ActivityMainBinding.inflate(layoutInflater)
        setContentView(controles.root)
        setContentView(R.layout.activity_main)
        // resto omitido
}
```

El método **ActivityMainBinding.inflate** usa el **layoutInflater** de la **Activity** para crear todos los elementos de la interfaz definida en **activity_main.xml** y nos devuelve un objeto **ActivityMainBinding** que los contiene a todos ellos.

Es necesario cambiar la llamada a **setContentView** para que no cargue el archivo **activity_main.xml**, sino que use como vista al objeto **controles.root**, que es el **LinearLayout** padre que hay en **activity_main.xml**

- Ahora podemos borrar el método **inicializarControles,** ya que no es necesario rellenar variables para guardar los objetos de la interfaz

```
1.  private fun inicializarControles(){
2.          this.spnCursos=findViewById(R.id.spnCurso)
3.          this.btnSeleccionarCurso=findViewById(R.id.btnSeleccionarCurso)
4.          this.txtAsignaturas=findViewById(R.id.txtAsignaturas)
5.          this.btnEnviar=findViewById(R.id.btnEnviar)
6.  }
```

- Por último, nos vamos a los lugares donde utilizamos los objetos de la interfaz, y lo único que necesitamos es usar el objeto **controles** para recuperarlos.

```
1.  override fun onCreate(savedInstanceState: Bundle?) {
2.      super.onCreate(savedInstanceState)
3.      controles = ActivityMainBinding.inflate(layoutInflater)
4.      setContentView(controles.root)
5.      controles.btnSeleccionarCurso.setOnClickListener{
6.          val curso: String = controles.spnCursos.selectedItem.toString()
7.          val asignaturas: List<String> = getListaAsignaturas(curso)
8.          val texto:String = getAsignaturas(asignaturas)
9.          controles.txtAsignaturas.text=texto
10.     }
11.     controles.btnEnviar.setOnClickListener{
12.        val observaciones = controles.txtAsignaturas.text.toString()
13.        Toast.makeText(this,getString(R.string.envio,observaciones),Toast.LENGTH_SHORT).show()
14.     }
15. }
```

El objeto **controles** posee una variable de instancia por cada elemento que tenga un id en el archivo **activity_main.xml**. Por ejemplo, el botón con id **btnEnviar** se encuentra disponible en el código fuente como **controles.btnEnviar**

- Ejecuta la app y comprueba que todo sigue funcionando igual que antes

El código fuente ahora es mejor porque:

- Es menos lioso y más corto, al tener menos líneas
- Es menos propenso a errores (al usar **findViewById** nos podemos equivocar al intentar guardar un objeto en una variable que no sea de su tipo, como por ejemplo un botón en una imagen)
- Es más eficiente (cada vez que se llama a **findViewById** el dispositivo necesita buscar el objeto entre todos los de la interfaz, lo cual es lento)

10 – Ejercicios

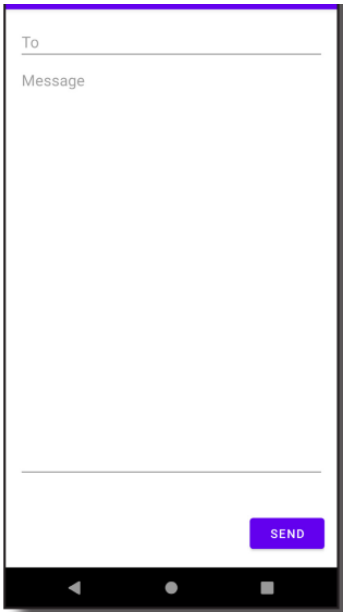
Ejercicio 1: Modifica la interfaz de la app **Informe Asignaturas** para que el texto aparezca alineado a la izquierda, y centrado en la pantalla, de esta forma:

Ejercicio 2: La app **Informe Asignaturas** tiene un pequeño problema. Al pulsar el botón, la lista de asignaturas le "come" espacio al cuadro de observaciones. Modifica la interfaz, de forma que la lista de asignaturas y el cuadro de observaciones se repartan proporcionalmente el espacio libre del layout, ocupando el cuadro de observaciones doble que la lista de asignaturas.

Ejercicio 3: Añade el inglés a la aplicación y comprueba que el idioma se cambia correctamente entre español e inglés cuando el móvil cambia de idioma.

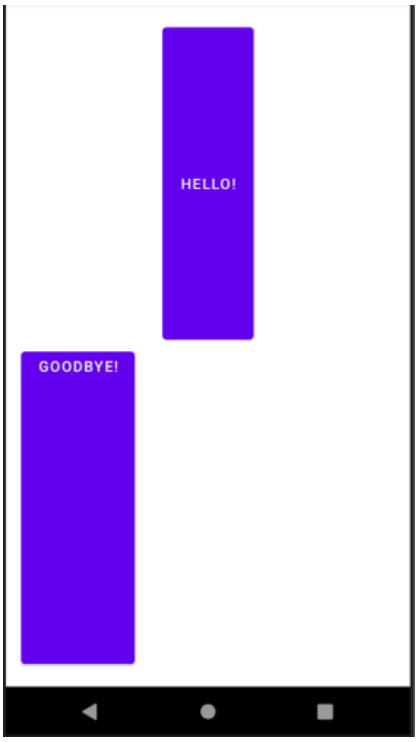
Ejercicio 4: Ejecuta la app **InformeAsignaturas**, selecciona un curso, dale al botón y escribe algo en los comentarios. A continuación, gira el móvil. ¿Qué sucede? ¿Por qué sucede esto?

Ejercicio 5: Crea un proyecto llamado **Ejercicio5** y diseña la siguiente interfaz usando View Binding. Al pulsar el botón, se mostrará en Logcat y en un Toast los valores de los dos cuadros de texto.

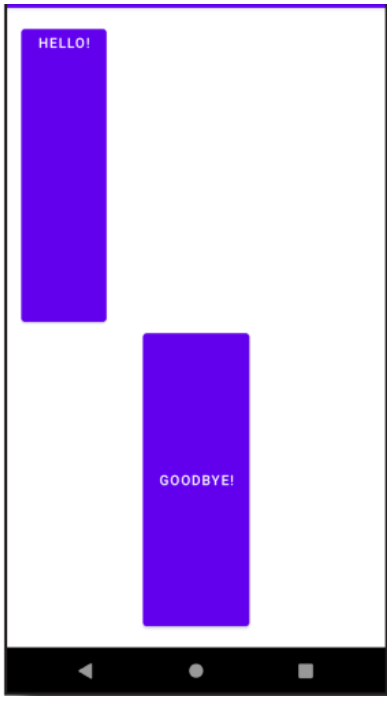


Ejercicio 6: Crea un proyecto llamado **Ejercicio6** y añade estos dos layouts, que están formados por dos botones, que al ser pulsados mostrarán un Toast que muestre el texto que hay dentro del correspondiente botón.

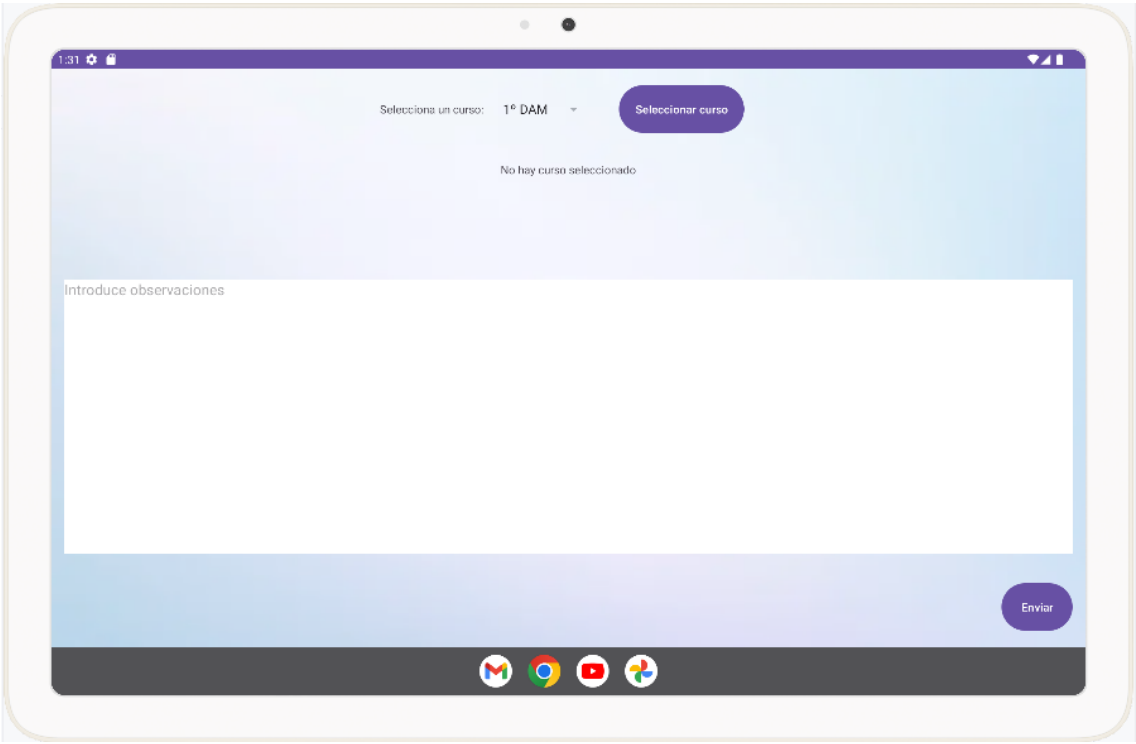
- **activity_main.xml**



- **activity_main.xml** para dispositivos con el idioma español



Ejercicio 7: Añade a la app **Informe Asignaturas** esta interfaz para tablets, usando View Binding



Ejercicio 8: Realiza una app llamada **Ejercicio 8** que muestre una imagen de fondo y un texto centrado en dicha imagen, como se ve en el siguiente ejemplo:

Pista: Usa un FrameLayout