
Kotlin

CONCEPTOS BÁSICOS DE SINTAXIS_

🧠 1. Variables en Kotlin

En Android se programa casi todo con **Kotlin**.
Hay dos formas de declarar variables:

kotlin Copiar código

```kotlin
var nombre = "Alvaro"    // mutable: puede cambiar
val edad = 20            // inmutable: no puede cambiar
```

- `var` → "variable normal", su valor se puede reasignar.
- `val` → "constante", su valor queda fijo después de asignarlo.

Ejemplo:

kotlin Copiar código

```kotlin
var contador = 0
contador = contador + 1 // ✅ permitido

val dni = "12345678A"
dni = "99999999B"       // ❌ error, val no se puede cambiar
```

Kotlin **infiera el tipo**, pero también puedes declararlo explícitamente:

kotlin Copiar código

```kotlin
val edad: Int = 21
val nombre: String = "Alvaro"
val precio: Double = 12.99
```

💡 3. If / Else (condiciones)

kotlin Copiar código

```kotlin
val edad = 18

if (edad >= 18) {
    println("Eres mayor de edad")
} else {
    println("Eres menor de edad")
}
```

También se puede usar **como expresión** (devuelve un valor):

kotlin Copiar código

```kotlin
val mensaje = if (edad >= 18) "Mayor" else "Menor"
println(mensaje)
```

🔁 4. Bucles

For: recorrer listas o rangos.

```kotlin
for (i in 1..5) {
    println(i)
}
```

While: repetir mientras se cumpla una condición.

```kotlin
var i = 0
while (i < 3) {
    println("Número $i")
    i++
}
```

📦 5. Arrays y Listas

Kotlin distingue entre **Array** (tamaño fijo) y **List** (más flexible).

Array:

```kotlin
val numeros = arrayOf(1, 2, 3, 4)
println(numeros[0])  // imprime 1
```

List (inmutable):

```kotlin
val frutas = listOf("Manzana", "Pera", "Plátano")
// frutas.add("Mango") ❌ no se puede modificar
```

MutableList (editable):

```kotlin
val frutas = mutableListOf("Manzana", "Pera")
frutas.add("Mango")   // ✅ se puede
println(frutas)
```

🧩 6. Funciones

Definen bloques de código reutilizables.

```kotlin
fun saludar(nombre: String) {
    println("Hola, $nombre!")
}

saludar("Alvaro")
```

Con retorno:

```kotlin
fun sumar(a: Int, b: Int): Int {
    return a + b
}

val resultado = sumar(3, 5)
println(resultado) // 8
```

🧠 8. Clases y objetos (mini repaso)

kotlin — Copiar código

```kotlin
class Persona(val nombre: String, var edad: Int) {
    fun saludar() {
        println("Hola, soy $nombre y tengo $edad años")
    }
}

val persona = Persona("Alvaro", 21)
persona.saludar()
```

CONCEPTOS BASICOS XML_

*Todos los xml que tengan que ver con las vistas van dentro de RES/LAYOUT/

📐 3. Propiedades clave

Todas las vistas tienen algunas propiedades básicas:

Propiedad	Qué hace	Ejemplo
`android:id`	Identificador único para usarlo en Kotlin	`@+id/btnEnviar`
`android:layout_width`	Ancho de la vista	`match_parent`, `wrap_content`, `200dp`
`android:layout_height`	Alto de la vista	igual que arriba
`android:layout_margin`	Margen externo	`16dp`
`android:padding`	Espacio interno	`8dp`
`android:text`	Texto mostrado	`"Aceptar"`
`android:background`	Fondo de color o imagen	`@color/azul` o `@drawable/bg_button`

📏 4. Unidades de medida

Android usa **unidades independientes de la pantalla**, para que la app se vea igual en móviles grandes o pequeños.

Unidad	Qué representa	Cuándo usarla
`dp`	density-independent pixel (escala adaptativa)	tamaños y márgenes
`sp`	scale-independent pixel	texto
`px`	píxeles reales (no recomend	casos específicos

🔍 Desglose por partes

Línea	Propiedad	Qué significa	Ejemplo o explicación
1	`<Button`	Tipo de vista	Define que es un botón (podría ser `TextView`, `ImageView`, etc.)
2	`android:id="@+id/btnCentrado"`	Identificador único	Le da un **nombre interno** al botón para usarlo en Kotlin (`findViewById(R.id.btnCentrado)`)
3	`android:text="Centro"`	Texto visible en el botón	Es lo que verá el usuario ("Centro")
4	`app:layout_constraintTop_toTopOf="parent"`	Fija la parte **superior** del botón al borde superior del contenedor (`parent`)	
5	`app:layout_constraintBottom_toBottomOf="parent"`	Fija la parte **inferior** del botón al borde inferior del contenedor	
6	`app:layout_constraintStart_toStartOf="parent"`	Fija el **inicio horizontal** (izquierda o derecha según idioma) al borde del contenedor	
7	`app:layout_constraintEnd_toEndOf="parent"`	Fija el **final horizontal** (lado opuesto) al borde del contenedor	
8	`/>`	Cierre de la vista	Marca el fin de la etiqueta XML (porque no tiene hijos dentro)

PASO A PASO.._

🧩 1️⃣ Qué es un layout raíz

El **layout raíz** es **la primera etiqueta** en tu archivo XML.
Es el "contenedor padre" de toda tu pantalla: dentro de él metes los botones, textos, imágenes, etc.

Ejemplo base:

xml Copiar código

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- Aquí van tus vistas hijas -->
</LinearLayout>
```

👉 Todo lo que esté dentro del `<LinearLayout>...</LinearLayout>` son **vistas hijas** (TextView, Button, ImageView, etc.).
El layout raíz define cómo se van a colocar y comportar esas vistas.

🧠 2️⃣ Pasos para crear un XML desde cero

1. **Crear el archivo:**
   - En Android Studio → `res/layout/` → clic derecho → `New → Layout Resource File`
   - Nómbralo (por ejemplo `activity_main.xml`).
   - Android te pide un **Root element (elemento raíz)** → aquí eliges el tipo de layout: `ConstraintLayout`, `LinearLayout`, etc.
2. **Definir el layout raíz:**
   - Este será el "contenedor principal" de todo lo que se ve en esa pantalla.
3. **Configurar su tamaño:**

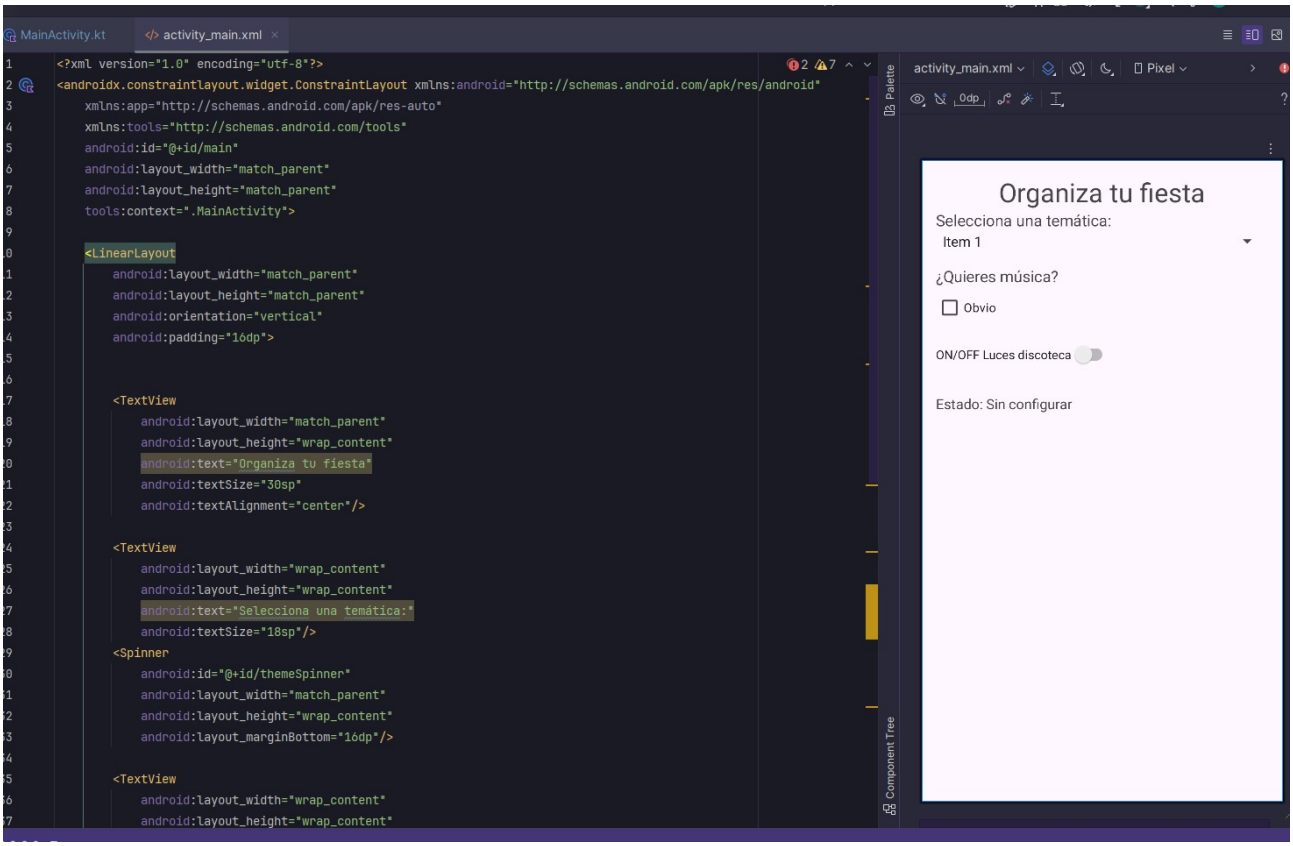
```xml
android:layout_width="match_parent"
android:layout_height="match_parent"
```

   Esto hace que ocupe toda la pantalla.
4. **Agregar tus vistas hijas:**
   Dentro metes cosas como botones, textos, imágenes...

```xml
<Button
    android:id="@+id/btnSaludo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Saludar" />
```

5. **Ajustar el diseño según el tipo de layout:**
   Cada layout organiza las vistas de manera distinta (por filas, columnas, posiciones, etc.).

90% SE USA ConstraintLayout, funcionamiento_



1. `ConstraintLayout` **(raíz)**

   Es el contenedor principal de tu pantalla. Android necesita *una raíz*, siempre.

   Pero —y aquí el detalle— tú **no estás usando sus constraints** directamente, sino que has puesto dentro un `LinearLayout` que se encarga del orden.

2. `LinearLayout` **dentro del ConstraintLayout**

   Está ocupando todo el espacio (`match_parent` en ancho y alto).

   Su orientación es **vertical**, así que todo lo que pongas dentro se apila uno debajo del otro:

   - Primero el título ("Organiza tu fiesta")
   - Luego "Selecciona una temática"
   - Luego el Spinner
   - Luego el CheckBox, etc.

🧱 Cómo se ordena el contenido

Por dentro, el `LinearLayout` hace esto:

xml Copiar código

```xml
<TextView
    android:text="Organiza tu fiesta"
    android:textSize="30sp"
    android:textAlignment="center" />

<TextView
    android:text="Selecciona una temática:"
    android:textSize="18sp" />

<Spinner
    android:id="@+id/themeSpinner"
    ... />

<TextView
    android:text="¿Quieres música?" />

<CheckBox
    android:text="Obvio" />

<Switch
    android:text="ON/OFF Luces discoteca" />

<TextView
    android:text="Estado: Sin configurar" />
```

👉 Como `orientation="vertical"`, cada vista aparece **una debajo de otra**, con el padding del contenedor (`16dp`) para no pegarse a los bordes.

🎯 Si quisieras hacerlo con `ConstraintLayout` (sin LinearLayout)

Tendrías que *atar* cada vista al padre o a la anterior:

xml Copiar código

```xml
<TextView
    android:id="@+id/tvTitulo"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Organiza tu fiesta"
    android:textSize="30sp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:gravity="center" />

<TextView
    android:id="@+id/tvTematica"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Selecciona una temática:"
    app:layout_constraintTop_toBottomOf="@id/tvTitulo"
    app:layout_constraintStart_toStartOf="parent" />
```

...y seguir así para cada elemento (`Spinner` , `CheckBox` , `Switch` , etc.).

Parte	Qué hace	Recomendación
`ConstraintLayout`	Layout raíz, controla posición con constraints	Úsalo cuando necesites precisión o pantallas complejas
`LinearLayout`	Alinea todo en fila o columna	Perfecto para formularios o listas simples
`Padding`	Espacio interno dentro del contenedor	Siempre usa al menos `16dp`
`match_parent` / `wrap_content`	Ocupa todo / se ajusta al contenido	Usa según necesites

Como enlazar xml. A activity y Strings.xml

🧩 1️⃣ Cómo se enlaza el XML a la Activity

Cada pantalla (Activity) de tu app tiene **dos archivos principales**:

- Un archivo **Kotlin** (`MainActivity.kt`) → contiene la lógica.
- Un archivo **XML** (`activity_main.xml`) → contiene la interfaz visual.

Android los **conecta automáticamente** mediante esta línea dentro de tu `MainActivity` :

```kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    setContentView(R.layout.activity_main)
```

Dentro del metodo Oncreate

🗂️ 3️⃣ ¿Qué es `strings.xml`?

En Android, todos los **textos visibles** (títulos, etiquetas, mensajes, etc.) se deben guardar en un archivo especial:
`res/values/strings.xml`.

Este archivo guarda todos los textos como **recursos reutilizables**.

Ejemplo:

xml — Copiar código

```xml
<resources>
    <string name="app_name">Organiza tu fiesta</string>
    <string name="label_tematica">Selecciona una temática:</string>
    <string name="checkbox_text">Obvio</string>
</resources>
```

💬 ¿Por qué se hace así?

1. Para poder **traducir** fácilmente tu app a otros idiomas (solo cambias `strings.xml`).
2. Para **mantener el código limpio** (sin textos "quemados" en XML o Kotlin).
3. Para **reutilizar textos** en distintos lugares.

Amiguet dijo que todas las String tienen que estar almacenadas en el Strings.xml

El archivo `strings.xml` *no guarda IDs*, sino **textos visibles** (los que el usuario ve en pantalla).
Las **IDs** (`@+id/...`) sirven para identificar vistas, pero los **strings** (`@string/...`) sirven para mostrar palabras, frases, títulos, botones, etc.

🎯 En resumen corto:

- `android:id` → nombre interno para el código (Kotlin).
- `@string/...` → texto visible que se guarda en `strings.xml`.

💬 Ejemplo visual completo

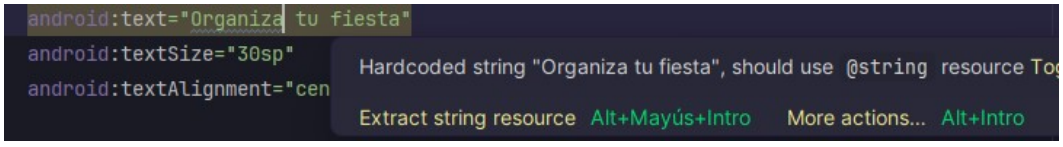
Archivo `activity_main.xml`

xml — Copiar código

```xml
<TextView
    android:id="@+id/tvTitulo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/titulo_app" />
```

Como hago que las Strings se guarden en el xml?

1 En android: text= "la fiesta"

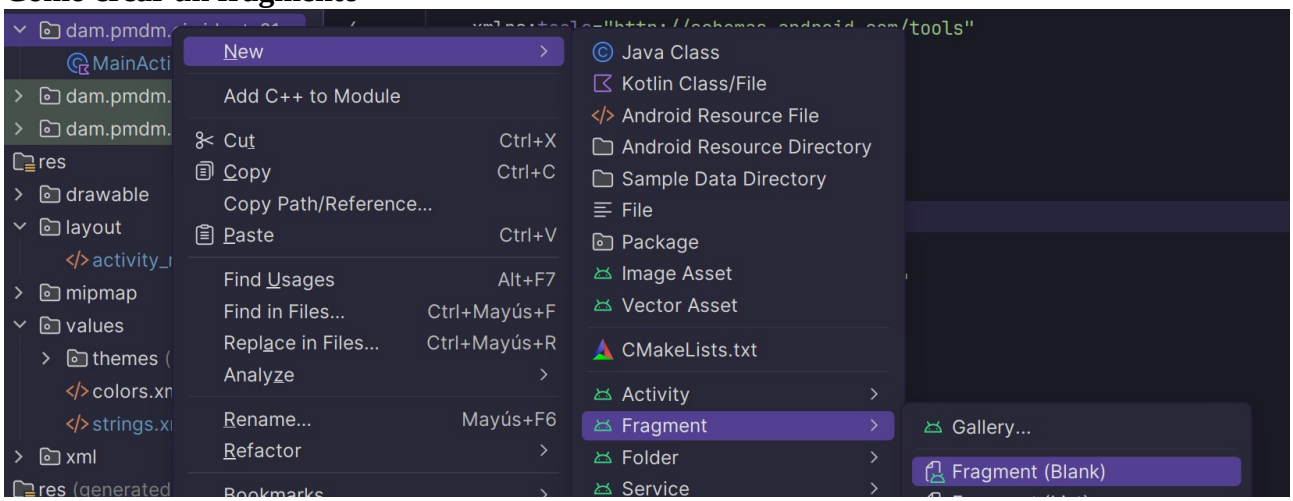


le das a Extrac y a tomar por culo directo al xml de Strings en res/values/strings.xml

NOS VAMOS A LA ACTIVITY

Como crear una activity

Como crear un fragmento



🧩 ESQUEMA DE PASOS — DE XML A LÓGICA

java Copiar código

```
1 CREAR EL LAYOUT XML
  ↓
2 CREAR LA ACTIVITY KOTLIN
  ↓
3 CARGAR EL XML CON setContentView()
  ↓
4 ENLAZAR LAS VISTAS (findViewById)
  ↓
5 CREAR ADAPTADORES (si el widget lo necesita)
  ↓
6 ASIGNAR LISTENERS / MÉTODOS (acciones del usuario)
  ↓
7 EJECUTAR LA LÓGICA (mostrar, cambiar, enviar, etc.)
```

🔧 PASO A PASO: QUÉ HACE CADA BLOQUE

Paso	Qué hace	Cuándo se ejecuta
1️⃣ Declaras las variables	Reservas espacio para las vistas	Antes del `onCreate()`
2️⃣ `setContentView()`	Carga el XML visual	Al iniciar la Activity
3️⃣ `findViewById()`	Conecta cada vista con su ID del XML	Justo después de cargar el XML
4️⃣ Adaptadores	Cargan datos (si el widget lo necesita)	Después del bindeo
5️⃣ Listeners	Reaccionan a acciones del usuario	Al final del `onCreate()`
6️⃣ Funciones auxiliares	Mantienen el código ordenado	Fuera del `onCreate()`

Bindeo:

```
// 3️⃣ Enlazar vistas con el XML
textView = findViewById(R.id.textView)
editText = findViewById(R.id.editText)
btnEnviar = findViewById(R.id.btnEnviar)
checkNoticias = findViewById(R.id.checkNoticias)
switchModo = findViewById(R.id.switchModo)
themeSpinner = findViewById(R.id.themeSpinner)
recycler = findViewById(R.id.recycler)
imageView = findViewById(R.id.imageView)
```

WIDGETS

TextView:

⚙️ 4️⃣ Usar la variable (mostrar o cambiar texto)

Por ejemplo:

kotlin — Copiar código

```
tvMensaje.text = "Hola desde Kotlin 👋"
```

Esa línea **cambia el texto visible en la pantalla**.

Puedes hacerlo también como reacción a un botón:

kotlin — Copiar código

```
btnEnviar.setOnClickListener {
    tvMensaje.text = "Botón pulsado 📌"
}
```

La variable es tvMensaje, .text es el metodo, que es como un seter, para cambiar el texto del textview

EdiText:

⚙️ 4️⃣ Usar el EditText

El `EditText` sirve **para leer texto que el usuario escribió**, por ejemplo cuando se pulsa un botón.

kotlin — Copiar código

```
btnEnviar.setOnClickListener {
    val nombre = etNombre.text.toString()
    tvMensaje.text = "Hola, $nombre 👋"
}
```

Explicación:

- `etNombre.text` devuelve el texto como tipo **Editable** (no es un String).
- `.toString()` lo convierte a texto normal.
- Luego lo usas donde quieras (mostrarlo, guardarlo, etc.).

⚙️ Ahora puedes usarla

Por ejemplo:

kotlin — Copiar código

```kotlin
val texto = etNombre.text.toString()
```

Aquí:

- `etNombre` → tu EditText (el campo que el usuario usa).
- `.text` → obtiene lo que el usuario escribió.
- `.toString()` → lo convierte en texto normal (`String`).

et es tu varibale editext, para recoger lo que el usuario a escrito tu declaras una variable que se llama texto que es = aetnombre(tueditext).txt.toString(), eso convierte lo que el usuario a escrito a una String guardada en la variable texto.

Button:

Tienes un buton en xml, haces el bindeo, y ahora el boton se llama:
btnEnviar = findViewById(R.id.btnEnviar)
Ahora la variable `btnEnviar` está **conectada con el botón real del XML**.
Puedes acceder a él y cambiar sus propiedades, por ejemplo:
btnEnviar.text = "Presiona aquí"

Listener? Lo que sucede cuando pulsas el boton, los listener van debajo de los adaptadores si los hubiese.

⚙️ 4️⃣ Usar el botón (listener principal)

El `Button` reacciona a pulsaciones con un método llamado `setOnClickListener { }`.

Ejemplo:

kotlin — Copiar código

```kotlin
btnEnviar.setOnClickListener {
    Toast.makeText(this, "Botón pulsado 🚀", Toast.LENGTH_SHORT).show()
}
```

checkbox

declarlas la variable, private lateinit var checkNoticias: CheckBox, haces el bindeo ,checkNoticias = findViewById(R.id.checkNoticias)

⚙️ 4️⃣ Listener principal: `setOnCheckedChangeListener`

El método que "escucha" los cambios (cuando se marca o desmarca):

```
checkNoticias.setOnCheckedChangeListener { _, isChecked ->
   if (isChecked) {
      Toast.makeText(this, "Activado ✅", Toast.LENGTH_SHORT).show()
   } else {
      Toast.makeText(this, "Desactivado ❌", Toast.LENGTH_SHORT).show()
   }
}
```

🔍 Qué pasa aquí:

- El primer parámetro `_` es el propio CheckBox (no lo necesitas, por eso se pone `_`).
- `isChecked` es un **booleano** → `true` si está marcado, `false` si no.
- El código dentro se ejecuta **cada vez que el usuario cambia el estado**.

Switch

es como el checkbox pero visualmente parece un interruptor.

Creamos la variable en kotlin private lateinit var switchModo: Switch, switchModo = findViewById(R.id.switchModo), y metemos el listener

⚙️ 4️⃣ **Listener principal →** `setOnCheckedChangeListener {}`

Detecta cada vez que el usuario lo activa o desactiva:

```
switchModo.setOnCheckedChangeListener { _, isChecked ->
    if (isChecked) {
        Toast.makeText(this, "🌙 Modo oscuro activado", Toast.LENGTH_SHORT).show()
    } else {
        Toast.makeText(this, "☀️ Modo claro activado", Toast.LENGTH_SHORT).show()
    }
}
```

Spinner (primer widget que necesita adaptador)

El Spinner es básicamente un *menú desplegable* (como un combo box de escritorio), ideal para elegir una opción de una lista corta.

Declaramos variable private lateinit var themeSpinner: Spinner, hacemos bindeo, themeSpinner = findViewById(R.id.themeSpinner)

⚙️ 4️⃣ **Crear el adaptador**

Aquí entra el **ArrayAdapter**, que actúa como puente entre tus datos y el Spinner.

```
val temas = arrayOf("Cumpleaños", "Boda", "Graduación", "Navidad")

val adaptador = ArrayAdapter(
    this,                           // contexto (la Activity)
    android.R.layout.simple_spinner_item,     // layout predefinido por Android
    temas                           // array de datos
)
adaptador.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
themeSpinner.adapter = adaptador
```

creamos un array para mostrar los datos, creamos el adaptador, (val adaptador) y luego lo incorporamos adaptador.setDropDownView….

Y luego metemos el **listener.**

🧩 5️⃣ **Listener →** `onItemSelectedListener`

Una vez el usuario elige algo, puedes capturarlo con este método:

```
themeSpinner.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
    override fun onItemSelected(parent: AdapterView<*>, view: View?, position: Int, id: Long) {
        val temaSeleccionado = parent.getItemAtPosition(position).toString()
        Toast.makeText(this@MainActivity, "Elegiste: $temaSeleccionado", Toast.LENGTH_SHORT).show()
    }
    override fun onNothingSelected(p0: AdapterView<*>?) {}
}
```

- `position` → índice del elemento seleccionado (0, 1, 2, ...).
- `getItemAtPosition()` → obtiene el valor real (por ejemplo `"Boda"`).
- `onNothingSelected()` se ejecuta si el usuario no selecciona nada (normalmente lo dejas vacío).

RadioGroup+RadioButton

RadioGroup + RadioButton, que son los clásicos para **elegir una sola opción** entre varias (como “sexo: hombre / mujer” o “modo: claro / oscuro”).

👉 **Importante:**

- Solo **un RadioButton puede estar activo a la vez** dentro del mismo grupo.
- El `RadioGroup` se encarga de gestionar eso automáticamente.

Inicias las variables , por un lado inicias la variable del radiogroup, y luego inicias una variable de tipo textview, para mostrar el resultado, y creas el listener
grupoOpciones = findViewById(R.id.grupoOpciones)
tvResultado = findViewById(R.id.tvResultado)

⚙️ 4️⃣ Listener principal → `setOnCheckedChangeListener`

Detecta cuando el usuario selecciona una opción:

```
grupoOpciones.setOnCheckedChangeListener { _, checkedId ->
    val radioSeleccionado = findViewById<RadioButton>(checkedId)
    tvResultado.text = "Has elegido: ${radioSeleccionado.text}"
}
```

🧩 Qué pasa aquí:

- `checkedId` → te da el **id** del `RadioButton` marcado.
- `findViewById<RadioButton>(checkedId)` → lo buscas para acceder a su texto.
- `radioSeleccionado.text` → te da el texto de esa opción (“Opción 2”, por ejemplo).

ejemplo en xml con boton para confirmar y textview que te muestra el resultado de la selección

ImageView

para mostrar imagenes.

🧱 1️⃣ En el XML

xml — Copiar código

```
<ImageView
    android:id="@+id/imgLogo"
    android:layout_width="150dp"
    android:layout_height="150dp"
    android:src="@drawable/logo_android"
    android:contentDescription="@string/desc_logo_android" />
```

Qué hace cada parte:

- `android:id="@+id/imgLogo"` → el id para conectarlo con Kotlin.
- `android:src="@drawable/logo_android"` → imagen que mostrará (guardada en `res/drawable`).
- `android:contentDescription` → texto alternativo para accesibilidad (obligatorio por buenas prácticas).
- `layout_width` y `layout_height` → tamaño de la imagen.

📁 Dónde va la imagen

Las imágenes se guardan dentro de:

css — Copiar código

```
app/src/main/res/drawable/
```

Por ejemplo:

bash — Copiar código

```
res/drawable/logo_android.png
```

MODO DESARROLLADOR

Iniciamos la variable private lateinit var imgLogo: ImageView, bindeo, imgLogo = findViewById(R.id.imgLogo),

para cambiar lo que se muestra usamos este metodo:

imgLogo.setImageResource(R.drawable.logo_kotlin)

logo_kotlin es el nombre de nuestra imagen

ejemplo:

```
class MainActivity : AppCompatActivity() {

    private lateinit var imgLogo: ImageView
    private lateinit var btnCambiar: Button
    private var alternar = false

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Bindeo
        imgLogo = findViewById(R.id.imgLogo)
        btnCambiar = findViewById(R.id.btnCambiar)

        // Listener del botón
        btnCambiar.setOnClickListener {
            if (alternar) {
                imgLogo.setImageResource(R.drawable.logo_android)
            } else {
                imgLogo.setImageResource(R.drawable.logo_kotlin)
            }
            alternar = !alternar
        }
    }
}
```

🧩 1️⃣ RECYCLER VIEW — la lista moderna de Android

🧠 ¿Qué es?

Un `RecyclerView` sirve para mostrar **listas grandes o dinámicas** (por ejemplo, contactos, productos, mensajes...).

Es el reemplazo del antiguo `ListView`.

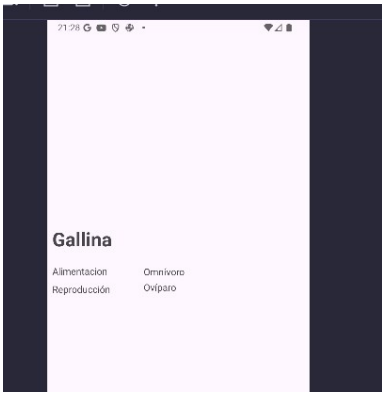
La gracia es que **reutiliza** ("recicla") las vistas para no gastar memoria.

🧱 ESTRUCTURA COMPLETA

Un RecyclerView necesita **3 piezas** para funcionar:

Pieza	Qué hace
1️⃣ **RecyclerView (vista)**	Es el contenedor (en el XML).
2️⃣ **LayoutManager**	Decide cómo se ordenan los ítems (vertical, horizontal, grid...).
3️⃣ **Adapter (adaptador personalizado)**	Crea y une cada elemento con sus datos.

Explicamos el recycler en base al ejercicio de la practica de animales.
Primer paso crear la clase animales que es la que contiene la info que se va a mostrar, nombre, tipo alimentacion etc **la finalidad:**



🧩 PASO 1️⃣ — Clase `Animales.kt` (los datos)

kotlin — Copiar código

```
data class Animales(
    val nombre: String,
    val tipoAlimentacion: String,
    val tipoReproduccion: String
)
```

🔍 Qué hace:

- Es un **modelo de datos** (data class) → define la estructura que tendrá cada animal.
- Cada objeto `Animales` tiene tres propiedades:
  - `nombre`
  - `tipoAlimentacion`
  - `tipoReproduccion`

📌 Así es como defines *qué información tendrá cada elemento* de tu lista.

👉 Ejemplo de un objeto:

kotlin — Copiar código

```
Animales("Gato", "Carnívoro", "Vivíparo")
```

paso 2 el recycler en xml, este xml se escribe en el xml del MAIN_ACTIVITY.XML

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler"
    android:layout_width="409dp"
    android:layout_height="729dp"
    tools:ignore="MissingConstraints" />
```

🔍 **Qué hace:**

- Es tu **pantalla principal**.
- Solo contiene el `RecyclerView`, que será el contenedor de toda la lista.
- Aquí no hay contenido aún: el RecyclerView se llena dinámicamente desde Kotlin usando el **Adapter**.

📌 Piensa en este XML como "la hoja vacía" donde se imprimirá toda la lista.

paso 3: el creamos otro xml, independiente, que se llame fila_animales o como te salga de los huevos y metemos el xml de lo que vamos a mostrar, quieres mostrar botones? Puedes hacerlo, quieres mostrar solo texview? Pos textview

```
<?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"
    xmlns:app="http://schemas.android.com/apk/res-auto">



    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_percent="0.35" />

    <TextView
        android:id="@+id/filaTitulo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="Animal"
        android:textSize="30sp"
        android:textStyle="bold"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/filaAlimentacion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="20dp"
        android:text="Alimentacion"
        android:textSize="15sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/filaTitulo" />

    <TextView
        android:id="@+id/tipoAlimentacion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="21dp"
        android:text="tipo_alim"
        android:textSize="15sp"
        app:layout_constraintStart_toStartOf="@+id/guideline"
        app:layout_constraintTop_toBottomOf="@+id/filaTitulo" />

    <TextView
        android:id="@+id/vivip_ovip"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:text="Reproducción"
        android:textSize="15sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/filaAlimentacion" />

    <TextView
        android:id="@+id/tipoReproduccion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:text="vivip/ovip"
        android:layout_marginTop="4dp"
        android:textSize="15sp"
        app:layout_constraintStart_toStartOf="@+id/guideline"
        app:layout_constraintTop_toBottomOf="@+id/tipoAlimentacion" />


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

si te das cuenta son los textsview que se ven en la pantalla, titulo(Animal) alimentacion, reproduccion, y el resultado

Paso 4: crear el adaptador , habian widgets que necesitaban adaptador y otros que no, pues este necesita. El adaptador es este: creas una clase kotlin y escribes;

```
class AnimalesAdapter(private val listaAnimales: List<Animales>) :
    RecyclerView.Adapter<AnimalesAdapter.AnimalesViewHolder>() {
```

🔍 **Qué hace:**

- Recibe la lista de animales como parámetro (`List<Animales>`).
- Usa esa lista para ir creando una vista por cada elemento.

ahora hacemos como un minibindeo de la vista, de los textview que hemos creado en el xml de fila animales

```
class AnimalesViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val nombre: TextView = itemView.findViewById(R.id.filaTitulo)
    val tipoAlimentacion: TextView = itemView.findViewById(R.id.tipoAlimentacion)
    val tipoReproduccion: TextView = itemView.findViewById(R.id.tipoReproduccion)
}
```

Ahora dices pero como hemos hecho el bindeo si no hemos enlazado el xml? Pos con este metodo, lo cargas "momentaneamente" date cuenta que esto no es una activity, entonces no haces el setcontentview con el xml.

***en .inflate(R.layout.fila_animales (le pasas el xml del adapter, el xml que has hecho)**

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): AnimalesViewHolder {
    val vista = LayoutInflater.from(parent.context)
        .inflate(R.layout.fila_animales, parent, false)
    return AnimalesViewHolder(vista)
}
```

🧩 Qué hace:

- "Infla" (crea en memoria) el XML `fila_animales.xml` .
- Lo convierte en un objeto `View` .
- Lo mete dentro de un `ViewHolder` (la clase de arriba).
- En resumen: **crea la fila vacía**.

ahora si que hacemos como una asociacion entre lo que contiene el objeto de animales y la asociacion del xml,

```
override fun onBindViewHolder(holder: AnimalesViewHolder, position: Int) {
    val animal = listaAnimales[position]
    holder.nombre.text = animal.nombre
    holder.tipoAlimentacion.text = animal.tipoAlimentacion
    holder.tipoReproduccion.text = animal.tipoReproduccion
}
```

- `nombre` → apunta al TextView del XML con id `filaTitulo`
- `tipoAlimentacion` → apunta al TextView con id `tipoAlimentacion`
- `tipoReproduccion` → apunta al TextView con id `tipoReproduccion`

por ultimo

```
override fun getItemCount(): Int = listaAnimales.size
```

eso devuelve el numero de objetos que tienes en el array que has creado para pasarle los parametros

LUEGO EN LA CLASE MAIN

creamos un array de objetos de tipo animales.

```
val animales = listOf(
    Animales("Conejo", "Herbívoro", "Vivíparo"),
    Animales("Gallina", "Omnívoro", "Ovíparo"),
    Animales("Águila", "Carnívoro", "Ovíparo"),
    Animales("Jirafa", "Herbívoro", "Vivíparo"),
    Animales("Cocodrilo", "Carnívoro", "Ovíparo"),
    Animales("Elefante", "Herbívoro", "Vivíparo"),
    Animales("Gato", "Carnívoro", "Vivíparo"),
    Animales("Ratón", "Omnívoro", "Vivíparo"),
    Animales("Murciélago", "Omnívoro", "Vivíparo"),
    Animales("Perro", "Carnívoro", "Vivíparo"),
    Animales("Pingüino", "Carnívoro", "Ovíparo"),
    Animales("Abeja", "Herbívoro", "Ovíparo")
)
```

los parametros que van dentro son los que hemos definido en la class date animales (Nombre,alimentacion,reproduccion)

// RecyclerView
val recycler = findViewById<RecyclerView>(R.id.recycler)
recycler.layoutManager = LinearLayoutManager(this)
recycler.adapter = AnimalesAdapter(animales)

🔍 **Qué hace cada línea:**

1. **findViewById** → vincula el RecyclerView del XML con Kotlin.
2. **layoutManager** → dice cómo se ordenan los elementos.
   - `LinearLayoutManager` = vertical (una lista normal).
3. **adapter** = **AnimalesAdapter(animales)** →
   conecta la lista de datos con el RecyclerView mediante tu adaptador.

📌 Resultado final:

El RecyclerView recorre tu lista, y para cada `Animales`, crea una vista (basada en `fila_animales.xml`) y muestra su contenido.

Intent (para llamar a otras activitys)

paso 1 crear otra activity, creas una activity con su xml con el contenido que sea.

```
class DetalleActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_detalle)
    }
}
```

paso 2 lanzar la activity con un intent, desde main activity lanzas la activity nueva aquí la lanzamos desde un listener de un boton

```
val boton = findViewById<Button>(R.id.btnAbrir)
boton.setOnClickListener {
    val intent = Intent(this, DetalleActivity::class.java)  // nombre de tu activity
    startActivity(intent)
}
```

📌 Aquí pasa esto:

- `Intent(this, DetalleActivity::class.java)` → crea la "orden" de abrir otra pantalla.
- `startActivity(intent)` → ejecuta esa orden.

pasar variables de una activity a otra con

Puedes mandar información con el intent usando `putExtra()`.

En MainActivity:

kotlin — Copiar código

```kotlin
val intent = Intent(this, DetalleActivity::class.java)
intent.putExtra("nombre", "Conejo")
intent.putExtra("alimentacion", "Herbívoro")
startActivity(intent)
```

En DetalleActivity:

kotlin — Copiar código

```kotlin
val nombre = intent.getStringExtra("nombre")
val alimentacion = intent.getStringExtra("alimentacion")

val tv = findViewById<TextView>(R.id.tvInfo)
tv.text = "Animal: $nombre\nAlimentación: $alimentacion"
```

📌 Esto es como pasar variables de una pantalla a otra.

tambien puedes crear intents para abrir otra cosa que no sea una activity, como por ejemplo un navegador con una url o la camara. Se ejecuta dentro de cualquier listener
Abrir navegador:

```
botonWeb.setOnClickListener {
        val intent = Intent(Intent.ACTION_VIEW)
        intent.data = Uri.parse("https://www.google.com")
        startActivity(intent)
    }
```

abrir camara:
botonWeb.setOnClickListener {
val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
startActivity(intent)

}

FRAGMENTS

🧩 1️⃣ ¿Qué es un Fragment?

Un **Fragment** es una *subpantalla* que vive dentro de una **Activity**.

Piensa en él como una **pieza modular** de una interfaz.

📌 En resumen:

> Un Fragment = una mini Activity dentro de otra.

Por ejemplo:

- En una app de música, puedes tener un Fragment con la lista de canciones 🎵
  y otro con los detalles de la canción que suena.
- Todo dentro de una sola Activity principal.

⚙️ 3️⃣ Estructura básica de un Fragment

Un fragmento tiene:

1. Un archivo Kotlin (`MiFragment.kt`)
2. Un layout XML (`fragment_mi.xml`)
3. Se inserta dentro de una Activity (`activity_main.xml`)

1 creas tu clase fragment

```
class MiFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Vincula el XML del fragment con el código
        return inflater.inflate(R.layout.fragment_mi, container, false)
    }
}
```

2 creas el xml con las cosas que tenga dentro el fragment.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center">

    <TextView
        android:id="@+id/tvTexto"
        android:text="Hola, soy un Fragment “
        android:textSize="20sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

3 en el activity_main.xml tienes que crear un objeto framelayout

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

📌 **`FrameLayout` sirve como contenedor donde cargarás tu fragment dinámicamente.**

4 Cargar el fragment desde la activity

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val fragment = MiFragment()

        supportFragmentManager.beginTransaction()
            .replace(R.id.contenedor, fragment)
            .commit()
    }
}
```

🧠 Qué pasa aquí:

- `supportFragmentManager` → controla los fragments.
- `beginTransaction()` → abre una "transacción" (una operación de cambio de fragmentos).
- `replace()` → reemplaza lo que haya en el contenedor por tu fragment.
- `commit()` → ejecuta el cambio.

como cambiar el fragment con un boton

```
val boton = findViewById<Button>(R.id.btnCambiar)
boton.setOnClickListener {
    val nuevoFragment = SegundoFragment()
    supportFragmentManager.beginTransaction()
        .replace(R.id.contenedor, nuevoFragment)
        .addToBackStack(null) // permite volver atrás
        .commit()
}
```

TOOLBAR

🧩 1️⃣ Qué es la Toolbar

La **Toolbar** es una vista (un widget) que actúa como la **barra superior** de una app.
Reemplaza a la vieja `ActionBar` que venía por defecto, y te da control total sobre su diseño.

📌 En resumen:

> Es la parte superior donde se pone el título, los botones de navegación y los menús.

1 crear la toolbar en el xml de tu main_activity

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:titleTextColor="@android:color/white"
    android:elevation="4dp"
    android:title="Mi App Cool " />
```

🔍 Explicación rápida:

- `actionBarSize` → usa el alto por defecto de las barras de acción.
- `colorPrimary` → color principal de tu tema.
- `titleTextColor` → color del texto.
- `title` → texto inicial.

⚙️ 3️⃣ Activar la Toolbar en Kotlin

En tu `MainActivity.kt`, después del `setContentView()`:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val toolbar = findViewById<Toolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
    }
}
```

🧠 Qué pasa aquí:

- `findViewById()` → encuentra la Toolbar del XML.
- `setSupportActionBar(toolbar)` → le dice a Android "esta es mi barra principal".

4 Podemos cambiar el titulo de la toolbar por ejemplo cuando llamamos a un fragment, metes dentro y ya , tenemos la clase activity

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val fragment = MiFragment()

        supportFragmentManager.beginTransaction()
            .replace(R.id.contenedor, fragment)
            .commit()

supportActionBar?.title = "Pantalla de inicio 🏠" // esto cambia el titulo de la toolbar
    }
}
```

5 añadir menu a la toolbar

📁 **Paso 1: crea un XML de menú**

Dentro de `res/menu/` crea un archivo llamado `menu_principal.xml`:

Puedes meter ahi los item que quieras.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_settings"
        android:title="Configuración"
        android:icon="@drawable/ic_settings"
        android:showAsAction="always" />
    <item
        android:id="@+id/action_info"
        android:title="Acerca de"
        android:showAsAction="never" />
</menu>
```

paso 2

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.menu_principal, menu) // inflamos el menu con el xml
    return true
}
```

EJEMPLO DE OPCIONES REALES DENTRO DEL MENU

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {

        R.id.action_settings -> {
            // Abrir otra pantalla
            val intent = Intent(this, SettingsActivity::class.java)
            startActivity(intent)
            true
        }

        R.id.action_info -> {
            // Mostrar un cuadro de diálogo
            AlertDialog.Builder(this)
                .setTitle("Información")
                .setMessage("Versión 1.0.0 de la app 🧠")
                .setPositiveButton("OK", null)
                .show()
            true
        }

        R.id.action_share -> {
            // Compartir algo por WhatsApp o similar
            val shareIntent = Intent(Intent.ACTION_SEND)
            shareIntent.type = "text/plain"
            shareIntent.putExtra(Intent.EXTRA_TEXT, "Descarga mi app 😍")
            startActivity(Intent.createChooser(shareIntent, "Compartir con..."))
            true
        }

        R.id.action_logout -> {
            // Cerrar sesión o limpiar datos
            val prefs = getSharedPreferences("user_prefs", MODE_PRIVATE)
            prefs.edit().clear().apply()
            startActivity(Intent(this, LoginActivity::class.java))
            finish()
            true
        }

        else -> super.onOptionsItemSelected(item)
    }
}
```

🧠 Qué está pasando aquí

Opción del menú	Qué hace	Tipo de acción
`action_settings`	Abre otra Activity con un `Intent`	Navegación
`action_info`	Muestra una ventana emergente	UI/Dialog
`action_share`	Lanza un intent implícito de compartir texto	Sistema Android
`action_logout`	Limpia datos y vuelve al login	Lógica de app

🧩 Puedes hacer literalmente de todo:

- 🔁 Cambiar de Fragment:

```kotlin
supportFragmentManager.beginTransaction()
    .replace(R.id.contenedor, PerfilFragment())
    .addToBackStack(null)
    .commit()
```

- 🎨 Cambiar un color o texto en pantalla:

```kotlin
findViewById<TextView>(R.id.titulo).text = "Nuevo texto"
```

- 💾 Guardar algo en SharedPreferences:

```kotlin
getSharedPreferences("prefs", MODE_PRIVATE)
    .edit()
    .putString("tema", "oscuro")
    .apply()
```

- 📷 Lanzar la cámara o galería:

```kotlin
val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
startActivity(intent)
```

PONER FLECHA RETROCESO EN TOOLBAR

Mostrar la flecha

supportActionBar?.setDisplayHomeAsUpEnabled(true)

y hacer que funcione

```
override fun onSupportNavigateUp(): Boolean {
    onBackPressedDispatcher.onBackPressed()
    return true
}
```

Ejemplo completo

```
class DetalleActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_detalle)

        val toolbar = findViewById<Toolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)
        supportActionBar?.setDisplayHomeAsUpEnabled(true)
        supportActionBar?.title = "Detalles del animal 🦊"
    }

    override fun onSupportNavigateUp(): Boolean {
        onBackPressedDispatcher.onBackPressed()
        return true
    }
}
```

ViewPage

🧩 1️⃣ Qué es el ViewPager2

El **ViewPager2** es un componente que permite **deslizar entre pantallas o fragments con un swipe horizontal**, tipo "stories de Instagram" o "pantallas de tutorial".

📌 En resumen:

> Es un contenedor que te deja moverte entre varios fragments con un gesto de deslizamiento.

🧠 2️⃣ Diferencia entre ViewPager y ViewPager2

- El viejo `ViewPager` ya está deprecado (obsoleto).
- `ViewPager2` usa `RecyclerView` por dentro → mucho más fluido y flexible
- Soporta **vertical y horizontal**, animaciones y hasta swipe desactivado.

IMPORTANTE:

🧩 OBJETIVO

Tener una app con una sola Activity (`MainActivity`) que muestra **varios fragments** dentro de un **ViewPager2**, para poder **deslizar entre ellos** (como una especie de "pestañas").

Paso 1: la activity principal es el contenedor donde vas a meter los fragments y el viewpage dentro. El activity es como el marco del cuadro pero en realidad no hay nada mas dentro.

Paso 2 crear los fragments: creas las clases de los fragments con su correspondiente xml, cada fragment tendra sus cosas dentro etc etc.

Paso 3 Adaptador crea un .kt (ViewPagerAdapter.kt) con esto dentro que es el adaptador.

```
class ViewPagerAdapter(activity: FragmentActivity) : FragmentStateAdapter(activity) {
    override fun getItemCount(): Int = 3 // Cuántos fragments hay
    override fun createFragment(position: Int): Fragment {
        return when (position) {
            0 -> FragmentUno()
            1 -> FragmentDos()
            2 -> FragmentTres()
            else -> FragmentUno()
        }
    }
}
```

le vas diciendo en cada posición que fragment tiene que usar

4 Modificar el XML de la Activity (activity_main.xml)

Ahora toca crear el contenedor donde se verá el ViewPager.

Si tu Activity tiene más cosas (botones, toolbar, etc.), no pasa nada:
el ViewPager es solo un elemento más dentro del `ConstraintLayout`.

```
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <!-- Toolbar (si tienes una) -->
    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:layout_constraintTop_toTopOf="parent" />

    <!-- ViewPager debajo de la Toolbar -->
    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/viewPager"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintTop_toBottomOf="@id/toolbar"
        app:layout_constraintBottom_toBottomOf="parent" />

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

3 Qué pasa exactamente cuando haces eso

Piensa en el flujo interno:

1 `ViewPagerAdapter(this)`
→ Crea el adaptador (llamada al constructor).
→ Guarda que tiene 3 fragments.

2 `viewPager.adapter = adapter`
→ Le dice al ViewPager: "usa este adaptador para crear tus páginas".

3 El ViewPager dice:

> "Vale, necesito mostrar la página 0, le voy a pedir al adaptador que me dé un fragment."

4 Android llama dentro del adaptador a:

kotlin — Copiar código

```
createFragment(0)
```

5 Tu adaptador devuelve:

kotlin — Copiar código

```
FragmentUno()
```

6 Android infla (`R.layout.fragment_uno`) y lo muestra en pantalla.

PASO 4 ESCRIBIR EL VIEWPAGE EN EL MAIN

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Bindeo
        val viewPager = findViewById<ViewPager2>(R.id.viewPager)

        // Crear el adaptador
        val adapter = ViewPagerAdapter(this)

        // Asignar adaptador al ViewPager
        viewPager.adapter = adapter
    }
}
```

Codigo con comentarios linea a linea

Main

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ① CARGA EL XML DEL ACTIVITY
        // Android infla activity_main.xml → crea la vista principal
        setContentView(R.layout.activity_main)

        // ② BINDEA EL VIEWPAGER DEL XML CON KOTLIN
        val viewPager = findViewById<ViewPager2>(R.id.viewPager)
        // Ahora viewPager en Kotlin apunta al ViewPager2 del XML

        // ③ CREA UNA INSTANCIA DEL ADAPTADOR (LLAMADA DIRECTA AL CONSTRUCTOR)
        val adapter = ViewPagerAdapter(this)
        // Aquí se ejecuta el constructor de ViewPagerAdapter y se guarda en memoria

        // ④ ENLAZA EL ADAPTADOR CON EL VIEWPAGER (AQUÍ OCURRE LA MAGIA)
        viewPager.adapter = adapter
        // En este punto, el ViewPager sabe que tiene que usar este adapter
        // para obtener los fragments que mostrará
    }
}
```

ADAPTER

```
class ViewPagerAdapter(activity: FragmentActivity) : FragmentStateAdapter(activity) {

    // ⑤ ANDROID PREGUNTA: "¿CUÁNTAS PÁGINAS TENGO?"
    override fun getItemCount(): Int = 3

    // ⑥ ANDROID PREGUNTA: "¿QUÉ FRAGMENT DEBO MOSTRAR EN ESTA POSICIÓN?"
    override fun createFragment(position: Int): Fragment {
        return when (position) {
            0 -> FragmentUno()   // crea el objeto del primer fragment
            1 -> FragmentDos()   // crea el objeto del segundo fragment
            2 -> FragmentTres()  // crea el objeto del tercero
            else -> FragmentUno()
        }
    }
}
```

FRAGMENTO 1

```
class FragmentUno : Fragment(R.layout.fragment_uno)
```

XML DE FRAGMENTO 1

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:gravity="center"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:text="Fragment 1 🦊"
        android:textSize="30sp"
        android:textStyle="bold" />
</LinearLayout>
```

🔁 5️⃣ Flujo completo en orden real de ejecución

1. Android ejecuta `MainActivity.onCreate()`
2. Se carga `activity_main.xml`
3. Tú haces `findViewById` → obtienes el `ViewPager2` del XML
4. Creas `ViewPagerAdapter(this)`
5. Asignas `viewPager.adapter = adapter`
6. Android dice: "vale, necesito mostrar el primer fragment (posición 0)"
7. Android llama a `createFragment(0)` dentro del adaptador
8. Tu adaptador devuelve un `FragmentUno()`
9. Android ve que ese fragment está vinculado al layout `fragment_uno.xml`
10. Android infla ese XML → lo muestra en pantalla

💥 Y así sucesivamente cuando deslizas:

- si vas a la posición 1 → se llama `createFragment(1)` → carga `FragmentDos` → infla `fragment_dos.xml`.