

E3. Herramientas de IDE Android Studio

Sitio: [Agencia de Habilidades para el Futuro](#)

Curso: Desarrollo de Aplicaciones para Dispositivos 2° D

Libro: E3. Herramientas de IDE Android Studio

Imprimido por: Eduardo Moreno

Día: miércoles, 3 de septiembre de 2025, 00:08

Descripción

Tabla de contenidos

1. Introducción al libro

2. TEMA I. IDE de un desarrollo móvil

2.1. ¿Qué es un IDE?

3. Android Studio

3.1. Instalación

3.2. Empty activity

3.3. Creación del proyecto

4. App modelo

4.1. Nuestro diseño

5. TEMA II. Controles de Android Studio

6. Tips para la práctica formativa

7. Los controles y el IDE

8. TextView

8.1. Código

8.2. Entorno XML diseño de la Activity

9. EditText

10. RadioButton (Botones de selección)

10.1. ¿Cómo manipulamos al RadioButton?

11. CheckBox (Casilla de verificación)

12. Uso de varias pantallas

12.1. Pasos para crear la nueva pantalla

12.2. Código para pasar a la nueva pantalla

13. TEMA III. Vínculo con Base de Datos

14. Android y base de datos

15. Estructura de la base de datos

16. SQLite

16.1. Constructor y métodos de la clase SQLiteOpenHelper

16.2. Métodos de la clase SQLiteDatabase

17. Ejemplos con base de datos

17.1. Diseño del ejemplo

17.2. Código de creación de los datos

17.3. Override del onCreate y del onUpgrade

17.4. Código para alta del ABM

17.5. Código para mostrar los datos

17.6. Código para modificación y baja del ABM

18. TEMA IV. Preparar el entorno del proyecto en Android Studio

19. Introducción al entorno del proyecto

20. Cómo visualizar la base de datos en el IDE

21. Creación de la base con varias tablas

22. Comenzando con la app del proyecto "Club Deportivo"

23. Primeros tips para el desarrollo de la app

23.1. Cambio de activity de inicio

23.2. Nuevo IDE nuevos usuarios



Introducción

En este libro se presentan cuatro temas relacionados, que permiten conocer a fondo una herramienta fundamental en el desarrollo de aplicaciones para dispositivos móviles: [Android Studio](#)

Los cuatro temas que encontrarás son:

- I** IDE de un desarrollo móvil
- II** Controles de Android Studio
- III** Vínculo con base de datos
- IV** Preparar el entorno del proyecto con Android Studio

Te recomendamos leer con atención el contenido y, al mismo tiempo, hacer pruebas sobre la herramienta. Recordá que se trata de un instrumento ¡necesita práctica!

¡Comenzamos!



IDE de un desarrollo móvil

¿Alguna vez te preguntaste cómo los desarrolladores solían construir aplicaciones antes de la llegada de los Entornos de Desarrollo Integrados (IDEs)?

La realidad era tediosa: cambiar entre editores de texto, compiladores y herramientas de verificación de errores consumía tiempo y esfuerzo. Pero, ¿qué revolucionó esta dinámica? [¡Los IDEs!](#)

Definición

Un **IDE**, más que un simple editor, es un conjunto de herramientas que simplifica el desarrollo de *software*. Por ejemplo, Android Studio, el IDE oficial para aplicaciones en Android, ofrece un entorno que reúne todo lo esencial para el desarrollo de la aplicación.

Siguiendo pistas, consejos, y analizando una app modelo, aprenderás a gestionar eventos, crear funciones y tomar decisiones en tu código, llevando tu proyecto a un nuevo nivel.

Preguntas clave de este tema:

- ¿Qué diferencia crucial existía en el proceso de desarrollo antes y después de la introducción de los entornos de desarrollo integrados (IDEs)?
- ¿Cuáles son las funcionalidades básicas que componen un IDE?
- ¿Cuál es el propósito y la importancia de Android Studio en el desarrollo de aplicaciones para Android?
- ¿Qué tipos de módulos están presentes en los proyectos de Android Studio, y cómo se organizan estos proyectos?
- ¿Qué elementos se encuentran dentro de un proyecto creado en Android Studio, y qué representan en el contexto del desarrollo de una aplicación?

¿Listo/a para empezar? ¡Adelante, el mundo del desarrollo te espera!



¿Qué es un IDE?

Antes de la llegada de los IDE, los desarrolladores utilizaban simples editores de texto para codificar, guardar la aplicación en un editor de texto, ejecutar el compilador, comprobar si había errores y volver al editor para comprobar el código. Todo este proceso consumía mucho tiempo y esfuerzo del desarrollador, ya que siempre tenía que cambiar entre varias aplicaciones.

Aquí es donde entra en juego un IDE que reúne todas las herramientas esenciales del desarrollador bajo un mismo marco.

Definición

IDE es un programa de software o una amalgama de herramientas que necesita para escribir y probar su software. En resumen, un IDE es una combinación de herramientas básicas necesarias para el desarrollo de aplicaciones.

El IDE consta al menos de

- ❖ un editor de texto,
- ❖ herramientas de automatización de la construcción y,
- ❖ un depurador.

Además, algunos IDE vienen con la ventaja de poder instalar *plugins* para ampliar sus funcionalidades a otro nivel.



Android Studio

Android Studio es el entorno de desarrollo integrado (IDE) oficial que se usa en el desarrollo de apps para Android.

Cada proyecto de Android Studio incluye uno o más módulos con archivos de código fuente y archivos de recursos. Entre los tipos de módulos, se incluyen los siguientes:

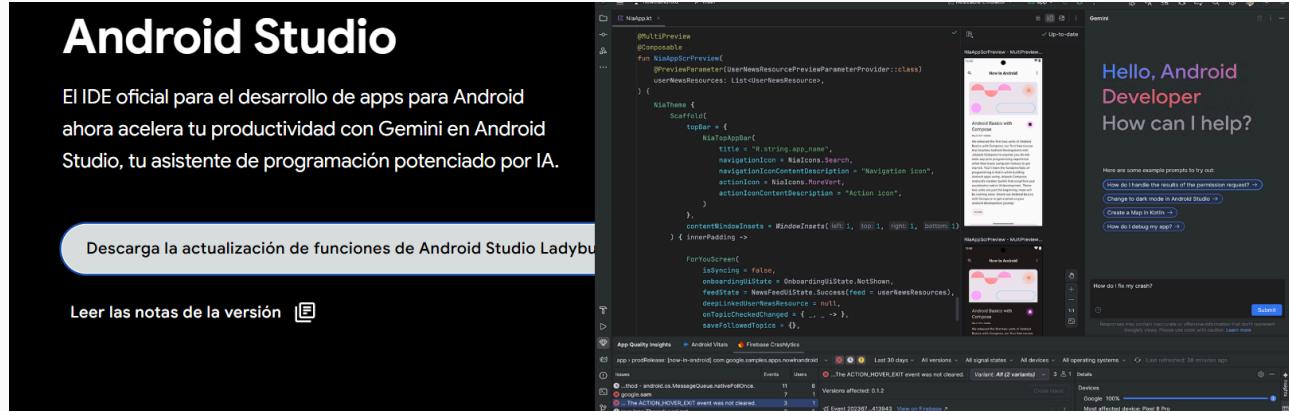
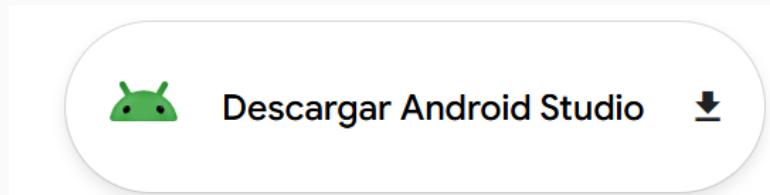
- ❖ Módulos de apps para Android
- ❖ Módulos de biblioteca.
- ❖ Módulos de Google App Engine

Los proyectos de Android Studio

contienen todo lo que define tu espacio de trabajo para una app: desde código fuente y recursos hasta código de prueba y configuraciones de compilación.

Cuando comienzas un proyecto nuevo, Android Studio crea la estructura necesaria para todos los archivos y los hace visibles en la ventana Project de Android Studio.

Vamos a la web oficial: <https://developer.android.com/>



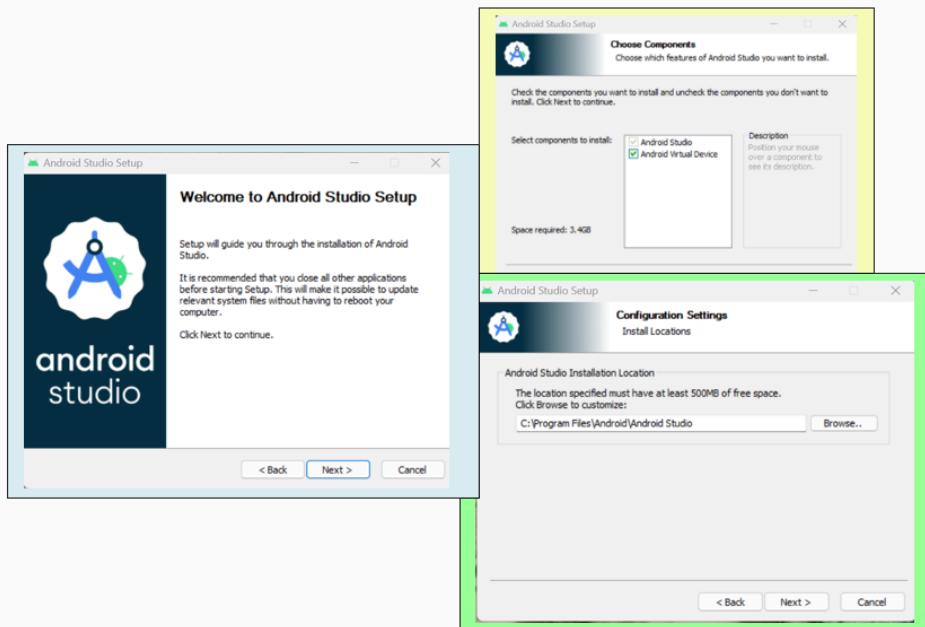
Veamos de que se trata...



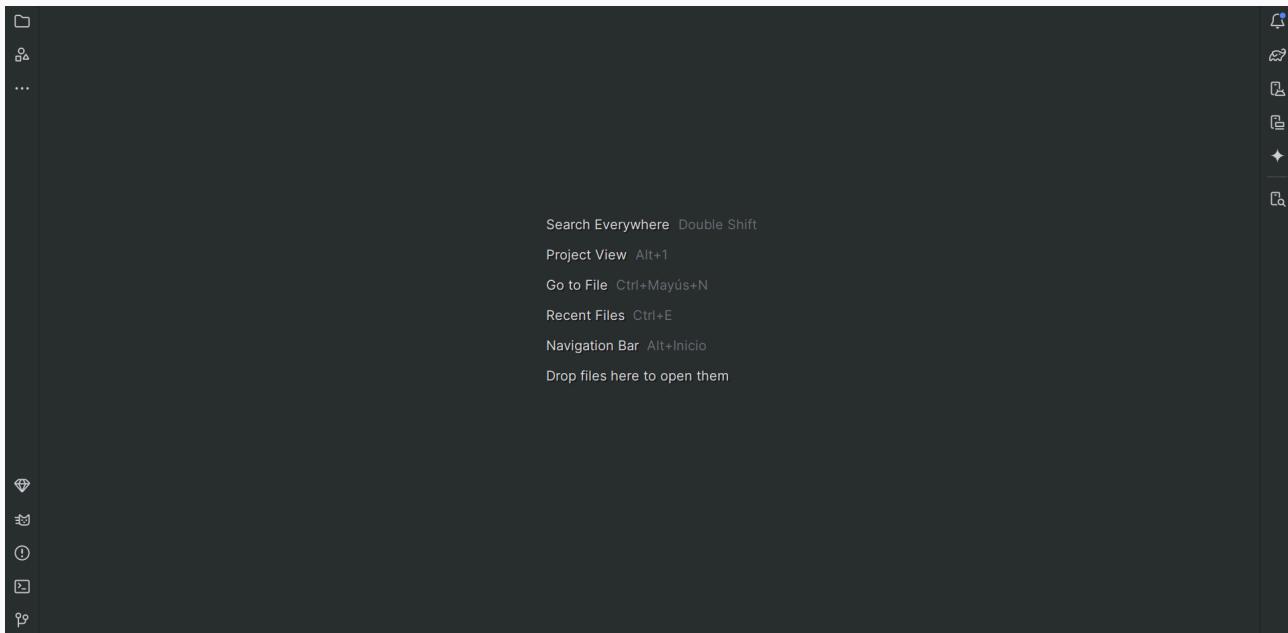
Instalación

Lo primero es [descargar el Android Studio](#) para proceder a la instalación siguiendo las instrucciones.

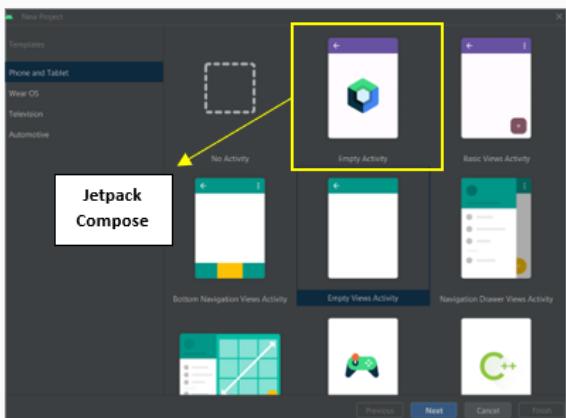
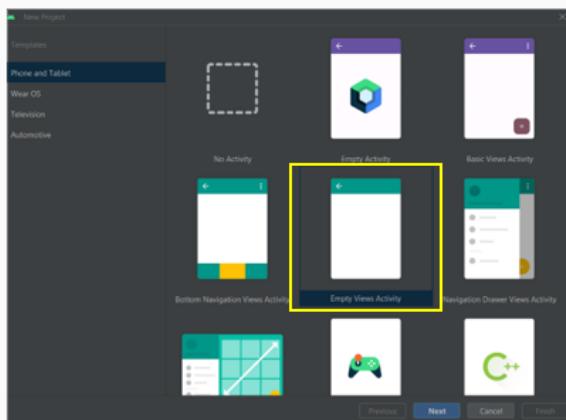
A medida que avances en la instalación verás las siguientes ventanas (pueden cambiar en función de la versión):



Llegarás a:



Al crear un nuevo proyecto nos encontramos con una serie de plantillas:



Te preguntarás cuál es la diferencia, Veamos...



Empty Activity vs Empty Views Activity

- **Empty Views Activity:** Crea una actividad vacía con la estructura mínima necesaria para empezar a desarrollar una aplicación. Se usa XML para definir la interfaz y código en Kotlin para manejar la lógica.
- **Empty Activity:** Es similar a la *Empty Activity*, pero ya viene configurada para trabajar con una forma más moderna de interactuar con los elementos de la interfaz, facilitando la manipulación de las vistas dentro del código, trabajando con Jetpack Compose. No se trabaja con XML, se integra vista y lógica dentro de Kotlin.

Diferencia entre Jetpack Compose y XML

Para diseñar interfaces en Android, hay dos enfoques principales:

XML (Views tradicionales)

- **Usa archivos XML para definir la interfaz de usuario.**
- **Se organiza con componentes como TextView, Button, LinearLayout, etc.**
- **Es el método más utilizado en proyectos existentes y ha sido el estándar en Android por mucho tiempo.**

Jetpack Compose

- **Permite construir interfaces de usuario con código Kotlin puro, sin necesidad de archivos XML.**
- **Se basa en un enfoque declarativo, más moderno y flexible.**
- **Requiere aprender nuevas formas de estructurar la UI, diferentes a las vistas tradicionales, algo más complejas.**

¿Por qué trabajaremos con XML?

- **Es el enfoque más utilizado actualmente en aplicaciones en producción.**
- **Permite comprender mejor la estructura de la interfaz y la relación entre diseño y lógica.**
- **Facilita la transición a otros conceptos como Fragments y Activity.**

Más adelante, si el tiempo lo permite, exploraremos *Jetpack Compose*.

¿Qué es una *activity*?

Activity o actividad es lo más básico y utilizado en el desarrollo de Android, se considera que una pantalla es un *activity* por lo tanto si nuestra aplicación tiene 3 pantallas podemos decir que tiene 3 *activity*. Cada *activity* está formada por dos partes, una parte lógica y una parte gráfica.

Si seleccionamos `empty views activity` la parte lógica es un archivo `.java` o un archivo `.kt` (si el lenguaje es `kotlin`) que nos muestra la clase que se crea para poder manipular, interactuar y colocar el código de esa actividad. La parte gráfica es un XML que tiene todos los elementos que estamos viendo de una pantalla declarados con etiquetas parecidas a las de HTML.

Si seleccionamos empty activity la parte lógica y la parte gráfica están unificadas en un mismo archivo .kt

Una vez elegida la *activity* avanzamos con la [creación del proyecto](#).



Creación del proyecto

Vemos la siguiente pantalla para asignarle el nombre y la ubicación:



Una vez que creamos el nuevo proyecto, llegamos al entorno de desarrollo de Android Studio.

Si es la primera vez que se trabaja con **Android Studio** y ejecutamos, se produce un error porque no tenemos seleccionado ningún dispositivo que sea capaz de ejecutar nuestra app.

Tendremos dos **opciones para proceder a la ejecución:**

- **Ejecutarlo en un emulador**, es decir, una utilidad que simule un dispositivo Android.
- **Ejecutarlo en un dispositivo físico.**

Para ver las partes del IDE vamos a mostrar en una app un botón con la leyenda “**presione aquí**” y, al hacerlo, que muestre un mensaje.

Manos a la obra...



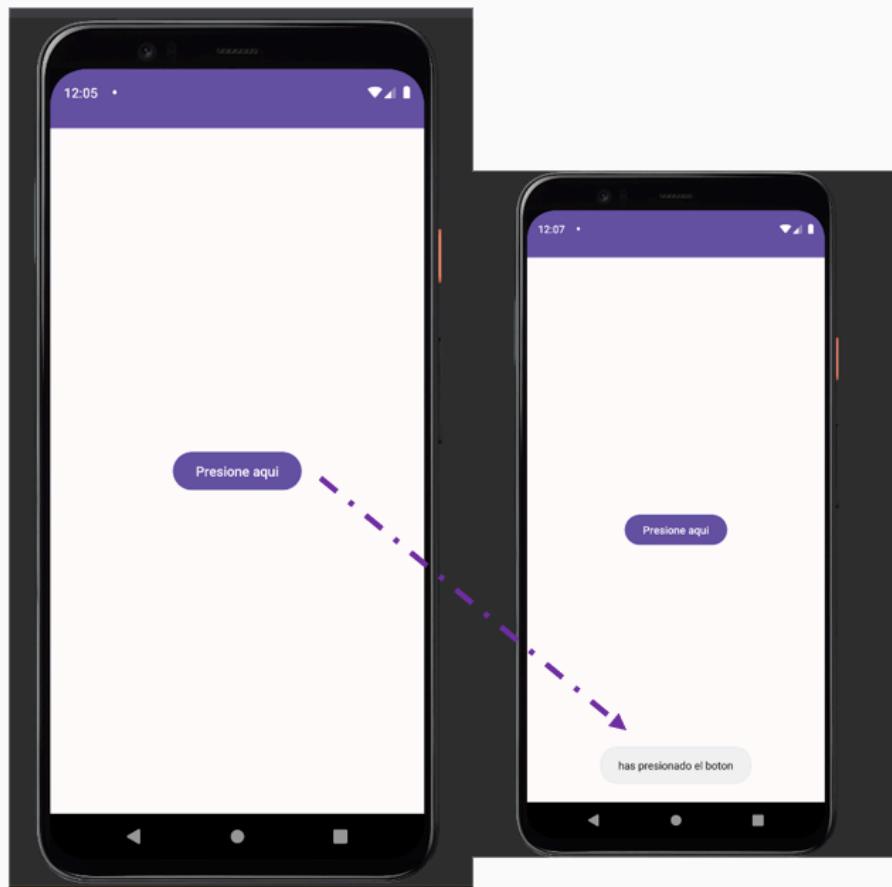
App modelo

Comencemos con el modelo de una App:

1. Creamos una [empty views activity](#)
2. Almacenamos el proyecto en el directorio bajo un nombre y seleccionamos como lenguaje a [Kotlin](#)
3. El IDE nos muestra en el extremo de la izquierda un árbol con la carpeta [app](#) y la carpeta [Gradle Scripts](#)

[App](#) es nuestra aplicación

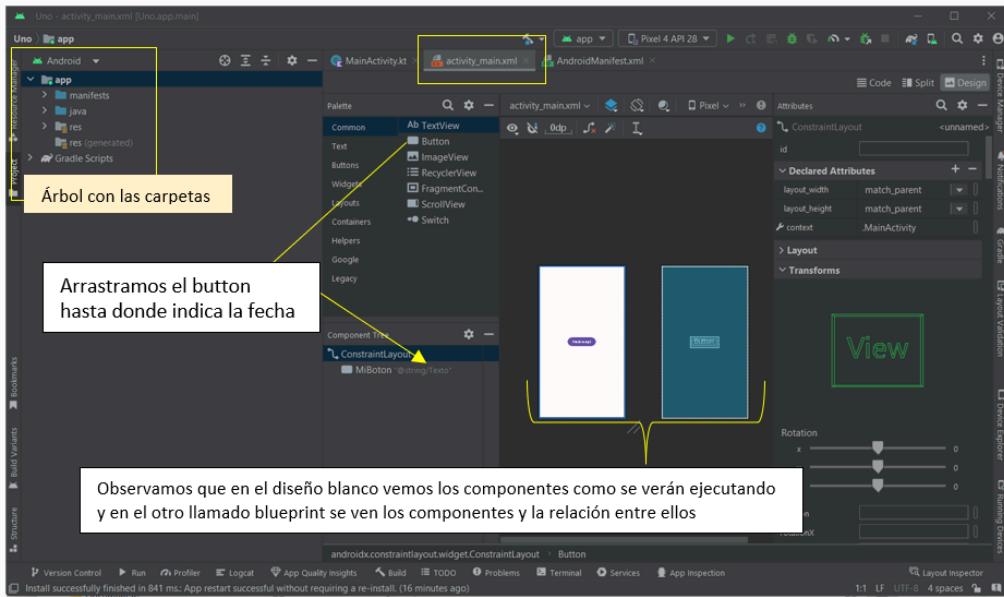
- En Manifests encontramos un archivo llamado `AndroidManifest.xml` que es el archivo de configuración principal donde se configuran las pantallas. Es una declaración de lo que tiene la aplicación y sus permisos. Acceso a internet. Escritura en el teléfono. Tiene estipulado cuantas pantallas tiene la aplicación, cual es el nombre, el icono, etc.
- Otro archivo es “`MainActivity`” que contiene la programación de nuestra app.
- En `res` se encuentra `layout` que contiene la parte gráfica.



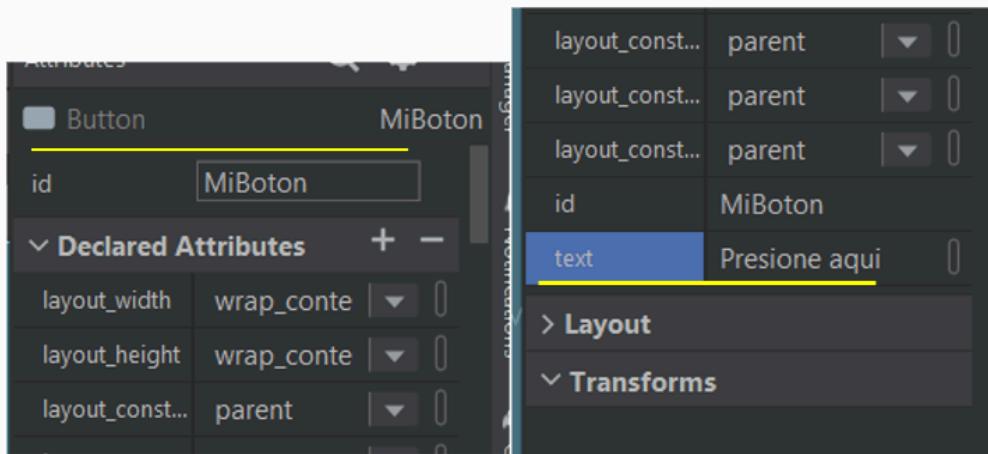
Construyamos el diseño...



Nuestro diseño



En la barra de los Atributos (ubicado en el lateral derecho) le damos nombre al componente y el texto que contiene. El nombre es el atributo **id** y el texto en el atributo **text**.



Construyamos el código en **MainActivity.kt**.

```
activity ) { onCreate
    MainActivity.kt
```

```
2
3     import ...
7
8     class MainActivity : AppCompatActivity() {
9         override fun onCreate(savedInstanceState: Bundle?) {
10             super.onCreate(savedInstanceState)
11             setContentView(R.layout.activity_main)
12
13             /* =====
14             * Se declara una variable tipo button para utilizar
15             * el objeto del diseño
16             ===== */
17
18             val button: Button = findViewById(R.id.MiBoton)
19             /* =====
20             * En el evento OnClick mostramos un mensaje
21             ===== */
22             button.setOnClickListener{ it: View!
23                 Toast.makeText( context: this, text: "has presionado el boton",Toast.LENGTH_SHORT).show()
24             }
25
26
27         }
28
29     }
30 }
```

Código por defecto

Referenciamos al botón del diseño

Crea el mensaje

Accedé al código haciendo clic [aquí](#).



Controles de Android Studio

¡Comenzamos un nuevo paso de nuestro recorrido! En este tema nos sumergiremos en el universo de los **controles para nuestras aplicaciones**. Olvidemos las definiciones aburridas y demos paso a ejemplos cotidianos y prácticos.

¿Alguna vez quisiste crear esos botones "Aceptar" o "Cancelar"? Pues bien, vamos a hacerlo. Y eso no es todo, hablaremos de **TextView, EditText, RadioButton y CheckBox**. Imagina diseñar una app de compras donde podemos elegir nuestra talla de remera, la cantidad y hasta el método de pago.

Además, **aprenderemos a cambiar de una pantalla a otra en una app**. Es como pasar de un enlace a otro en internet. ¡Emocionante, verdad? Así que preparate para una semana práctica y divertida.

Preguntas clave del tema:

- ¿Cómo personalizaríamos un TextView en una aplicación para mostrar un mensaje específico?
- ¿Cuáles son las diferencias clave entre un RadioButton y un CheckBox en Android?
- Expliquen cómo crear un nuevo proyecto en Android Studio y agregar una segunda pantalla (*activity*).
- ¿Qué es un Intent en el contexto de desarrollo de aplicaciones móviles y cuándo lo usaríamos?
- ¿Cómo manipularíamos los atributos de un EditText para que acepte solo un tipo específico de entrada, como un número de teléfono o una contraseña?

¿Listo/a para empezar?



Tips para la práctica

Definición

Un **tip** es un dato o una pista para resolver un problema. Teniendo en cuenta esto, te mostraremos algunas pistas para aplicar en la actividad propuesta.

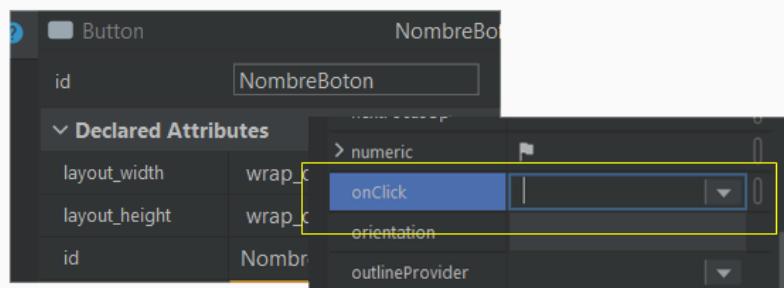
Cuando insertamos en el diseño componentes, estos esperan que desde el exterior se accione algún evento. ¿Te acordás de desarrollo de sistemas orientados a objetos? En los formularios colocamos botones, cajas de texto, opciones, cajas de validación entre otros objetos y programamos en los eventos correspondientes para procesar los datos recibidos.

Veamos entonces:

- Si la App tiene botones el evento por defecto es el **click**, podemos tratarlo desde:
 - **Código**

```
NombreBoton.setOnClickListener{  
}
```

- Crear una función y agregarla al **OnClick** de los atributos



- ¿Cómo creamos una función?: en este caso el parámetro de la función es **view: View**. ¿Qué es un **View**? Es un componente que permite controlar la interacción del usuario con la aplicación.

```
fun NombreFuncion(view: View){  
}
```

- Podemos usar lo que conocemos como múltiples opciones en la toma de decisiones, es decir un **Case**, en Kotlin se llama **when() {}**. En los paréntesis se coloca la variable o atributo del componente que puede tomar múltiples valores y que en función del dominio es el código que debe ejecutar.
- Es conveniente el uso de las banderas (término utilizado mucho en programación) es decir valores en una variable que tengan un significado propio.
- En este caso para identificar las diferentes operaciones, por ejemplo, 1 para suma, 2 para resta, 3 para multiplicación, 4 para división.



Los controles y el IDE

A esta altura de la cursada ya estás familiarizado/a con los términos **clases, instancia y objetos**.

Muchos programadores cuando instancian una clase y obtienen un objeto para colocarlo en el diseño de una programación le dan el nombre de **control**. Se comienza a manipular los atributos de esos controles para personalizarlos en función del diseño que queremos lograr.

En nuestro intercambio cotidiano con los dispositivos nos encontramos con un botón “Aceptar” o “Cancelar”, o con mensajes que nos brindan información. Lo interesante es que no debemos preocuparnos por crearlos sino que ya vienen con el IDE.

Nuestra tarea:

se centra en modificar sus propiedades: tamaño, color, etc. para incorporarlos en nuestras aplicaciones y asociarles el código necesario para que se comporten como esperamos al ejecutar la app.

¿Qué esperamos?



TextView

Definición

Es un **componente** que podemos incluir en las vistas de la aplicación en la que estamos trabajando con el propósito de presentarle un texto al usuario.

Se utiliza mucho en las interfaces de usuario, debido a que nos da la posibilidad de presentar títulos, contenido que pretende ser informativo.

¿Cómo lo creamos en el diseño?

Podemos hacerlo desde código o desde la ventana de controles que nos proporciona el entorno XML de diseño de la *activity*.

Analicemos los dos casos el código y la ventana del entorno.



Código

Se utiliza la etiqueta asignada “`TextView`”. Con ella se le indica al programa lo que queremos incluir y, luego, se especifica los diferentes parámetros para el mismo.

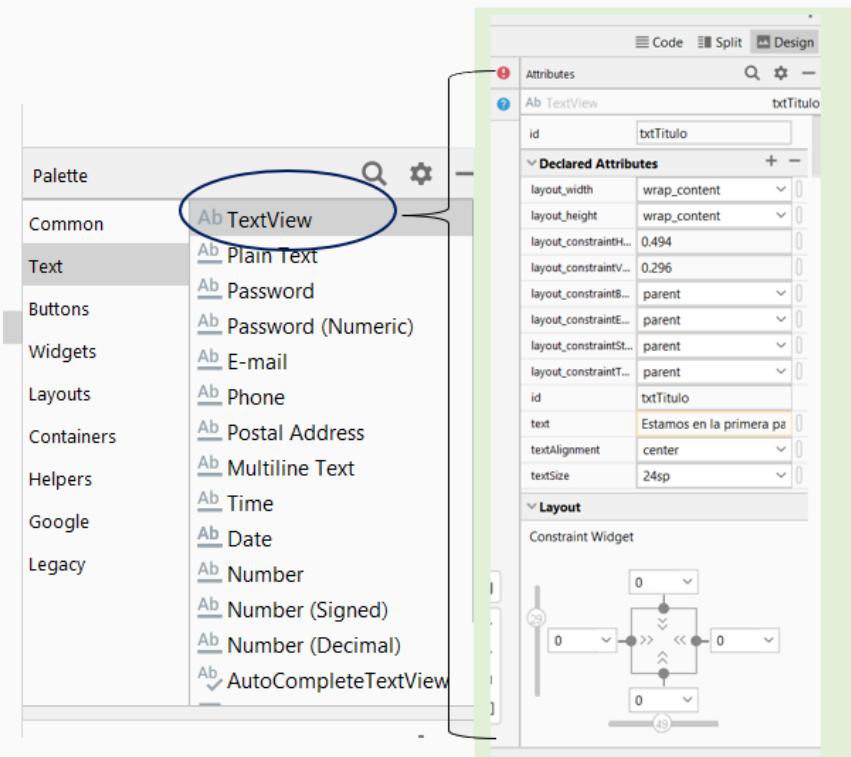
```
<TextView  
    1 → android:id="@+id/viewId"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content" 2  
    3 → android:text="@string/text"  
        android:layout_margin="8dp" 4  
        android:textSize="@dimen/text_size" 5  
    6 } android:textColor="@android:color/holo_purple"  
        android:background="@android:color/holo_orange_dark"  
    />
```

- 1 El identificador no es un *string*, sino que al hacer uso del “@” le estamos indicando al programa que es una referencia a otros componentes. En este caso, estamos referenciando a un “id” llamado *viewId*. A su vez, con el signo +, le estamos diciendo al compilador que, si no llega a encontrar ese nombre (*viewId*) en el listado de **id** que se genera de forma interna, debe crearlo.
- 2 Especifica que el tamaño del componente debe variar según el contenido.
- 3 Debe usar el texto que se encuentra en la referencia string.
- 4 Es el margen o espacio que debe haber entre el `TextView` y el recuadro de la vista.
- 5 Tamaño del texto.
- 6 Las referencias a los colores que queremos usar.



Entorno XML diseño de la Activity

Como en todo IDE que nos facilita la creación de un diseño lo que debemos hacer es trasladar el control a la plantilla de la *activity* y manipular sus propiedades a través de la ventana de “atributos”, encontramos las que visualizamos en la explicación del punto anterior. [¿Cuáles son?](#)





EditText

Los EditText en Android se consideran un TextView que presenta algunas modificaciones. Puede mostrarle un texto al usuario; no obstante, su función es permitirle o brindarle la opción de introducir texto.

Veamos la etiqueta de creación desde código, recordando que si lo creamos desde el diseñador de XML del *activity* debemos cliquear en los atributos que deseamos personalizar.

```
<EditText  
    android:id="@+id/txtUsuario"  
    android:layout_width="345dp"  
    android:layout_height="38dp"  
    android:ems="10"  
    android:inputType="text"  
    app:layout_constraintBottom_toTopOf="@+id(btnUno"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintHorizontal_bias="0.337"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/txtTitulo"  
    app:layout_constraintVertical_bias="0.501" />
```

Para tener en cuenta

- A diferencia del código del textView, los atributos **width y height** tienen valores asignados, la escala es **dp** que es una unidad de píxel virtual que prácticamente equivale a un píxel en una pantalla de densidad media.
- “**ems**” se usa para determinar el tamaño de un carácter según la cantidad de puntos que use la fuente del texto.
- “**inputType = 'text'**” le indica al código que ese EditText va a aceptar todo lo que se considere texto. A su vez, permite que la app le indique al dispositivo del usuario el tipo de caracteres que acepta.

Por ejemplo si el valor del atributo es **phone** le indicamos que el ingreso es un número de teléfono, pero si es **textPassword** el ingreso es una contraseña y como tal debe estar camuflada.

- El resto de los atributos que nos muestra están relacionados a la disposición del EditText con respecto a los otros controles del diseño.



RadioButton (Botones de selección)

Los [botones de selección](#) permiten al usuario seleccionar una opción de un conjunto. A continuación veamos algunas de sus características:

Creación de opciones del Botón de Selección:	<ul style="list-style-type: none">Se selecciona un RadioButton en el diseño.
Exclusividad mutua	<ul style="list-style-type: none">Los botones de selección son mutuamente excluyentes.Se deben agrupar dentro de un RadioGroup para garantizar que solo se pueda seleccionar un botón a la vez.
Eventos de click	<ul style="list-style-type: none">Cuando el usuario selecciona uno de los botones de selección, el objeto RadioButton correspondiente recibe un evento de click.
Controlador de eventos de click	<ul style="list-style-type: none">Se utiliza el atributo <code>android:onClick</code> al elemento <code><RadioButton></code> en el diseño XML.El valor de este atributo debe ser el nombre del método al que se desea llamar en respuesta a un evento de click, similar a los botones regulares.La Activity que aloja el diseño debe implementar el método correspondiente.

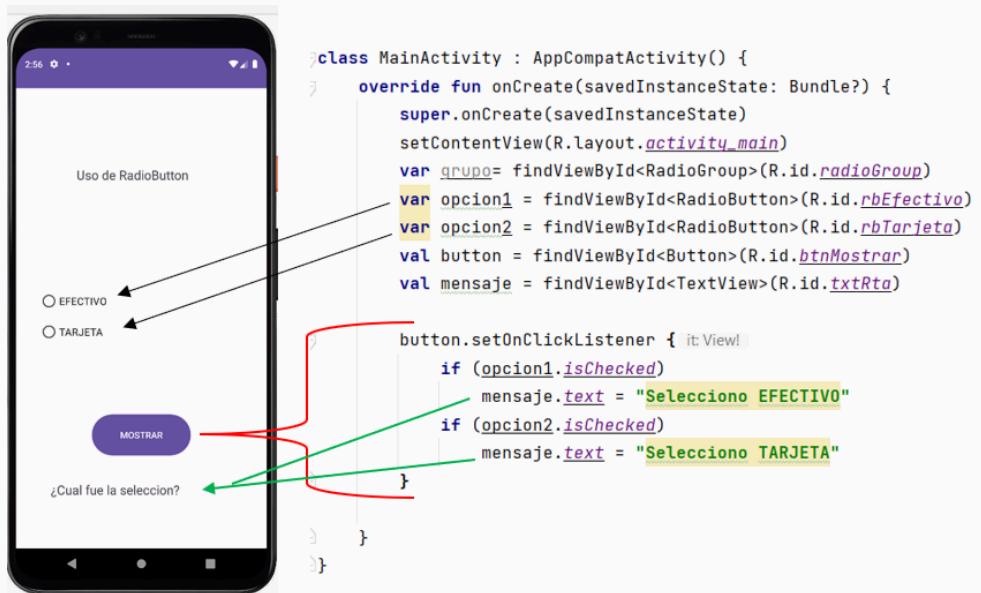
Veamos como los usamos ...



¿Cómo manipulamos al RadioButton?

Lo primero que hacemos es **crear variables asociadas a ellos, personalizándolas** con la asignación del `id` que establecimos. Colocamos en el diseño también un botón para tener control sobre la selección.

Veamos entonces:





CheckBox (Casilla de verificación)

Las **casillas de verificación** permiten que el usuario seleccione una o más opciones de un conjunto. Por lo general, debés presentar cada opción de casilla de verificación en una lista vertical.

Un Checkbox es un botón de dos estados (marcado, no marcado) que actúa como control de selección.

Para manipular los CheckBox usamos la misma lógica empleada en los RadioButton.

Veamos entonces:

```
class MainActivity2 : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main2)  
  
        val check1 = findViewById<CheckBox>(R.id.check1)  
        val check2 = findViewById<CheckBox>(R.id.check2)  
        val button1 = findViewById<Button>(R.id.btnSele)  
        val nota = findViewById<TextView>(R.id.txtNota)  
  
        button1.setOnClickListener { it: View! ->  
            if ((check1.isChecked) && (check2.isChecked))  
                nota.text = "Selecciono LACTEOS"  
  
            if ((check2.isChecked) && (check1.isChecked))  
                nota.text = "Selecciono CARNES"  
            if ((check2.isChecked) && (check1.isChecked))  
                nota.text = "Selecciono LACTEOS Y CARNES"  
        }  
    }  
}
```



Uso de varias pantallas

Anteriormente hablamos de las [activity](#) ¿te acordás?

Dijimos que cuando tenemos una pantalla en el desarrollo móvil, podemos decir que tenemos una *activity*, y que generalmente en una aplicación solemos tener más de una pantalla, es decir, más de una *activity*.

Para incorporar varias pantallas debemos hablar de [Intents](#).

Definición

Los [Intents](#) son objetos que nos ayudan a iniciar una actividad o proceso a través de la configuración con la cual se haya creado, esto nos permite navegar a nuevas pantallas o acceder a funciones extras que no tiene nuestra app, pero si el resto del sistema Android.

Hay 2 tipos de Intents:

- ❖ **Explícitos:** son los que indican qué se desea lanzar exactamente, es el más habitual y se refiere a otra activity. Este se da cuando tengo una app con una pantalla y quiero lanzar la segunda pantalla.
- ❖ **Implícitos:** son los que se utilizan cuando quiero lanzar tareas abstractas, es decir lanzar un servicio del tipo “Quiero sacar una foto”, en este caso no estoy lanzando otra pantalla sino que aparecerá la aplicación de cámara para el dispositivo móvil siempre que el usuario de permiso o acceso a la cámara.

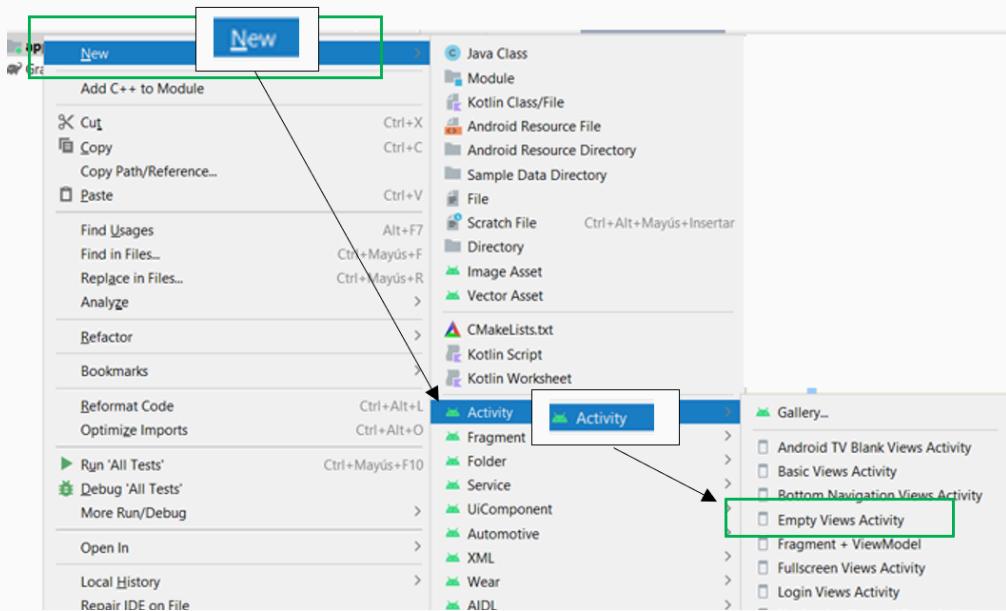
Pero, ¿Cómo hacemos para incorporar más de una *activity* al desarrollo?



Pasos para crear la nueva pantalla

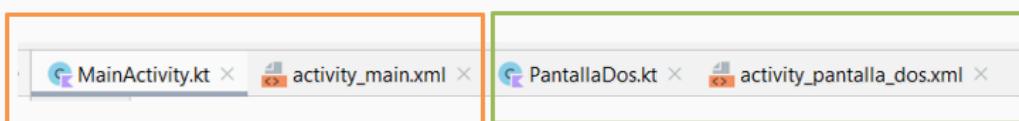
Lo primero es abrir un nuevo proyecto en el IDE, y como estuvimos viendo está compuesto de 2 elementos que son el archivo .kt y el archivo .xml (siempre que trabajemos con la plantilla [empty views activity](#))

Ya estamos listo para agregar la segunda *activity* ...



Seguimos todas las instrucciones que nos pide las ventanas con respecto al nombre, ubicación etc. Para esta explicación llamamos a la segunda *activity* [PantallaDos.kt](#).

El IDE nos muestra las siguientes solapas, las dos primeras corresponden a la *activity* inicial y las dos segundas a la *activity* que agregamos.



¿Cuáles son las líneas de código que nos permite pasar a la nueva *activity*?



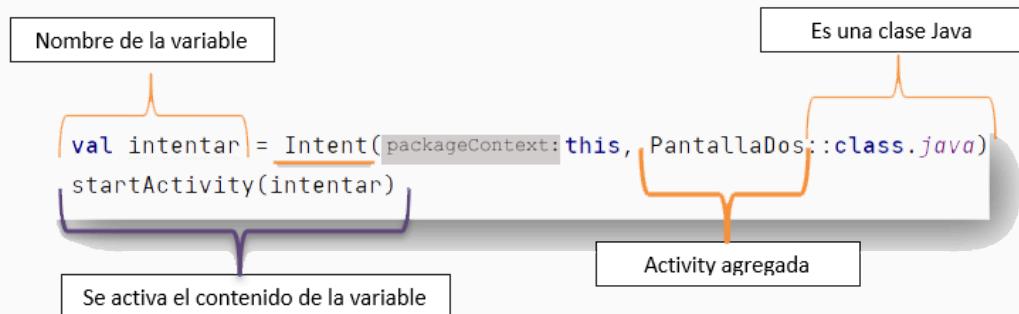
Código para pasar a la nueva pantalla

Aquí es donde usamos el Intent, veamos que tiene que tener la *activity*:

1. El **import** que haga referencia a este objeto (al comienzo).

```
import android.content.Intent
```

2. **Código:** En general estas líneas se ubican en un Button que tiene asignado el evento **setOnClickListener{ }**





Vínculo con base de datos

En esta tema te proponemos un viaje fascinante por el mundo de las [bases de datos con Android Studio](#). Imaginemos que nuestra aplicación es como una casa, y la base de datos es el armazón que sostiene y organiza todos los elementos que conforman nuestra experiencia digital.

Vamos a explorar [cómo almacenar y gestionar datos de manera eficiente](#), permitiendo que nuestros usuarios, como invitados en esta casa, tengan una experiencia más personalizada y duradera. Pero, ¿cómo creamos un sistema que permita a diferentes usuarios ingresar a nuestra aplicación?

Aquí es donde entra la magia de las bases de datos. No solo [aprenderemos a usar SQLite](#), un sistema integrado en Android Studio, sino que también descubriremos cómo conectarnos a bases de datos externas como [MySQL](#), brindándonos flexibilidad y capacidad de actualización constante.

Preguntas clave de este tema:

- La información es su materia prima.
- ¿Cuál es la importancia de la persistencia de datos en una aplicación Android?
- ¿Qué base de datos utiliza Android Studio de forma predeterminada?
- ¿Cómo se estructura una base de datos relacional?
- ¿Cuál es la función de SQLiteOpenHelper en la creación y gestión de bases de datos SQLite?
- ¿Cómo se realiza la modificación de datos en una base de datos?

¡Empecemos!

Android y base de datos

¿Cómo podemos considerar cualquier usuario con su contraseña?

Para ello necesitamos tener almacenados los datos en una estructura externa que perdure en el tiempo, y que pueda ser actualizada. Esta estructura es la [base de datos](#) que nos permite crear una app dinámica y totalmente actualizada.

¿Qué base de datos usa Android Studio?

Podemos utilizar un sistema embebido como SQLite o conectar de diversas formas a una base de datos externa, como MySQL.

Veamos de qué se trata.



Estructura de la base de datos

La **persistencia de datos** (almacenar o conservar datos en el dispositivo) es una parte importante del desarrollo de Android. Los datos persistentes garantizan que no se pierda el contenido generado por usuarios cuando se cierra la app.

Con el **objetivo** de conservar datos en las apps para Android, el SDK de Android ofrece SQLite. SQLite proporciona una base de datos relacional que te permite representar datos de forma similar a como estructura los datos con clases de Kotlin.

Es un concepto que has visto en asignaturas anteriores. Una base de datos relacional funciona de la siguiente manera:

- ❖ Las tablas definen agrupaciones de datos de alto nivel que querés representar.
- ❖ Las columnas definen los datos que contiene cada fila de la tabla.
- ❖ Las filas contienen los datos en sí que constan de valores para cada columna de la tabla.

La estructura de una base de datos relacional también refleja lo que ya sabés sobre clases y objetos en Kotlin.

```
data class Estudiante(  
    id: Int,  
    nombre: String,  
    apellido: String,  
    nacionalidad: String  
)
```

- ❖ Las clases, como las tablas, modelan los datos que deseáis representar en tu app.
- ❖ Como las columnas, las propiedades definen los datos específicos que debe contener cada instancia de la clase.
- ❖ Los objetos, como las filas, son los datos reales. Los objetos contienen valores para cada propiedad definida en la clase, al igual que las filas contienen valores para cada columna definida en la tabla de datos.

Recordá que cada tabla de una base de datos relacional contiene un identificador único para las filas. Este identificador se conoce como la clave primaria.

Cuando una tabla hace referencia a la clave primaria de otra tabla, se conoce como clave externa. La presencia de una clave externa significa que existe una relación entre las tablas.

Nota

- Al igual que con las clases de Kotlin, la convención es usar la forma singular para el nombre de las tablas de la base de datos.



SQLite

Definición

SQLite es una base de datos relacional de código abierto que se utiliza para realizar operaciones de bases de datos en dispositivos Android, como almacenar, manipular o recuperar datos persistentes de la base de datos; hace referencia a una biblioteca C liviana para la administración de bases de datos relacionales con el lenguaje de consulta estructurado SQL.

De forma predeterminada, **la base de datos SQLite está integrada en Android**. Por lo tanto, no es necesario realizar ninguna tarea de administración o configuración de la base de datos.

La clase **SQLiteOpenHelper** proporciona la funcionalidad para utilizar la base de datos SQLite .

La clase **android.database.sqlite.SQLiteOpenHelper** se utiliza para la creación de bases de datos y la gestión de versiones. Para realizar cualquier operación de base de datos, debe proporcionar la implementación de los métodos `onCreate()` y `onUpgrade()` de la clase `SQLiteOpenHelper`.



Constructor y métodos de la clase SQLiteOpenHelper

Constructores:

- Si queremos crear un objeto de SQLiteOpenHelper para crear, abrir y administrar la base de datos.

```
SQLiteOpenHelper(context: Context?, name: String?,  
factory: SQLiteDatabase.CursorFactory?, version: Int)
```

Parámetros del constructor de SQLiteOpenHelper

1. context: Context? → Opcional (puede ser nulo)

- Representa el contexto de la aplicación.
- Se usa para acceder a los recursos del sistema y abrir la base de datos.
- Recomendado:** No debería ser `null`, ya que se necesita para operar con la base de datos.

2. name: String? → Opcional (puede ser nulo)

- Nombre del archivo donde se almacenará la base de datos.
- Si es `null`, la base de datos se creará en memoria (solo existirá mientras la app esté abierta).

3. factory: SQLiteDatabase.CursorFactory? → Opcional (puede ser nulo)

- Permite definir una fábrica de cursos personalizados para consultas en la base de datos.
- Si es `null`, se usa el manejador de cursos predeterminado de SQLite.
- Normalmente se deja en `null` a menos que se necesite personalizar el comportamiento de los cursos.

4. version: Int → Obligatorio (no puede ser nulo)

- Número de versión de la base de datos.
- Se usa para manejar actualizaciones y cambios en la estructura de la base de datos.
- Debe ser mayor a 0.** Si cambia, se ejecutará `onUpgrade()` para actualizar la estructura.



Métodos de la clase SQLiteDatabase

La clase **SQLiteDatabase** contiene métodos que se realizarán en la base de datos SQLite, como crear, actualizar, eliminar, seleccionar, etc.

- 1 Ejecuta la consulta SQL, no una consulta de selección.
- 2 Inserta un registro en la base de datos. La tabla especifica el nombre de la tabla, nullColumnHack no permite valores completamente nulos. Si el segundo argumento es nulo, Android almacenará valores nulos si los valores están vacíos. El tercer argumento especifica los valores que se almacenarán.
- 3 Actualiza una fila.
- 4 Devuelve un cursor sobre el conjunto de resultados.

```
// 1. Ejecutar una consulta SQL (no de selección)
db.execSQL(sql: "DELETE FROM usuarios WHERE id = 1")

// 2. Insertar un registro
val valores = ContentValues().apply {
    put("nombre", "Juan")
    put("edad", 30)
}
db.insert(table: "usuarios", nullColumnHack: null, valores)

// 3. Actualizar una fila
val valoresActualizados = ContentValues().apply {
    put("edad", 31)
}
db.update(table: "usuarios", valoresActualizados, whereClause: "id = ?", arrayOf("1"))

// 4. Seleccionar datos (devuelve un cursor)
val cursor = db.rawQuery(sql: "SELECT * FROM usuarios", selectionArgs: null)
cursor.close() // Cerrar el cursor después de usarlo
```



Ejemplo con base de datos

El ejemplo que proponemos para la explicación crea solamente una tabla de usuario cuyos atributos son el nombre de ese usuario y su clave o contraseña.

Vamos a mostrar cómo crear la tabla y vincularla a la [clase](#) correspondiente (creada por nosotros en el IDE de programación) para tomar los datos desde el teclado del usuario de la App.

También te mostraremos la sintaxis necesaria para el famoso ABM de programación que es, [Alta](#), [Baja](#), [Modificación](#).

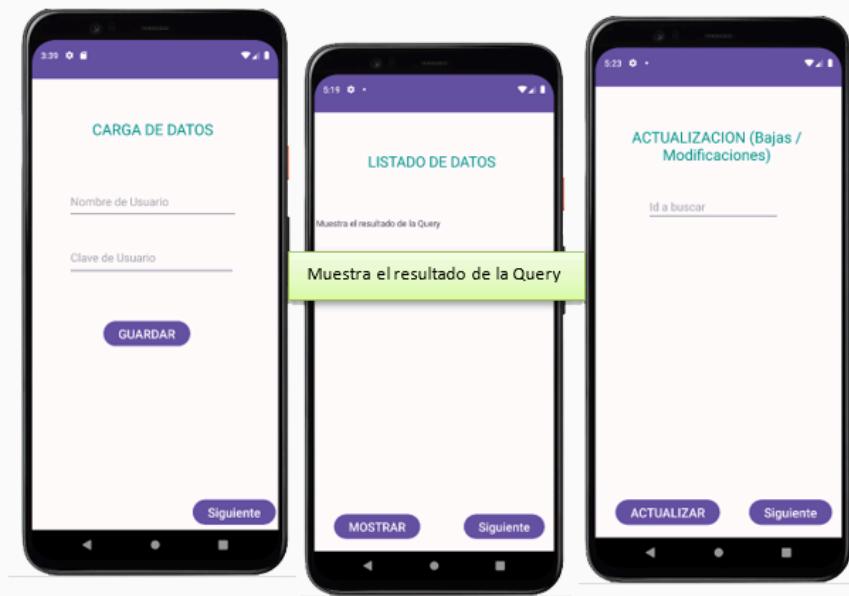
Tenemos que crear la [clase base de datos](#) donde escribiremos el código para manipular los datos.

Luego desde el [MainActivity.kt](#) con los componentes que tenga nuestro diseño usaremos las clases mencionadas.

¡Manos a la obra!



Diseño del ejemplo



La lógica es la siguiente, primero se debe cargar la tabla con el nombre y la contraseña del usuario, ya que consideramos un **Id** como *primary key* de tipo auto_incrementable.

Una vez cargados los datos podemos ver la carga mostrando el contenido total de la tabla.

En la actualización se debe ingresar el **id** que queremos eliminar o el que corresponda a la fila que queremos modificar.

Nota

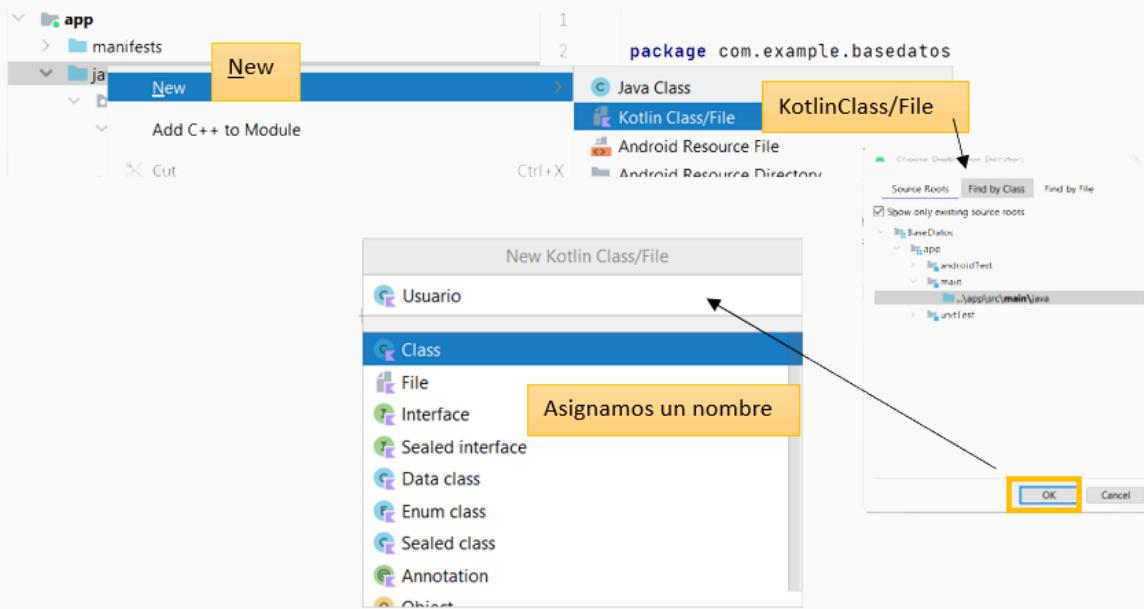
- Recordá que en cualquiera de las tres operaciones de ABM es conveniente mostrar algún mensaje que pregunte si son correctos los datos ingresados.



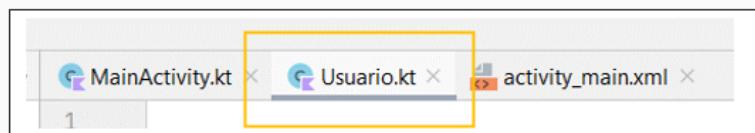
Código de creación de los datos

Como mencionamos al inicio de la explicación del ejemplo.

Después de crear un [nuevo](#) proyecto, creamos la [clase](#) que nos permite cargar cada una de las tablas de la base de datos, para ello debemos [agregarla](#) al proyecto. ¿Cómo lo hacemos?



Como la tabla de nuestra base de datos trata sobre los usuarios, nuestra clase es [Usuario](#), nos quedan las pestañas del IDE de la siguiente forma:



Creamos el archivo de Clase para la base de datos (igual que te mostramos en la imagen anterior).

Usamos el [SQLiteOpenHelper](#), podemos crear una variable con el nombre de la base para incluirlo en la escritura de la sentencia. Completamos todos los parámetros. A continuación debemos implementar los miembros de la clase. ¿Cómo lo hacemos?

Seguramente nos aparece marcada como error la palabra reservada `Class` y el nombre que le dimos y al costado izquierdo la lámpara que automáticamente coloca el IDE, al hacer click en ella debemos tener como opción [implementar miembros](#).

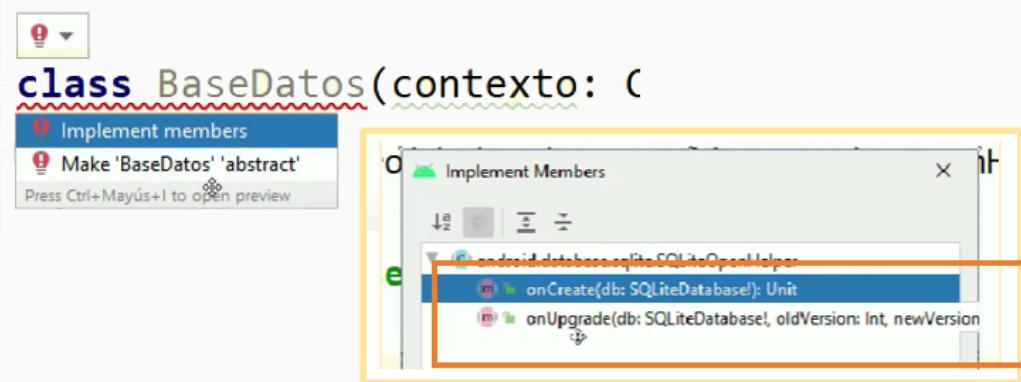
De esta forma creamos el [override](#) del [onCreate](#) y del [onUpgrade](#). Implementamos los dos.

¡Veamos!



Override del onCréate y del onUpgrade

¿Como los creamos?



Quedando la clase de esta forma, observá que se llamó a la base [BaseDatos](#).

```
var BD = "BaseDatos"

class BDatos(contexto: Context): SQLiteOpenHelper(contexto, BD, factory: null, version: 1) {
    override fun onCreate(p0: SQLiteDatabase?) {
        TODO( reason: "Not yet implemented")
    }

    override fun onUpgrade(p0: SQLiteDatabase?, p1: Int, p2: Int) {
        TODO( reason: "Not yet implemented")
    }
}
```

Ya tenemos establecido en el entorno de nuestro celular que vamos a crear una base de datos con las tablas que la forman.

Usamos OnCreate para crear las tablas, contamos con una variable que almacena nuestra Query y luego usamos el [execSQL](#) para que se ejecute la consulta formulada.

```
class BDatos(contexto: Context): SQLiteOpenHelper(contexto, BD, factory: null, version: 1) {
    override fun onCreate(p0: SQLiteDatabase?) {
        var sql = "create table Usuario" +
            "(id INTEGER primary key autoincrement, nombre VARCHAR(100), clave INTEGER)"
        p0?.execSQL(sql)
    }
}
```



Código para alta del ABM

Creamos una función para insertar los dominios. A esta función le agregamos un retorno tipo texto para que nos indique si el *insert* fue exitoso o no.

```
fun insertarDatos(usuario: Usuario):String{
    val p0 = this.writableDatabase
    var contenedor = ContentValues()

    contenedor.put("nombre",usuario.nombre)
    contenedor.put("clave",usuario.clave)

    var resultado= p0.insert( table: "Usuario ", nullColumnHack: null,contenedor)
    if(resultado == -1.toLong()){
        return "Falla en la carga de datos"
    } else{
        return "Insert exitoso"
    }

}
```

Vamos a analizar el código:

Tomamos la referencia a la base (en este ejemplo `p0`) y especificamos que será de escritura, luego en otra variable creamos un “contenedor” de valores para tomar los ingresos a cada atributo de la tabla. Observá que usamos el constructor de la clase que creamos. Por último, para saber si el *insert* se ejecutó bien la variable `resultado` almacená el resultado de `p0.insert`.

Nota

Cuando estés escribiendo una sentencia como en este caso, la de `p0.insert`, las palabras que ves en gris claro **no debés escribirlas**, se completan solas, quiere decir que el entorno reconoce los miembros de la clase.



Código para mostrar los datos

Cuando queremos mostrar el contenido de la base de datos, escribimos una Query tan simple o compleja según sea el requerimiento de visualización.

Hasta el momento vimos que trabajamos con la variable que tenemos vinculada nuestra base, en toda la explicación en [p0](#) y, como es de suponer, accedemos a ella en modo [lectura](#).

El resultado de la consulta es almacenado en un [cursor](#), ¿recordás el término de base de datos?, lo usaste en [procedures](#) y/o [triggers](#)

Este cursor contendrá las filas resultantes y tendrá la cantidad de columnas según sean los atributos proyectados en el [select](#) de la Query. Tendremos que posicionarnos en la primera fila y pasar una a una hasta el final para tomar cada dato que nos trae.

Veamos entonces cómo declaramos esas variables, dónde las guardamos para darle salida al exterior.

Creamos una función [traerDatos](#) que nos devuelve una lista tipo [MutableList](#) que contiene la clase [Usuario](#).

```
fun traerDatos(): MutableList<Usuario> {
    val p0 = this.readableDatabase
    var lista: MutableList<Usuario> = ArrayList()
    val sql = "select * from usuario"
    val resultado = p0.rawQuery(sql, selectionArgs: null)

    if (resultado.moveToFirst()) {
        do {
            var usu = Usuario()
            usu.id = resultado.getString(resultado.getColumnIndex( columnName: "id")).toInt()
            usu.nombre = resultado.getString(resultado.getColumnIndex( columnName: "nombre")).toString()
            usu.clave = resultado.getString(resultado.getColumnIndex( columnName: "clave")).toInt()

            lista.add(usu)
        } while (resultado.moveToNext())
    }
    p0.close()
    resultado.close()
}

return(lista)
}
```



A. Estas variables hacen referencia:

- `p0` al modo de uso de la base de datos [lectura](#),
- `lista` es la salida de la función de tipo [array](#),
- `sql` la consulta que vamos a ejecutar para recuperar los datos,
- `resultado` representa a nuestro [cursor](#) que está relacionado a la query.

B. Con el `if` nos aseguramos de estar en la primera fila:

- `usu` representa a la clase [Usuario](#),
- en cada atributo de la clase almacenamos el contenido de la columna del cursor (se lo puede referenciar también por la posición recordando que comienza de 0),
- lo cargamos al [array](#) con el método [add](#),
- con [moveToNext](#) pasamos de fila en fila hasta el final,

- cerramos la base de datos y el cursor.
- Finalmente retornamos la [lista](#).



Código para modificación y baja del ABM

Cuando queremos [actualizar \(modificar\)](#) algún dato de la base debemos identificar la tabla y la fila a modificar, salvo que se cambie el contenido de toda la columna de una tabla.

Si modificamos alguna fila en particular no debemos olvidar el valor que tiene la *primary key*.

Como venimos haciendo, construimos una función, la base debe ser accedida de tipo escritura ya que se modifican todos o alguno de los contenidos de la fila/s.

Pasemos a analizar entonces el código.

```
fun actualizaDatos(id:String,nombre:String,clave:Int):String{
    val p0 = this.writableDatabase
    var contenedor = ContentValues()

    contenedor.put("nombre",nombre)
    contenedor.put("clave",clave)

    var resultado= p0.update( table: "usuario",contenedor, whereClause: "id=?", arrayOf(id))
    if(resultado == 0){
        return "No se pudo actualizar la tabla"
    } else {
        return "Actualizacion exitosa"
    }
} // fin actualizarDatos
```

Para borrar una fila de la tabla debemos usar la *primary key*.

Ya estamos familiarizados con las líneas de código ¿verdad?, veamos entonces cómo escribir el borrado.

```
fun borrarDatos(codigo:String){
    if(codigo.length > 0){
        val p0 = this.writableDatabase
        p0.delete( table: "Usuario", whereClause: "id=?", arrayOf(codigo))
        /* si queremos borrar todos
        pB.delete("Usuario",null,null)
        */
        p0.close()
    }
}// fin borrarDatos
```



Preparar el entorno del proyecto en Android Studio

A lo largo de este tema, nos adentraremos en la transformación de proyectos previos, llevando los conocimientos adquiridos con [C#](#) a la práctica con [Kotlin](#).

En esta oportunidad, nos sumergiremos en el análisis documentado, rescatando la esencia de nuestras bases de datos y plantillas de casos de uso.

La misión es clara: [convertir ese valioso diseño en una aplicación vibrante y funcional](#).

Comenzaremos explorando cómo visualizar nuestras tablas en el IDE de Android, desentrañando las herramientas del App Inspection para analizar datos en tiempo real. Nos adentraremos en la creación de bases de datos con múltiples tablas, comprendiendo el poder del SQLiteOpenHelper y las mejores prácticas en la escritura de códigos.

Además, enfrentaremos el reto de cambiar la Activity de inicio y exploraremos cómo el cambio de IDE implica repensar a nuestros usuarios. Con consejos prácticos y la creación de la base de datos para el proyecto "Club Deportivo", esta semana nos sumergiremos en la realidad de programar una aplicación desde cero.

Preguntas clave de este tema:

- ¿Cómo podemos visualizar las tablas creadas en la base de datos SQLite dentro del IDE de Android?
- ¿Cuál es el propósito de usar el App Inspection y qué nos permite hacer?
- ¿Cómo debemos escribir el código cuando nuestra base de datos tiene más de una tabla?
- ¿Qué significa el uso de "companion object" en Kotlin y cómo se relaciona con las constantes en las Querys?
- ¿Cómo podemos cambiar la Activity de inicio en Android Studio?

¿Listo/a para empezar? ¡Adelante, el mundo del desarrollo te espera!



Introducción al entorno del proyecto

Hemos dado hasta el momento los primeros pasos en la programación de App utilizando Kotlin. Seguramente te preguntarás ¿qué haremos a partir de ahora?

Nuestra propuesta es transformar el proyecto de la materia *Desarrollo de sistemas orientado a objetos* en una App

Tendrás los [conceptos y tips](#) necesarios para codificar en Kotlin lo que codificaste en C#

¿Qué deberemos hacer?

Tendremos que recuperar el *Análisis* documentado, teniendo en cuenta que el diseño de la base de datos y las plantillas de caso de uso se convertirán en las estrellas del resto de las semanas.

Si pensamos en el desarrollo y las actividades que se deben resolver, podemos decir que la primera y fundamental es la comprensión del objetivo.

Teniéndolo claro, podemos dividir nuestras actividades en:

- ❖ **Análisis de los datos**, es decir construcción de la base de datos.
- ❖ **Programación de las acciones** que toman los datos y las convierte en salidas que cumplan con el objetivo.

¡Comencemos!



Cómo visualizar la Base de datos en el IDE

Para visualizar la/s tabla/s creadas en la base de datos en SQLite dentro del IDE de Android debés acceder al [App Inspection](#)



Si no encontrás al [App Inspection](#) en la base del IDE a la izquierda podemos desplegar un menú donde aparece el App Inspection, una vez seleccionado en el resto de los proyectos aparecerá en la barra inferior.

Al hacer click abre una ventana que muestra la base de datos del proyecto y la lista de tablas al seleccionar la que deseamos consultar, accedemos a los dominios ingresados.

Esta herramienta es utilizada para la visualización y análisis de datos en tiempo de ejecución, y no está diseñada para realizar operaciones de actualización directa en la base de datos.



Creación de la Base con varias tablas

Los datos almacenados en la Base de datos están distribuidos en varias tablas vinculadas a través de relaciones respetando las reglas de normalización.

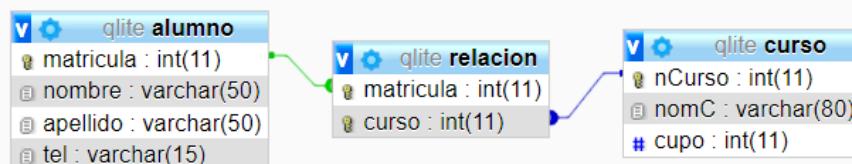
¿Cómo debemos escribir el código?

Sabemos que nuestra app debe tener una clase dedicada a la estructura de la base y además implementamos el onCreate y el onUpgrade de SQLiteOpenHelper.

Como todo lenguaje de programación existen comunidades o reglas a la hora de escribir código que permite que otros/as desarrolladores/as puedan comprender su sintaxis y lógica.

Te mostramos como escribir las Querys cuando la base tiene más de una tabla.

Este ejemplo corresponde al siguiente modelo:



Ya creada la clase destinada a la Base de datos, creamos el onCreate y el onUpgrade.

```
package com.example.nueva_dos  
Nombre del proyecto → nueva_dos  
  
import android.annotation.SuppressLint  
import android.content.ContentValues  
import android.content.Context  
import android.database.sqlite.SQLiteOpenHelper  
import android.database.sqlite.SQLiteDatabase  
  
class Datos(contexto:Context):SQLiteOpenHelper(contexto, name: "BaseDos.db", factory: null, version: 2) {  
    override fun onCreate(db: SQLiteDatabase?) {  
        // Creacion de las tablas  
        db?.execSQL(CREATE_ALUMNO_TABLE)  
        db?.execSQL(CREATE_CURSO_TABLE)  
        db?.execSQL(CREATE_RELACION_TABLE)  
    }  
  
    override fun onUpgrade(db: SQLiteDatabase?, p1: Int, p2: In  
        // se usa para las actualizaciones de las tablas  
        db?.execSQL( sql: "DROP TABLE IF EXISTS alumno")  
        db?.execSQL( sql: "DROP TABLE IF EXISTS curso")  
        db?.execSQL( sql: "DROP TABLE IF EXISTS relacion")  
        onCreate(db)  
    }  
}
```

En cada execSQL encontramos como argumento una constante

Invoca antes de finalizar a la función onCreate()

Si observamos hay un db?.execSQL para cada tabla del modelo, y en lugar de colocar la sintaxis de la query de creación de la tabla nos encontramos con constantes que pasaremos a explicar.

```
// COMPANION OBJECT define miembros estáticos en una clase.
companion object {

    private const val CREATE_ALUMNO_TABLE = "CREATE TABLE alumno " +
        "(matricula INTEGER PRIMARY KEY AUTOINCREMENT, nombre TEXT, " +
        " apellido TEXT, tel INTEGER)"

    private const val CREATE_CURSO_TABLE = "CREATE TABLE curso " +
        "(nroC INTEGER PRIMARY KEY AUTOINCREMENT, nombreCur TEXT, cupo INTEGER)"

    private const val CREATE_RELACION_TABLE = "CREATE TABLE relacion " +
        "(matricula INTEGER, curso INTEGER, fecha TEXT, " +
        " PRIMARY KEY (matricula, curso), " +
        " foreign key(matricula) references alumno(matricula), " +
        " foreign key(curso) references curso(nroC))"
}
```

Estas constantes están dentro de `companion object {}` pero ¿qué significa?

***Companion object* en Kotlin**

es una característica que te permite definir miembros estáticos en una clase. Los miembros definidos dentro de un *companion object* son similares a los miembros de una clase estática en otros lenguajes de programación.

Companion object también puede implementar interfaces, extender otras clases, y contener propiedades y funciones como cualquier otro objeto. Es especialmente útil cuando necesitas definir miembros que deben ser compartidos por todas las instancias de una clase o cuando necesitas proporcionar un espacio de nombres para ciertos elementos relacionados con la clase.

Observá que el nombre asignado a cada constante es el argumento de cada execSQL.

```

package com.example.nueva_dos

import android.annotation.SuppressLint
import android.content.ContentValues
import android.content.Context
import android.database.sqlite.SQLiteOpenHelper
import android.database.sqlite.SQLiteDatabase

class Datos(contexto:Context):SQLiteOpenHelper(contexto,"BaseDos.db",null,2) {
    override fun onCreate(db: SQLiteDatabase?) {
        // Creacion de las tablas
        db?.execSQL(CREATE_ALUMNO_TABLE)
        db?.execSQL(CREATE_CURSO_TABLE)
        db?.execSQL(CREATE_RELACION_TABLE)
    }

    override fun onUpgrade(db: SQLiteDatabase?, p1: Int, p2: Int) {
        // se usa para las actualizaciones de las tablas
        db?.execSQL("DROP TABLE IF EXISTS alumno")
        db?.execSQL("DROP TABLE IF EXISTS curso")
        db?.execSQL("DROP TABLE IF EXISTS relacion")
        onCreate(db)
    }

    // COMPANION OBJECT define miembros estáticos en una clase.
    companion object {

        private const val CREATE_ALUMNO_TABLE = "CREATE TABLE alumno " +
            "(matricula INTEGER PRIMARY KEY AUTOINCREMENT, nombre TEXT, " +
            " apellido TEXT, tel INTEGER)"

        private const val CREATE_CURSO_TABLE = "CREATE TABLE curso " +
            "(nroC INTEGER PRIMARY KEY AUTOINCREMENT, nombreCur TEXT, cupo INTEGER)"

        private const val CREATE_RELACION_TABLE = "CREATE TABLE relacion " +
            "(matricula INTEGER, curso INTEGER, fecha TEXT, " +
            " PRIMARY KEY (matricula, curso)," +
            " foreign key(matricula) references alumno(matricula), " +
            " foreign key(curso) references curso(nroC)"
    }
}

```

[¿Cómo procedemos luego de estas líneas de código?](#)

Igual que lo que estudiamos en la semana 7, es decir, cada tabla tiene su clase correspondiente y en la clase de la base de datos codificamos las funciones que nos permitan actualizar (*insert*, *update*, *delete*) los dominios de las tablas. Luego serán invocadas dichas funciones en el MainActivity que corresponda.



Comenzando con la App del proyecto “Club deportivo”

La mejor manera de aprender un nuevo lenguaje de programación junto a su entorno es tener un objetivo que cumplir, y si ese objetivo ya fue desarrollado nos permite enfocarnos en la sintaxis propiamente dicha despreocupándonos del razonamiento lógico.

Vamos a transformar el proyecto de la materia *Desarrollo de sistemas orientado a objetos* en una *App*.

¿Migro la sintaxis de cada línea codificada en C# a Kotlin?, ¿Mantengo el mismo diseño de las pantallas?, ¿El acceso a la aplicación es igual?

Estas preguntas pueden surgir de manera espontánea, todas sus respuestas pueden ser afirmativas para un/a estudiante y negativas para otro/a, esto sucede porque **están íntimamente ligadas a la lógica de razonamiento del/dla diseñador/a y programador/a.**

- 1 **Lo primero que debemos hacer** es armar un croquis de la cantidad de pantallas (Activity's) que tendrá nuestra App, y cuales son las acciones que nos permiten interactuar entre ellas.
- 2 **Lo segundo es** determinar el usuario que identificamos en el desarrollo en C#, si es el mismo en la App de Kotlin. En este caso trabajaremos únicamente en la app para el personal administrativo.
- 3 **Y, lo tercero es** crear las clases necesarias para tener la base de datos a punto en SQLite para comenzar con la codificación.



Primeros tips para el desarrollo de la App

Tips

En programación generalmente se refiere a [consejos, sugerencias o trucos](#) que los programadores comparten para mejorar la eficiencia, la calidad del código, o para evitar errores comunes.

Estos tips [pueden abordar aspectos específicos del lenguaje de programación](#), las herramientas de desarrollo, las prácticas de codificación o la gestión de proyectos.

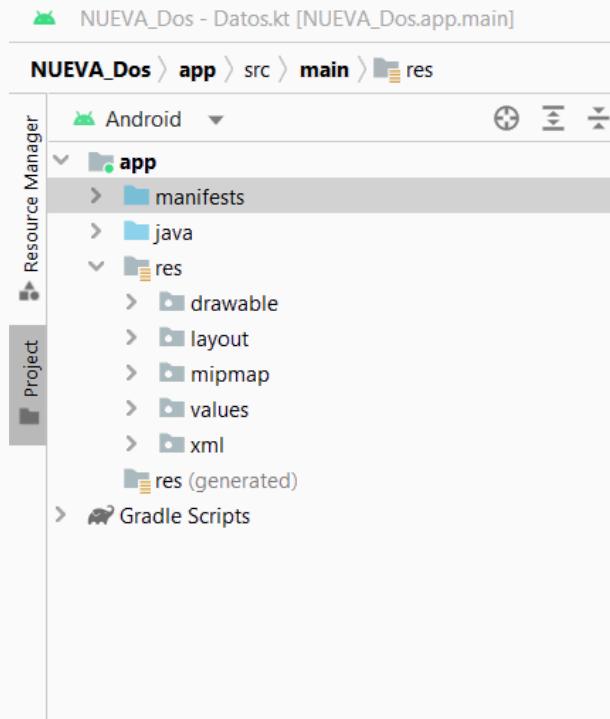


Cambio de Activity de inicio

Cuando construimos un desarrollo que involucra diferentes pantallas, comenzamos muchas veces con la codificación de algún módulo y luego a medida que avanzamos y encontramos la funcionalidad y el mejor acceso para el usuario cambiamos la *pantalla inicial*.

¿Cómo lo hacemos en este IDE?

Debemos identificar en el árbol del proyecto el archivo **Manifests**.



Dentro del contenido de este archivo *AndroidManifest.xml* nos muestra la cantidad de Activity que contiene el proyecto. Siguiendo el ejemplo mostrado en el modelo de datos de esta semana tenemos las siguientes Activity para cargar datos y mostrar a los *Alumnos, Cursos y su relación*.

```
        <activity
            android:name=".MainMostrar"
            android:exported="false" />
        <activity
            android:name=".MainRelacion"
            android:exported="false" />
        <activity
            android:name=".MainCurso"
            android:exported="false" />
        <activity
            android:name=".MainAlumno"
            android:exported="false" />
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Vemos que la llamada MainActivity tiene el bloque `<intent-filter>` este bloque identifica cual es la actividad de inicio. Simplemente cambia el nombre de la activity que tiene el bloque `<intent-filter>` por la que deseas iniciar.

Después de hacer estos cambios, debes guardar el archivo AndroidManifest.xml y vuelve a compilar y ejecutar la aplicación. La nueva actividad que has configurado debería ser ahora la actividad de inicio.



Nuevo IDE nuevos usuarios

Tal vez te preguntes [por qué decimos nuevo IDE nuevos usuarios](#).

Cuando desarrollaste el sistema de Club Deportivo en C# el usuario fue el [empleado del gimnasio](#) que registraba las actividades del socio cumpliendo los requerimientos del objetivo de ese sistema.

Entre las acciones codificadas existe un módulo que generó la impresión del carnet del socio, ¿verdad? Ahora todo este desarrollo se convierte en una App y si manejas un modulo de usuario y contraseña podes internamente dividir la funcionalidad de los procesos para dos usuarios uno de ellos sigue siendo el empleado del gimnasio y el otro el [socio](#).

Incorporar al socio permite que su carnet lo pueda visualizar de forma directa sin necesidad de tener la impresión como en el desarrollo anterior.

Observa que si incorporas este [tip](#) deberías agregar a la tabla socio una contraseña para el ingreso a la App utilizando como usuario el dato que es único como su documento de identificación o si es de tu agrado agregarle un alias, pero atención a la hora de ingresar ese alias ningún otro socio lo debe tener.

A medida que avancemos en el desarrollo, iremos agregando otros tips.