# Kotlin.pdf

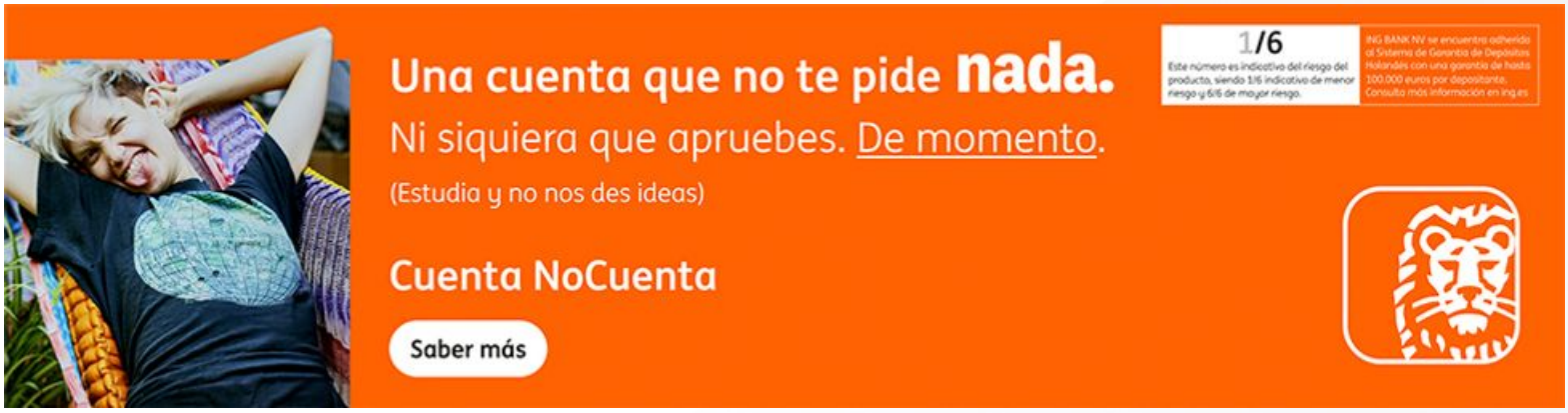
quiico_

Programación multimedia y dispositivos móviles

1º Desarrollo de Aplicaciones Multiplataforma

Estudios España

Accede al documento original




Reservados todos los derechos.
**No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.**

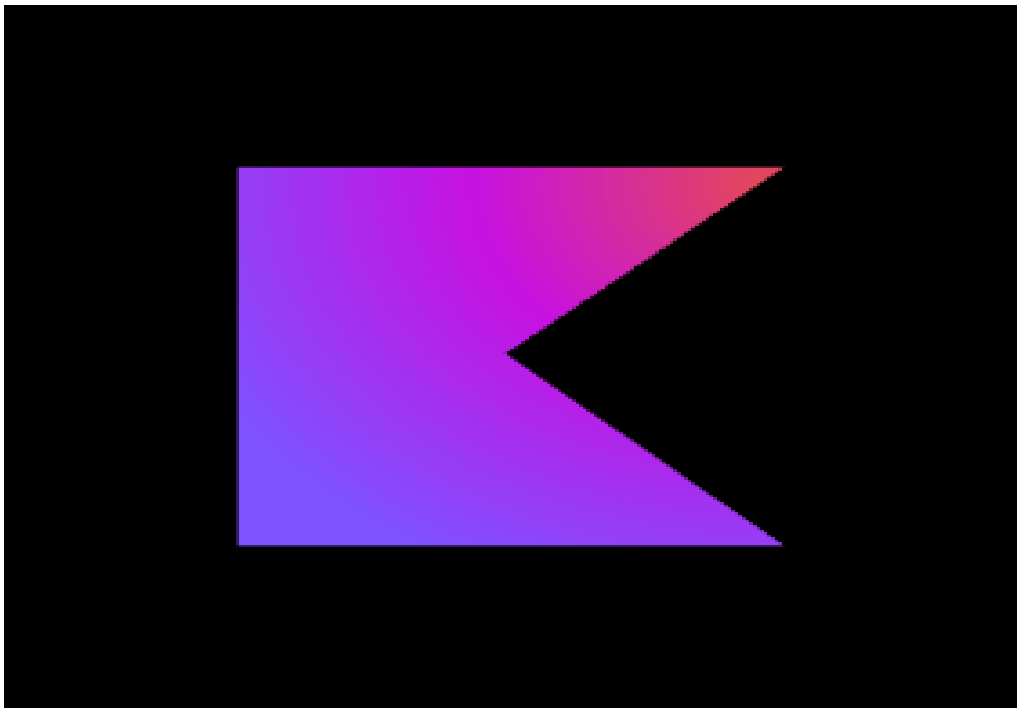
CARPE DIEM
Organiza tu futuro:
estudia hoy para destacar mañana.
En Carpe Diem te esperan cursos adaptados a ti.
Una forma fácil y real de avanzar profesionalmente.
¡Quiero formarme!
Acreditados
Económicos
Desde casa
Elige tu curso

# Kotlin

2º DAM  Curso 2024/2025

**Asignatura:** Programación Multimedia y Dispositivos Móviles

- **Profesor:** Fran López
- **Alumno:** Francisco José Alonso de Caso Ortiz Nº2



WUOLAH

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

# Índice





Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

WUOLAH

Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

**Leyenda**

…. Quiere decir que son palabras textuales del profesor o que es información que él ha subido.
Si está subrayado el título de una sección o una subsección quiere decir que todo lo que compete a esa sección o subsección hasta el siguiente encabezado ha sido proporcionado por el profesor. → 100% Mirar para el examen

…. Puede caer en el examen/ Definiciones.

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

CARPE DIEM
Organiza tu futuro:
estudia hoy para destacar mañana.
En Carpe Diem te esperan cursos adaptados a ti.
Una forma fácil y real de avanzar profesionalmente.
¡Quiero formarme!
Acreditados
Económicos
Desde casa
Elige tu curso



# Arquitectura

## Patrón Repositorio:

El patrón de repositorio es una forma de organizar la capa de acceso a datos en la arquitectura de una aplicación. Se basa en la idea de crear una interfaz de repositorio para cada grupo funcional de la aplicación y definir las operaciones que se pueden realizar en esa funcionalidad, como crear, actualizar, eliminar o consultar. Luego, la clase de repositorio implementa la interfaz definida anteriormente y utiliza distintas fuentes de datos (DataSource), como una base de datos local como Room (LocalDataSource), una capa de servicios como Firebase (RemoteDataSource) o una preferencia/caché (CacheDataSource). La clase repository actúa como mediador entre el origen de datos y el resto de la aplicación, y oculta los detalles de cómo se almacenan, recuperan o manipulan los datos.

## DataSource:

Se trata de una fuente de datos, es decir, de dónde se van a obtener los datos. Por lo tanto, todas las implementaciones de Firebase, Room, etc; estarán en estas clases y nadie fuera de ellas debe conocer los detalles.
Representa la capa encargada de gestionarse con los datos (de cualquier fuente). Contiene objetos como entidades de base de datos o modelos de red, que pueden ser utilizados por otras capas. Aquí se incluyen operaciones como la consulta, la actualización y el almacenamiento de datos.

Ejemplo: clases que representan objetos que se van a guardar o recuperar de bases de datos o servicios web.

| Remote | Local |
|---|---|
| Hace referencia a los datos obtenidos de fuentes externas, como servicios web, APIs o servidores. Es la capa encargada de manejar la comunicación con servidores remotos. | Representa los datos almacenados localmente, generalmente en bases de datos locales, como Room o SharedPreferences. Es la capa que gestiona el almacenamiento persistente en el dispositivo. |
| Ejemplo: realizar peticiones HTTP a una API REST utilizando Retrofit o cualquier otro cliente HTTP. | Ejemplo: acceder a una base de datos local o archivos para almacenar y recuperar datos cuando no se tiene conexión a internet. |



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

## Domain:

Es la capa de negocio o lógica de la aplicación. Aquí se manejan las reglas de negocio y la lógica que no depende de cómo se almacenan o recuperan los datos. En esta capa se suelen ubicar los use cases o interactors.

Ejemplo: clases o funciones que gestionan la lógica de negocio, como "Registrar usuario", "Hacer un pedido", etc.

## UI (User Interface):

Es la capa responsable de mostrar los datos al usuario y recibir sus interacciones. La capa UI es donde se encuentran las actividades, fragmentos, composables (en Jetpack Compose), y vistas en general.

Ejemplo: las pantallas de la app, como una pantalla de login o el dashboard.

# ¿Qué es un Manifest?

En Kotlin, **Manifest** generalmente se refiere al archivo **AndroidManifest.xml** en proyectos de Android. Este archivo es crucial para definir la configuración básica de la aplicación, como:

- **Permisos** requeridos por la app.
- **Componentes principales**, como actividades, servicios y receptores de difusión.
- **Intenciones** y configuraciones del sistema.

Es un archivo esencial para que el sistema operativo Android sepa cómo manejar la app y sus componentes.

# ¿Qué es un Fragment?

Un **Fragment** en Kotlin (y Android en general) es una **parte modular y reutilizable de la interfaz de usuario** de una actividad. Los fragments permiten dividir una actividad en múltiples secciones independientes que pueden tener su propia interfaz y comportamiento. Se usan para mejorar la flexibilidad y facilitar el manejo de interfaces en dispositivos con pantallas grandes (como tabletas).

Un **Fragment** se suele gestionar dentro de una **Activity** y puede ser reemplazado o añadido dinámicamente.

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

OFERTAS BLACK FRIDAY

msi

# Esta oferta

es como una doble victoria tuya: disfrútala ahora porque no pasa dos veces.

Ver ofertas


HASTA -40%

Tu viejo portátil ya dio lo que tenía que dar. Pásate a MSI: rápido, potente y sin dramas. Lo enciendes y estás listo para todo. Aprovecha las ofertas y despídete del modo “se cuelga cada dos por tres”.

# ¿Qué es un Activity?

Es como se le llama a una pantalla, pantalla que va a ver el usuario.
Se encarga solo del diseño.

En **Kotlin**, un **Activity** es una clase que representa una pantalla o interfaz de usuario en una aplicación Android. Cada **Activity** gestiona la interacción con el usuario y su ciclo de vida. Se usa para mostrar contenido, responder a acciones del usuario (como clics) y coordinar la navegación entre diferentes pantallas de la aplicación.

## El ciclo de vida de un Activity

La clase Activity proporciona una serie de métodos de llamada (onCreate, onStart, onResume, etc) que informan a la actividad cuando cambia un estado o que el sistema crea, detiene o reanuda una actividad, o destruye el proceso en el que reside la actividad.
**Más información en:**
https://developer.android.com/guide/components/activities/activity-lifecycle?hl=es-419

## ¿Qué es un Layout?

En Kotlin, un **Layout** se refiere a la estructura o diseño visual de una interfaz de usuario en una aplicación Android. Define cómo se organizan y distribuyen los elementos (como botones, imágenes, textos, etc.) en la pantalla. Los layouts se definen generalmente en archivos XML o programáticamente en el código Kotlin, y Android proporciona varios tipos como `LinearLayout`, `RelativeLayout`, `ConstraintLayout`, entre otros, para gestionar la disposición de los elementos.

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.





## Método onCreate()

Es el primer método que se ejecuta al iniciar la pantalla.

El método `onCreate()` en Kotlin, utilizado en el contexto de Android, es un método de ciclo de vida de una **actividad**. Se ejecuta cuando la actividad se **crea por primera vez** y se utiliza para inicializar la actividad, configurar la interfaz de usuario y realizar otras configuraciones iniciales.

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

  return go(f, seed, [])
}
```

Generalmente, dentro de `onCreate()`, se llama a `setContentView()` para establecer el layout de la actividad y se pueden inicializar variables, vistas o realizar tareas que deben ejecutarse al inicio.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main) // Establece el layout de la actividad
}
```



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

## Método onStart()

El método `onStart()` en Kotlin es un **método del ciclo de vida** de una **Activity** o **Fragment** en Android. Se ejecuta cuando la **actividad o fragmento** está a punto de volverse visible para el usuario, justo después de `onCreate()` y antes de `onResume()`.

Es útil para inicializar recursos que deben estar disponibles cuando la interfaz esté a la vista, como registrar listeners o iniciar procesos que no requieren interacción con el usuario.

```kotlin
override fun onStart() {
    super.onStart()
    // Código a ejecutar cuando la actividad está por hacerse visible
}
```

## Método onResume()

El método `onResume()` en Kotlin (y en Android en general) es un método del ciclo de vida de una **Activity** o **Fragment**. Se llama cuando la actividad o fragmento se está reanudando después de haber sido pausado o detenido (por ejemplo, cuando el usuario vuelve a la aplicación después de haberla minimizado).

En este método, puedes colocar código que necesite ejecutarse cuando la actividad o fragmento vuelve a estar en primer plano, como reiniciar animaciones, actualizar la UI o reanudar tareas que se pausaron en `onPause()` o `onStop()`.

```kotlin
override fun onResume() {
    super.onResume()
    // Código para reanudar tareas o actualizar la UI
}
```

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

## Método onPause()

El método `onPause()` en Kotlin es parte del ciclo de vida de una **actividad** o **fragmento** en Android. Se llama cuando la actividad o el fragmento está a punto de pasar a un estado en el que ya no está en primer plano, pero aún no se ha detenido completamente (por ejemplo, cuando otra actividad aparece en pantalla).

Es un buen lugar para **guardar datos** o **liberar recursos** temporales (como pausas en animaciones o detener servicios) antes de que la actividad pase a segundo plano.

```kotlin
override fun onPause() {
    super.onPause()
    // Guardar datos o liberar recursos
}
```

## Método onStop()

El método `onStop()` en Kotlin es parte del ciclo de vida de una actividad o fragmento en Android. Se llama cuando la actividad o fragmento ya no está visible para el usuario, es decir, cuando está a punto de ser detenida o reemplazada por otra actividad.

Es común usarlo para liberar recursos, guardar el estado de la aplicación o realizar tareas de limpieza, como detener procesos que no deben continuar cuando la actividad ya no está en primer plano.

```kotlin
override fun onStop() {
    super.onStop()
    // Realiza tareas de limpieza aquí
}
```

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

Ya has abierto los apuntes,
**te mereces ese descanso.**
También te mereces que no te cobren por tener una cuenta. **Cositas.**
Ven a la
Cuenta NoCuenta
Saber más



## Método onDestroy()

El método `onDestroy()` en Kotlin, especialmente en el contexto de actividades o fragmentos en Android, es un **método del ciclo de vida** que se llama cuando la actividad o fragmento está a punto de ser destruido. Esto ocurre cuando la actividad se cierra o se destruye debido a cambios en la configuración (como rotaciones de pantalla) o cuando el sistema necesita liberar recursos.

En este método, puedes realizar tareas de limpieza, como liberar recursos, cancelar hilos, cerrar conexiones de bases de datos, o eliminar escuchas de eventos.

```kotlin
override fun onDestroy() {
    super.onDestroy()
    // Realizar tareas de limpieza, como liberar recursos
}
```

# ¿Qué es un Logcat?

**Logcat** en Kotlin (y en Android en general) es una herramienta que permite **registrar y mostrar mensajes de depuración** y errores durante la ejecución de una aplicación. Los mensajes se pueden mostrar en la consola de Android Studio usando la clase `Log`.

```kotlin
import android.util.Log

Log.d("MiTag", "Este es un mensaje de depuración")
```

# ¿Qué es una Preview?

Una **preview** en Kotlin, específicamente en el desarrollo con Jetpack Compose, es una anotación (`@Preview`) que permite visualizar el diseño de una composición directamente en el editor de Android Studio, sin necesidad de ejecutar la aplicación.

Se utiliza para mostrar cómo se verá una función de UI y facilita la creación y prueba de interfaces gráficas de forma más rápida.

Para hacer una **preview** en Kotlin con Jetpack Compose, sigue estos pasos:



do your thing
Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

1. **Importa las librerías necesarias**: Asegúrate de tener Jetpack Compose configurado en tu proyecto.
2. **Crea una función de UI**: Define una función anotada con `@Composable` que represente la interfaz que deseas mostrar.
3. **Agrega la anotación `@Preview`**: Encima de la función que deseas previsualizar, usa `@Preview` para habilitar la vista previa.

```
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.material3.Text

@Composable
fun Greeting(name: String) {
    Text(text = "Hola, $name!")
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    Greeting(name = "Mundo")
}
```

4. Visualiza la preview
    - Abre la pestaña de **Preview** en Android Studio.
    - La función anotada con `@Preview` aparecerá renderizada ahí.

# ¿Qué es un Composable?

Un **Composable** en Kotlin es una función especial utilizada en **Jetpack Compose**, el framework de UI para Android. Permite crear interfaces de usuario declarativas y reutilizables. Las funciones Composable definen la estructura de la interfaz de usuario, y se pueden componer de manera flexible y eficiente, actualizando solo las partes necesarias de la UI cuando cambian los datos. Se anotan con la palabra clave `@Composable`.
**Más información en:**
https://developer.android.com/compose



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.
WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

## ¿Qué es un Scaffold?

En **Kotlin**, específicamente en el contexto de **Jetpack Compose** (la librería de UI para aplicaciones Android), un **Scaffold** es un componente que proporciona una estructura básica de diseño para tu interfaz de usuario. Incluye elementos como:

- Barra de aplicación (App Bar)
- Cuerpo principal de la pantalla
- Barra de navegación inferior
- Menú flotante (Floating Action Button)

El **Scaffold** ayuda a organizar la disposición de estos componentes comunes, permitiendo crear aplicaciones con una estructura consistente.

# ¿Qué es un ViewModel?

Un **ViewModel** en Kotlin es una clase que se utiliza para **almacenar y gestionar datos relacionados con la interfaz de usuario** (UI) de manera eficiente. Se utiliza principalmente en aplicaciones Android, especialmente en el contexto de **MVVM (Modelo-Vista-ViewModel)**.

El **ViewModel** permite:

- **Mantener datos a través de cambios de configuración** (como rotaciones de pantalla).
- **Desacoplar la lógica de negocio de la UI**, gestionando la interacción con el modelo de datos.
- Proveer datos de manera observable para que la vista se actualice automáticamente cuando los datos cambian.

Se encuentra en el paquete `androidx.lifecycle` y se usa junto con **LiveData** o **StateFlow** para manejar datos reactivos.

La clase ViewModel es dónde haremos la lógica de presentación o un contenedor de estado a nivel de pantalla. Expone el estado a la IU y encapsula la lógica de presentación relacionada a esa pantalla lanzando las interacciones de los usuarios en los distintos casos de uso. Su principal ventaja es que almacena en caché el estado y lo conserva durante los cambios de configuración (rotaciones de pantallas, cambios de idioma, etc).
En esta clase se encuentra la lógica de presentación o un contenedor de estado a nivel de pantalla.
Expone el estado a la IU y encapsula la lógica de presentación/negocio relacionada.
Su principal ventaja es que almacena en caché el estado y lo conserva durante los cambios de configuración. Esto significa que la IU no tiene que recuperar datos cuando navegas entre actividades o si sigues cambios de configuración, como cuando rotas la pantalla.

**ViewModel sin lógica de negocio.** La lógica de datos y de negocio la hemos quitado del ViewModel.



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.
WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

Ya has abierto los apuntes,
**te mereces ese descanso.**
También te mereces que no te cobren por tener una cuenta. **Cositas.**
Ven a la
Cuenta NoCuenta
Saber más



**No UseCases.** No vamos a usar los casos de uso para evitar una complejidad mayor en nuestra arquitectura por el poco tiempo que tenemos, para ello los Flow los implementaremos en los ViewModel y se llamarán a los distintos métodos de los repositorios para hacer cosas.

**Más información en:**
https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419

## ¿Qué es un Memory Leak?

Un **memory leak** en Kotlin ocurre cuando la memoria utilizada por objetos que ya no son necesarios no se libera, lo que provoca un consumo innecesario de memoria y eventualmente puede hacer que la aplicación se quede sin recursos. Esto suele pasar cuando las referencias a objetos se mantienen de manera inapropiada (por ejemplo, en un ciclo de referencias) y no se pueden recolectar por el **garbage collector**.

Si en MainViewModel se hace instancia del Activity, se puede crear una fuga de memoria (memory leak).

## ¿Qué es un MainViewModel?

Un **MainViewModel** en Kotlin es una clase que forma parte de la arquitectura **MVVM** (Modelo-Vista-ViewModel). Su propósito es gestionar los datos y la lógica de presentación de la interfaz de usuario, separándolos de la vista (Activity o Fragment). El **ViewModel** mantiene el estado y los datos durante los cambios de configuración (como rotaciones de pantalla) y permite que la vista se actualice de manera reactiva a través de observadores.

Por ejemplo, un **MainViewModel** podría manejar la lógica de cargar datos desde una API y exponer esos datos a la interfaz para que se muestren de manera eficiente.

Tiene la lógica de presentación → Normalmente hay un ViewModel por cada pantalla

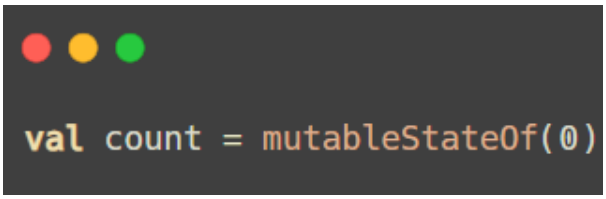
# ¿Qué es un MutableState?

En Kotlin, un `mutableState` es un tipo de variable que permite almacenar y modificar su valor de manera reactiva. Se utiliza principalmente en la programación con **Jetpack Compose** para representar un estado que puede cambiar a lo largo del tiempo.

Un ejemplo típico es `mutableStateOf()`, que se usa para crear una variable que puede ser observada y actualizada:



do your thing
Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.




El valor de `count` puede ser modificado con `count.value = nuevoValor`, y cualquier cambio en él actualizará automáticamente la UI si está vinculado a un componente de la interfaz en Jetpack Compose.

## ¿Qué es un MutableStateFlow?

Un **`mutableStateFlow`** en Kotlin es una clase de flujo (flow) que permite almacenar y emitir valores de forma reactiva y mutable. Es parte de la **`StateFlow`** que pertenece a las **corrutinas** de Kotlin y se utiliza para manejar y observar estados en aplicaciones reactivas.

- **Mutable**: Puedes modificar su valor usando el método **`value`**.
- **StateFlow**: Es un flujo que siempre tiene un valor actual y puede ser observado por otros componentes para recibir actualizaciones cuando el valor cambia.

Es útil para mantener y compartir estados en una aplicación de manera eficiente y segura en entornos de concurrencia.

Se trata de un contenedor observable de estados que son emitidos cuando son nuevos (la primera vez que observas el flow) o de estados nuevos utilizando el método "update". El valor de estado actual también se puede leer a través de su propiedad value. Para actualizar el estado y enviarlo al flujo, asigna un nuevo valor a la propiedad value de la clase MutableStateFlow usando el método "update".

| mutableState | mutableStateFlow |
|---|---|
| Se utiliza en el contexto de **Jetpack Compose** para gestionar estados locales en la interfaz de usuario. Es más específico para el manejo de la UI, y su valor se actualiza mediante la función **`value`**. | Es parte de **Kotlin Coroutines** y es un flujo reactivo que permite emitir y observar cambios de estado de forma más general, no limitado solo a la UI. Es adecuado para manejar estados en aplicaciones que utilizan corrutinas y programación reactiva. |

En resumen, mutableState es para estados en la UI en Jetpack Compose, y mutableStateFlow es un flujo reactivo más general para gestionar estados en toda la aplicación.

**Más información en:**
https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=es-419

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

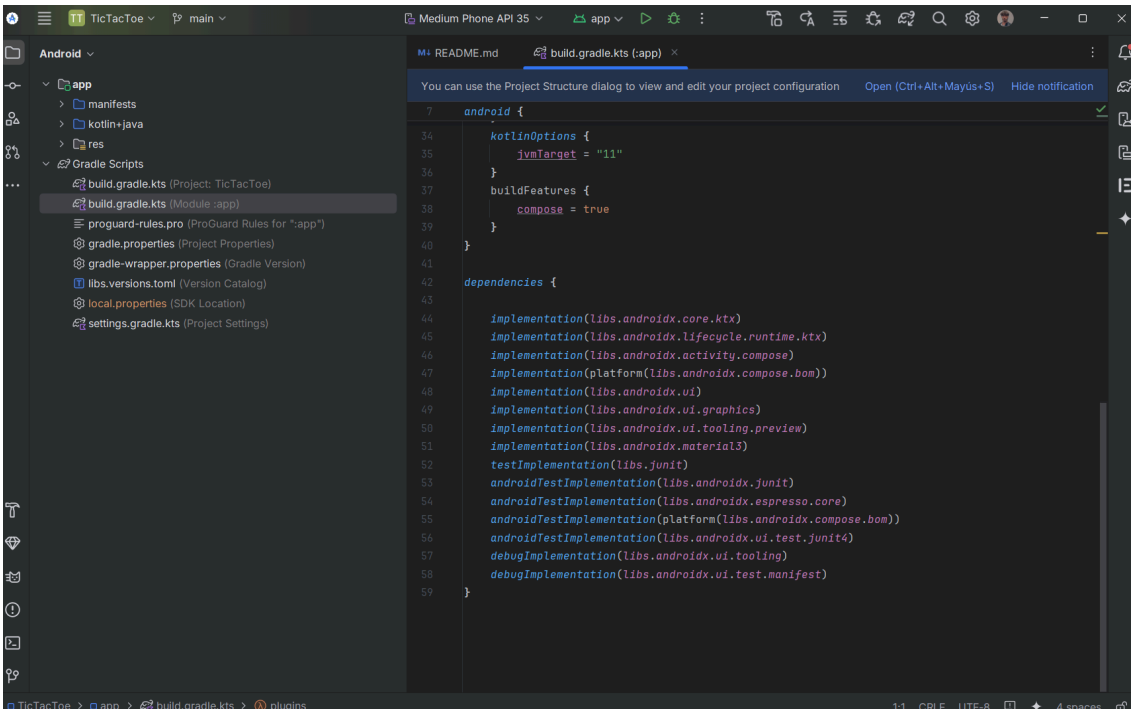


WUOLAH

Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

# ¿Cómo añadir una dependencia?

1. Copiamos el enlace de la dependencia
2. Dentro de build gradle del app, nos vamos a dependencias y la pegamos




3. Sincronizamos

# ¿Qué es o para qué sirve Weight/Peso?

En Kotlin, dentro de Jetpack Compose, el atributo weight se utiliza en los elementos de un Row o Column para distribuir el espacio disponible proporcionalmente entre los componentes.

El **`weight`** define cuánto espacio debe ocupar un componente relativo al resto de los componentes dentro de un **`Row`** o **`Column`**. Un mayor valor de **`weight`** significa que el componente ocupará más espacio.

El peso indica la prioridad que tiene un elemento frente a sus hermanas.

```
Column {
    Text("Item 1", Modifier.weight(1f)) // Ocupa más espacio
    Text("Item 2", Modifier.weight(2f)) // Ocupa el doble que Item 1
}
```



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

WUOLAH

Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.





# ¿Que es un singleton?

Es una instancia única de un objeto.
Un **Singleton** en Kotlin es un patrón de diseño que garantiza que una clase tenga **una única instancia** durante toda la ejecución del programa, proporcionando un punto de acceso global a esa instancia.

### Implementación en Kotlin

En Kotlin, los singletons se implementan fácilmente usando la palabra clave object. Por ejemplo:

```kotlin
object MySingleton {
    var data: String = "Hola, soy un Singleton"
    fun doSomething() {
        println("Ejecutando una función del Singleton")
    }
}
```

### Características:

- Instancia única: object asegura que se cree solo una instancia automáticamente.
- Globalmente accesible: Puedes acceder a sus propiedades y métodos desde cualquier parte del programa.

### Usos:

- Manejo de configuraciones.
- Gestión de recursos compartidos (como conexión a una base de datos).
- Controladores de logging o caché.



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

# Método init

El init es el primer método que se usa en cualquier clase de kotlin.
En Kotlin, un **método `init`** es un bloque de inicialización que se ejecuta automáticamente cuando se crea una instancia de una clase. Se utiliza para inicializar propiedades o ejecutar código que debe ejecutarse al crear el objeto.

### Características:

- Declarado con la palabra clave init.
- Puede haber varios bloques init en una clase, y se ejecutan en el orden en que aparecen.
- Se ejecuta después de que las propiedades primarias de la clase se inicializan.

```kotlin
class Persona(val nombre: String) {
    init {
        println("Hola, mi nombre es $nombre")
    }
}

val persona = Persona("Juan") // Imprime: Hola, mi nombre es Juan
```

El bloque `init` es útil para realizar configuraciones adicionales al inicializar un objeto.

# LocaDataSource.

Tendremos la implementación para el guardado/obtención de nuestros datos de forma local. Tendrá los métodos necesarios para que nuestro Repositorio pueda obtenerlos los datos. Usaremos SharedPreferences para la parte local.

# SharedPreferences

SharedPreferences en Kotlin es una API de Android que permite almacenar datos clave-valor de manera persistente, incluso cuando la aplicación se cierra. Es ideal para guardar configuraciones, preferencias del usuario o pequeñas cantidades de datos que no necesitan una base de datos completa.
Usos comunes:

- Guardar el estado del usuario (ej.: si está logueado).
- Almacenar configuraciones (tema oscuro, idioma preferido, etc.).
- Recordar información pequeña (última puntuación, nombre de usuario, etc.).

Ventajas:

- Fácil de implementar.
- Ligero y eficiente para datos pequeños.

Sin embargo, no es adecuado para manejar grandes volúmenes de datos o datos relacionales.



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.
WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

Es como una especie de xml, cuanto más complejos sea lo que metamos mas complejo será sacarlo. Como una base de Datos rústicas. Cuando borras caché, borras SharedPreferences
Tiene limite de 1 mega de String
Listas

## Preferencias compartidas

Puedes crear un nuevo archivo de preferencias compartidas o acceder a uno existente llamando a uno de estos métodos:

- **getSharedPreferences()**: Usa este método si necesitas varios archivos de preferencias compartidas identificados por nombre, que especificas con el primer parámetro. Puedes llamar a este método desde cualquier instancia de Context en tu app.
- **getPreferences()**: Usa este método desde una instancia de Activity si necesitas utilizar un solo archivo de preferencias compartidas para la actividad. Como este método recupera un archivo de preferencias compartidas predeterminado que pertenece a la actividad, no necesitas indicar un nombre.

```
val sharedPref = activity?.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE)
```

## Como escribir en preferencias

Para realizar operaciones de escritura en el archivo de preferencias compartidas, crea un SharedPreferences.Editor llamando a edit() en tu SharedPreferences.

Pasa las claves y los valores que desees escribir con métodos como putInt() y putString(). Luego, llama a apply() o a commit() para guardar los cambios.

```
val sharedPref = activity?.getPreferences(Context.MODE_PRIVATE) ?: return
with (sharedPref.edit()) {
    putInt(getString(R.string.saved_high_score_key), newHighScore)
    apply()
}
```

**Más información en:**
https://developer.android.com/training/data-storage/shared-preferences?hl=es-419

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.




# Sealed class

Una sealed class en Kotlin es una clase que restringe la herencia a un conjunto fijo de subclases conocidas en tiempo de compilación. Esto la hace útil para representar tipos jerárquicos o modelos limitados, como estados, resultados o eventos.

**Características principales:**

- Las subclases deben definirse en el mismo archivo que la clase sellada.
- Garantiza la exhaustividad en expresiones when (no necesitas el caso else si manejas todas las subclases).

```
sealed class Result {
    data class Success(val data: String) : Result()
    data class Error(val error: Throwable) : Result()
    object Loading : Result()
}

// Ejemplo de uso
fun handleResult(result: Result) {
    when (result) {
        is Result.Success -> println("Éxito: ${result.data}")
        is Result.Error -> println("Error: ${result.error.message}")
        Result.Loading -> println("Cargando...")
    }
}
```

# .lastIndex

En Kotlin, lastIndex es una propiedad de colecciones (como listas o arrays) que devuelve el índice del último elemento de la colección.

Características:

- Es equivalente a size - 1 o length - 1.
- Es útil para acceder al último índice sin necesidad de calcularlo manualmente.

```
val list = listOf("A", "B", "C")
println(list.lastIndex) // Salida: 2 (índice del último elemento)
```



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

# intrinsicSize

Un elemento padre crece todo lo que le permitan sus hijos
En Kotlin, intrinsicSize no es un término específico del lenguaje, pero en el contexto de diseño y desarrollo de interfaces (como en Compose o Jetpack), se refiere al tamaño "intrínseco" o natural de un elemento basado en su contenido.
Por ejemplo:

- El intrinsicSize de un texto sería el tamaño necesario para contener todos los caracteres sin truncarlos.
- En un diseño, este tamaño puede ser usado para adaptar contenedores u otros elementos a sus hijos, asegurando que el diseño sea dinámico y fluido.

Se utiliza principalmente para calcular dimensiones en componentes que dependen del contenido, especialmente en layouts personalizados.

# Funciones de extensión

Una **función de extensión** es una función que añade nuevas funcionalidades a una clase existente sin modificar su código original. Estas funciones se definen fuera de la clase y se acceden como si fueran parte de ella.

Se usan con el operador punto (`.`), y su ventaja principal es mejorar la legibilidad del código y extender funcionalidades sin herencia.

## Las 5 funciones de extensión más importantes (según ChatGPT)

1. **let**: Ejecuta un bloque de código sólo si el objeto no es null. Devuelve el resultado del bloque.

```kotlin
val name: String? = "Kotlin"
name?.let { println(it.uppercase()) } // Imprime "KOTLIN" si no es null
```

2. **apply**: Configura un objeto y devuelve el objeto mismo. Útil para inicialización.

```kotlin
val user = User().apply {
    name = "John"
    age = 30
}
```

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

1/6
Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

# POV: quedas con alguien

## y no se parece a sus fotos.

Si te pasa con un pedido, el seguro de compra online* de tu **Cuenta NoCuenta** de ING te cubre. **¡Y es gratis!**

**Saber más**

*Compras superiores a 30 €. Más info en ing.es

3. **run**: Combina la configuración (apply) y el procesamiento (let). Devuelve el resultado del bloque.

```
val result = user.run {
    "$name es $age años viejo."
}
```

4. **also**: Similar a let, pero devuelve el objeto original en lugar del resultado del bloque. Útil para realizar acciones adicionales (como logs).

```
val list = mutableListOf(1, 2, 3).also {
    println("Lista inicial: $it")
}
```

5. **with**: Funciona como run, pero no requiere un receptor explícito. Se usa para ejecutar acciones en un objeto.

```
val message = with(user) {
    "$name tiene $age años."
}
```

# ¿Qué es un Modifier?

En Kotlin, un modifier es una palabra clave que se usa para alterar el comportamiento o la visibilidad de clases, funciones, propiedades o constructores. Los modifiers controlan aspectos como la visibilidad, la herencia y cómo se accede o utiliza un miembro dentro de un programa.

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

CARPE DIEM
Organiza tu futuro:
estudia hoy para destacar mañana.
En Carpe Diem te esperan cursos adaptados a ti.
Una forma fácil y real de avanzar profesionalmente.
¡Quiero formarme!
Acreditados
Económicos
Desde casa
Elige tu curso



## Ejemplos de modifiers:

1. **De visibilidad:**
   - private: Accesible solo dentro del archivo o clase donde se define.
   - protected: Accesible dentro de la clase y sus subclases.
   - internal: Accesible dentro del mismo módulo.
   - public (por defecto): Accesible desde cualquier lugar.
2. **Para herencia y sobreescritura:**
   - open: Permite que una clase o función sea heredada o sobreescrita.
   - final: Impide que una clase o función sea heredada o sobreescrita.
   - abstract: Declara un miembro sin implementación, obligando a las subclases a implementarlo.
3. **Otros:**
   - data: Marca una clase como "data class" para representar datos.
   - inline: Optimiza funciones en tiempo de compilación.
   - suspend: Define funciones que se pueden usar en coroutines.

Los modifiers son esenciales para controlar cómo los elementos interactúan dentro de un proyecto y garantizar la encapsulación adecuada.

# Chunked

En Kotlin, chunked es una función de extensión que divide una colección (como una lista o una cadena) en partes (chunks) de un tamaño especificado y devuelve una lista de esas partes.

| Uso | Ventaja |
|---|---|
| **Listas o colecciones:** Divide una lista en sublistas de tamaño fijo. **Cadenas:** Divide una cadena en fragmentos de longitud fija. | Es útil para procesar datos en bloques más manejables. |



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

# I18N y A11Y

Ambos conceptos son esenciales para crear aplicaciones inclusivas y globales.

## I18N

Es el proceso de preparar una aplicación para soportar múltiples idiomas y configuraciones regionales sin necesidad de cambiar el código fuente. En Kotlin (y Android), esto se implementa usando archivos de recursos (res/values/strings.xml), donde se definen cadenas para diferentes idiomas (res/values-es/strings.xml para español, por ejemplo).

## A11Y

Es el diseño y desarrollo de aplicaciones para que sean usables por personas con discapacidades o necesidades especiales. En Kotlin (y Android), se mejora la accesibilidad usando herramientas como TalkBack, etiquetas descriptivas (contentDescription), y soporte para gestos y navegación alternativa.

| I18N | A11Y |
|---|---|
| Internationalization | Accesibility |

# Context

Toda Activity tiene un contexto.
El Context en Kotlin (y Android) es una interfaz que proporciona acceso a información global de la aplicación, como recursos, archivos, bases de datos, preferencias y otros servicios del sistema. Es necesario para interactuar con elementos del sistema operativo, como obtener SharedPreferences o acceder a la base de datos.

# Patrón observer

Hace que el activity escuche a una cajita en la que ViewModel va metiendo órdenes, así el Activity recibe actualizaciones.
El patrón Observer en Kotlin es un patrón de diseño que permite a un objeto (el "sujeto") notificar a otros objetos (los "observadores") sobre cambios en su estado sin necesidad de que los observadores estén estrechamente acoplados al sujeto. Es muy útil cuando un objeto necesita notificar a múltiples objetos interesados en su estado, como en interfaces gráficas o sistemas de eventos.

En Kotlin, este patrón se puede implementar utilizando MutableList para almacenar los observadores y una función para notificar a todos los observadores cuando ocurra un cambio.



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.
WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

# .copy()

Solo existe para data class.
Este se utiliza para crear una copia de un objeto, especialmente en el contexto de las data clases.

Cuando defines una data class en Kotlin, el compilador genera automáticamente un método copy(). Este método permite crear una nueva instancia del objeto con los mismos valores que el objeto original, pero permitiendo modificar algunos de sus atributos si es necesario.

```kotlin
data class Persona(val nombre: String, val edad: Int)

fun main() {
    val persona1 = Persona("Juan", 30)

    // Crear una copia de persona1 con una edad diferente
    val persona2 = persona1.copy(edad = 31)

    println(persona1)  // Persona(nombre=Juan, edad=30)
    println(persona2)  // Persona(nombre=Juan, edad=31)
}
```

El método copy() crea una nueva instancia de la data class con los mismos valores que el objeto original, pero puedes sobrescribir algunos parámetros al llamarlo. En el ejemplo anterior, la edad de persona2 se cambia a 31, pero el nombre sigue siendo el mismo.

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

CARPE DIEM
Organiza tu futuro:
estudia hoy para destacar mañana.
En Carpe Diem te esperan cursos adaptados a ti.
Una forma fácil y real de avanzar profesionalmente.
¡Quiero formarme!
Acreditados
Económicos
Desde casa
Elige tu curso



# Flow

Flow en Kotlin es un tipo de flujo de datos asíncrono que emite múltiples valores a lo largo del tiempo. Es similar a una List, pero en lugar de contener los datos todos a la vez, los emite secuencialmente.

Se usa en programación reactiva y permite manejar emisiones de datos de forma suspendida, sin bloquear hilos. Se define con flow {} y se recoge con collect {}.

```kotlin
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

fun miFlow(): Flow<Int> = flow {
    for (i in 1..3) {
        emit(i) // Emite valores uno a uno
    }
}

fun main() = runBlocking {
    miFlow().collect { valor ->
        println(valor) // Recoge los valores emitidos
    }
}
```

Ideal para streams de datos como eventos de UI o llamadas a API.

## Operadores

**onEach** → Sí se usa con corrutinas. Es un operador de Flow que permite ejecutar código cada vez que se emite un valor, sin suspender el flujo.



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

# Corrutinas

En Kotlin, una corrutina es una forma de manejar tareas asíncronas y concurrentes de manera eficiente y sin bloquear el hilo principal. Permiten ejecutar código de manera suspendida y retomarlo posteriormente sin consumir recursos innecesarios. Se gestionan con suspend functions y constructores como launch y async dentro de un CoroutineScope.

| Corrutinas | Hilos |
|---|---|
| Es más ligera y se ejecuta dentro de un hilo. No bloquea el hilo en el que se ejecuta, sino que lo **suspende y reanuda** sin cambiar de contexto, lo que mejora la eficiencia y el rendimiento. | Es una unidad de ejecución administrada por el sistema operativo. Cambiar entre hilos es costoso en términos de rendimiento, ya que implica cambios de contexto. |

## dispatcher

En Kotlin, un Dispatcher determina en qué hilo o pool de hilos se ejecuta una coroutine. Forma parte de CoroutineContext y puede ser:

- Dispatchers.Main: Para la UI (solo en Android).
- Dispatchers.IO: Para operaciones de I/O (lectura/escritura de archivos, red).
- Dispatchers.Default: Para tareas intensivas en CPU.
- Dispatchers.Unconfined: No fija un hilo específico.

Es el donde se va lanzar
Main → hilo principal
io → background

## Scope

En Kotlin, scope (alcance) se refiere al contexto en el que una variable, función u objeto es accesible dentro del código. Kotlin proporciona funciones de alcance (scope functions) como let, run, with, apply y also, que ayudan a estructurar el código de manera más concisa y evitar repeticiones.
Se refiere al ámbito de la corrutina. Es decir que si yo mi corrutina la lanzo a una pantalla, cnd se deje de usar, la corrutina muere

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.

CoroutineScope

**viewModelScope** → Sí está relacionado con corrutinas. Es un CoroutineScope que vive mientras el ViewModel esté activo y se usa para lanzar corrutinas de manera segura en ViewModel.
**GlobalScope** → CoroutineScope que vive durante toda la ejecución de la aplicación y no está vinculado a ningún componente específico, como Activity o ViewModel.
⚠️ Precaución con GlobalScope

- No se cancela automáticamente cuando la UI cambia o la app se cierra.
- Puede generar filtraciones de memoria si no se maneja correctamente.
- Se recomienda usar viewModelScope, lifecycleScope o CoroutineScope personalizado en su lugar.

Ejemplo de uso (⚠️ No recomendado en muchos casos):

```
GlobalScope.launch {
    delay(1000)
    println("Ejecutando en GlobalScope")
}
```

GLOBAL SCOPE CANTA A CHAT GPT

# Suspend

En Kotlin, suspend es una palabra clave que indica que una función es suspendida, lo que significa que puede ejecutarse de forma asíncrona sin bloquear el hilo actual.

**Características de suspend:**

- Solo puede ser llamada desde otra función suspend o dentro de una corrutina.
- Permite el uso de funciones como delay(), withContext(), etc.

```
suspend fun obtenerDatos(): String {
    delay(1000) // Simula una tarea asíncrona
    return "Datos cargados"
}

fun main() = runBlocking {
    println("Cargando...")
    val resultado = obtenerDatos() // Se ejecuta sin bloquear
    println(resultado)
}
```

**No crea un nuevo hilo**, solo pausa y reanuda la ejecución de manera eficiente.
Suspende la ejecución de la corrutina. La corrutina se pausa.



Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.
WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.




## by lazy

Es un delegado de propiedad que permite la inicialización diferida (**lazy initialization**) de una variable. Esto significa que el valor de la variable solo se calculará la primera vez que se acceda a ella, y luego se almacenará en caché para usos posteriores.

```kotlin
val mensaje: String by lazy {
    println("Inicializando...")
    "Hola, Kotlin!"
}

fun main() {
    println("Antes de acceder a mensaje")
    println(mensaje) // Aquí se inicializa la variable
    println(mensaje) // Aquí usa el valor ya inicializado
}
```

Su salida sería:

```
Antes de acceder a mensaje
Inicializando...
Hola, Kotlin!
Hola, Kotlin!
```

En este caso, `"Inicializando..."` solo se imprime una vez, ya que `mensaje` se inicializa solo en su primer acceso y reutiliza el valor en las siguientes llamadas.

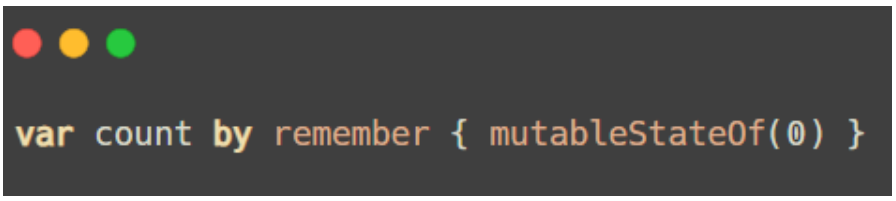


Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.

# remember

Recuerda pero no almacena.

En **Kotlin**, específicamente en **Jetpack Compose**, `remember` es una función utilizada para almacenar y preservar el estado de una variable en recomposiciones. Ayuda a evitar que se reinicialicen valores cada vez que la interfaz de usuario se vuelve a componer.



```
var count by remember { mutableStateOf(0) }
```

En este caso, `count` conservará su valor a través de recomposiciones dentro de una `@Composable`.

Si necesitas que el estado sobreviva cambios de configuración (como rotaciones de pantalla), usa `rememberSaveable` en lugar de `remember`.

Reservados todos los derechos. No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH
Te has descargado este apunte gracias a la publicidad. También puedes eliminarla con 1 coin.