96. Android avanzado

•2.1.Hilos. •2.2.Hilo Secundarios. •2.3.Tareas Asíncronas. •2.4.Timer •2.5.TimerTask. •2.6.Planificador de tareas (Scheduler). •3.Los Servicios. •3.1.Ciclo de vida de los Servicios. •3.2.Started Service. •3.3.Bound Service. •3.4.Intent Service. •3.5.Servicios en Primer Plano. •4.Receptores de Broadcast. •4.1.Envío de Broadcast. -->

Que contiene este tema: * Aplicación Android * Proceso * Hilo (thread) * Corrutina * Flow * Services * Receptores de Broadcast

96.1 Android Application

`Application` es la clase inicial de cualquier aplicación y "contiene" todas las Activity, Service, etc.

Sobrecargar sus métodos puede servir para:

- **Inicialización Global**: Si necesitas inicializar recursos o configuraciones que se utilizarán en toda la aplicación, como bibliotecas de terceros, es el lugar ideal para hacerlo. Por ejemplo, si estás usando una base de datos como Room o una biblioteca de análisis de datos, puedes inicializarla aquí.
- **Gestión de Estado de Aplicación**: Puedes usar la clase Application para mantener estados globales. Por ejemplo, si tienes datos de usuario que quieres acceder desde varias actividades o fragmentos.
- **Seguimiento y Análisis**: Para realizar seguimientos de uso o errores a nivel de aplicación, como enviar informes de errores a Firebase o Google Analytics, implementar esto en la clase Application asegura que esté disponible desde el inicio.
- **Customización del Contexto**: Como la clase Application proporciona un contexto de aplicación, es útil para situaciones en las que necesitas un contexto que esté disponible durante todo el ciclo de vida de tu aplicación.

Ejemplo.
inicializar una base de datos.

1. Derivamos de la clase `Application`

```
class MiAplicacion : Application() {

    override fun onCreate() {
        super.onCreate()
        // Inicializar la base de datos
        MiBaseDeDatos.init(this)
    }
}
```

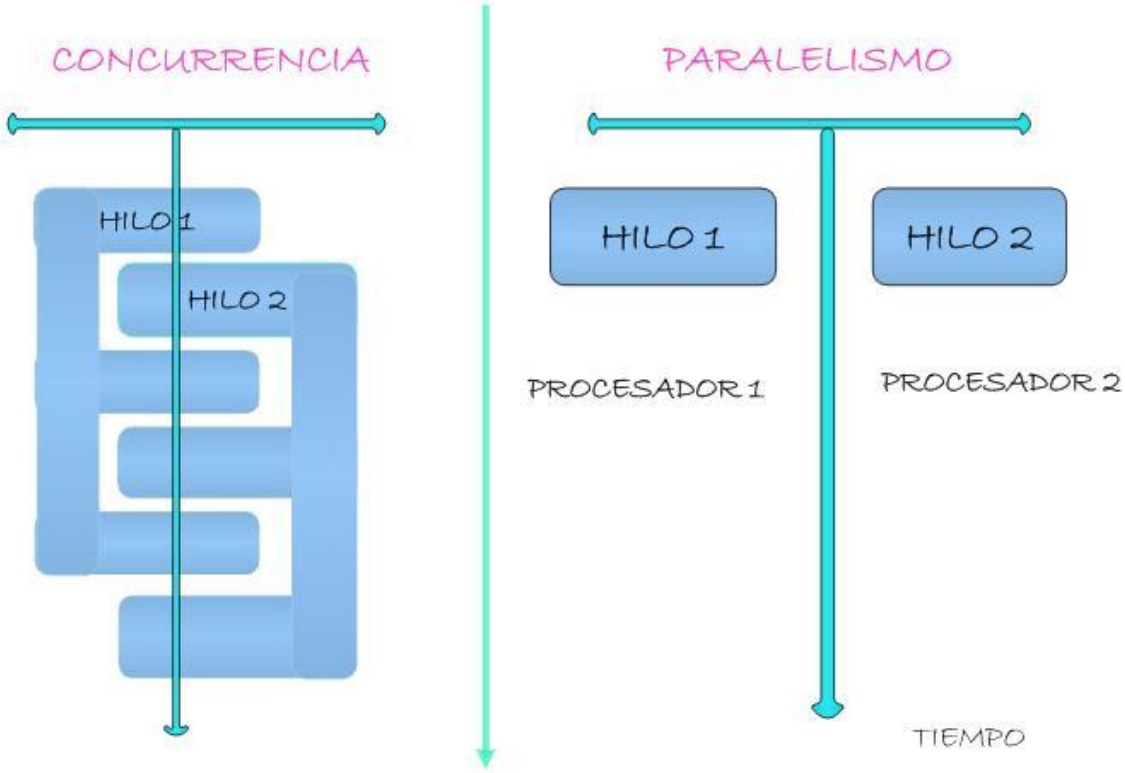
2. Añadimos en el Manifest el atributo "name" para poder usar el código de nuestra aplicación:

```
<application
    android:name=".MiAplicacion"
```

```
    ...>
    ...
</application>
```

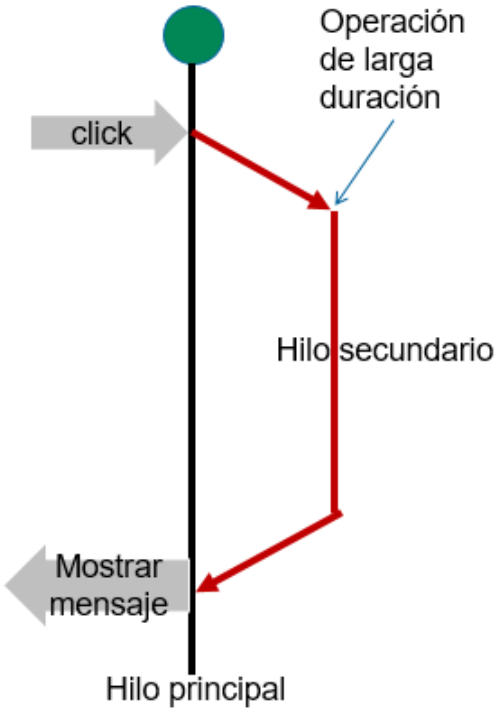
96.2 Timer, Hilos (Threads)y procesos

Concurrencia y paralelismo



96.2.1 Hilos

Un hilo (Thread) es una unidad de ejecución dentro de un proceso. Android, como la mayoría de los sistemas operativos, es multitarea y permite ejecutar múltiples hilos simultáneamente.



En Android las aplicaciones se inician en un `Main Thread` : * **UI Thread o Main Thread**: Cada aplicación de Android tiene un hilo principal, conocido como `UI Thread` o `Main Thread`. Este hilo se encarga de gestionar la interfaz de usuario y las interacciones con el usuario. **Es fundamental no bloquear este hilo** con operaciones largas, ya que puede causar que la aplicación parezca congelada o no responda.
* **Background Threads**. Para operaciones largas, como llamadas a la red, operaciones de base de datos o cálculos intensivos, debes usar hilos en segundo plano. Esto previene el bloqueo del UI Thread.

Kotlin proporciona varias formas de manejar la ejecución en segundo plano, como **Coroutines**, **Threads** y **Executors**.

En kotlin se prefieren las **Coroutines** para operaciones asíncronas y multitarea. Las coroutines son más ligeras que los threads tradicionales y proporcionan una sintaxis más sencilla y directa para manejar operaciones asíncronas.

Cada Hilo tiene asignada una **prioridad**, ejecutandose primero los de mayor prioridad. Un Hilo puede estar marcado como **daemon** (iniciado por el SO)
Cuando un hilo crea uno nuevo, este tiene la misma prioridad. Un nuevo hilo *daemon* solamente se puede iniciar a partir de otro hilo daemon.

Disponemos de dos formas de crear nuevos Hilos de ejecución:

1. Crear una subclase de la clase `Thread` y sobreescribir el método run()

```
class MiHilo : Thread() {
    override fun run() {
        // Aquí va el código que se ejecutará en este hilo.
        println("¡Hola desde MiHilo!")
    }
}
```

Y para crea y ejecuta este nuevo hilo usamos el método `start()` :

```
val thread = MiHilo()
thread.start()
```

1. La segunda forma es crear una clase que implemente el interfaz `Runnable`

```
class MiRunnable : Runnable {
    override fun run() {
        // Aquí va el código que se ejecutará en este hilo.
        println("¡Hola desde MiRunnable!")
    }
}
```

Y se inicializa usando una instancia como parámetro de `Thread` :

```
val hiloRun = Thread(MiRunnable())
hiloRun.start()
```

3. Una forma alternativa es usar la convención SAM, para llamar tanto interfaces Java como Kotlin. Esto significa que una función literal Kotlin puede ser convertida automáticamente en la implementación de un interfaz Java (que tenga un único método) siempre que los tipos de los parámetros de la interfaz coincidan con los tipos de la función Koltin

Ejemplo:

```
val thread = Thread {
    println("${Thread.currentThread()} has run.")
}
thread.start()
```

4. También podemos usar la función kotlin:

```
thread(
  start: Boolean = true,
  isDaemon: Boolean = false,
  contextClassLoader: ClassLoader? = null,
  name: String? = null,
  priority: Int = -1,
  block: () -> Unit
): Thread
```

* start: Si es true, el thread comienza a ejecutarse inmediatamente después de su creación. * isDaemon: Un daemon thread se ejecuta en segundo plano y no impide que la JVM se cierre. Ponlo en true para un daemon thread. * contextClassLoader: El ClassLoader para usar en este thread. * name: El nombre del thread, útil para depuración. priority: La prioridad del thread. Por defecto, es la prioridad normal. * block: Un bloque de código que el thread ejecutará.

Ejemplo de uso:

```
thread(start = true, name = "MiThread") {
    // Aquí va el código que quieres ejecutar en el thread
    println("¡Esto se está ejecutando en un thread separado!")
}
```

Referencia de la clase Thread para Java

96.2.1.1 Sincronización de hilos

La sincronización de hilos es el proceso mediante el cual controlas el acceso a recursos compartidos en un entorno de múltiples hilos (threads) para prevenir conflictos y asegurar la coherencia de los datos. Esto es importante porque varios hilos pueden intentar leer o modificar el mismo recurso al mismo tiempo, lo que puede llevar a errores o a un estado inconsistente del recurso y dar lugar a situaciones como:

- Condiciones de **Carrera**: Ocurren cuando varios hilos acceden y modifican un recurso compartido de manera concurrente, y el resultado final depende del orden en que se ejecuten los accesos.
- **Deadlocks**: Cuando dos o más hilos se bloquean esperando que los otros liberen recursos, creando un círculo de espera del que no pueden salir.
- **Inconsistencia de Datos**: Sin una adecuada sincronización, los datos pueden corromperse o volverse inconsistentes.

Para sincronizar hilos utilizamos un bloque `synchronized`

```
synchronized(lock) {
    // código que solo puede ser ejecutado por un hilo a la vez
}
```

`lock` es un objeto único para esta sincronización que no se puede utilizar con otras `sychronized`
Ejemplo:

```
fun incrementarContador() {
    synchronized(this) {
        contador++
        println("Contador incrementado a $contador")
    }
}

// Creamos diez hilos
fun main() {
    val hilos = List(10) {
        Thread {
            incrementarContador()
        }
    }

    hilos.forEach { it.start() }
    hilos.forEach { it.join() } // Esperar a que todos los hilos terminen
}
```

Declaraciones de la librería de Threads.

```
inline fun <R> synchronized(lock: Any, block: () -> R): R
```

96.2.2 Procesos

Un proceso en Android es una instancia de la aplicación en ejecución. Cada aplicación corre en su propio proceso, aislada de otras aplicaciones, lo que mejora la seguridad y la estabilidad.

Manejo de Procesos:

Generalmente, no necesitas manejar procesos directamente en Android. El sistema operativo se encarga de crear y matar procesos según sea necesario para administrar los recursos.

Comunicación entre Procesos (IPC):

Si necesitas que las aplicaciones se comuniquen entre sí, puedes usar IPC (Inter-Process Communication), por ejemplo, mediante `Intents`, `ContentProviders` o `AIDL` (Android Interface Definition Language). Servicios:

Los servicios en Android son componentes que pueden ejecutarse en segundo plano, incluso cuando la interfaz de usuario no está activa. Pueden correr en el mismo proceso que la UI o en un proceso separado.

96.2.3 Timer

Un Timer es una forma de programar una tarea, conocida como TimerTask, para que se ejecute una vez o repetidamente después de un periodo de tiempo definido.

TimerTask es una clase que debes extender para definir el código que se ejecutará cuando el Timer se active. Hilos en Timer:

Las tareas programadas con Timer se ejecutan en un **hilo separado del hilo principal de la UI**, lo que significa que no debes actualizar la interfaz de usuario directamente desde un TimerTask.

En Kotlin y Compose es más adecuado usar Corrutinas para estas tareas.

Ejemplo de Timer:

```
val timer = Timer()
val timerTask = object : TimerTask() {
    override fun run() {
        // Código a ejecutar
    }
}
// Programa la tarea para que se ejecute después de un retraso
timer.schedule(timerTask, 1000) // Se ejecutará después de 1000 ms (1 segundo)

// Para una tarea periódica:
timer.scheduleAtFixedRate(timerTask, 1000, 5000) // Inicia después de 1s y se repite
cada 5s
```

96.3 Planificador de tareas Scheduler

El programador de tareas Scheduler en Android es una herramienta que sirve para planificar y ejecutar tareas de manera eficiente y efectiva en un momento específico o bajo ciertas condiciones.

Tipos de Schedulers en Android:

- **AlarmManager**: Permite ejecutar operaciones en momentos específicos, incluso cuando la aplicación no está en ejecución. Es útil para tareas que necesitan precisión en el tiempo, como alarmas o recordatorios.
- **JobScheduler**: Introducido en Android 5.0 (API nivel 21), está diseñado para tareas diferibles y que no requieren ejecución inmediata. Puedes definir criterios como la conectividad de red o el estado de

carga para ejecutar el trabajo.

- **WorkManager**: Una parte de Android Jetpack, WorkManager es la solución recomendada para tareas de fondo. Combina la facilidad de uso de JobScheduler con la compatibilidad hacia atrás, y es ideal para tareas que deben sobrevivir a reinicios del sistema o que requieren ejecución garantizada.

Recordar: **WorkManager** se prefiere cuando se trabaja con Compose.

Algunas observaciones:

- Para tareas que dependen de condiciones específicas del sistema o que deben sobrevivir a reinicio del sistema, JobScheduler o WorkManager son más adecuados.
- Operaciones en Segundo Plano y Red:

Las **coroutines** son ideales para operaciones en segundo plano que no necesitan ejecutarse en un momento específico o bajo condiciones específicas del sistema, como llamadas a APIs o procesamiento de datos.

- Uso Conjunto:

Puedes usar coroutines dentro de los trabajos gestionados por JobScheduler o WorkManager para beneficiarte de la facilidad de uso de las coroutines al realizar la tarea en sí.

Ejemplo de AlarmManager:

1. Establecer permisos en AndroidManifest.xml

```
<uses-permission android:name="android.permission.SET_ALARM"/>
```

1. Crear un Broadcast Receiver

```
class AlarmReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        // Lógica a ejecutar cuando se activa la alarma
        Toast.makeText(context, "Alarma activada!", Toast.LENGTH_SHORT).show()
    }
}
```

Registrar

```
<application ...>
    <receiver android:name=".AlarmReceiver"/>
</application>
```

1. Configurar la alarma

```
val alarmManager = getSystemService(Context.ALARM_SERVICE) as AlarmManager
val intent = Intent(this, AlarmReceiver::class.java)
val pendingIntent = PendingIntent.getBroadcast(this, 0, intent, 0)

// Establece la alarma para que se dispare después de 10 segundos
val triggerTime = System.currentTimeMillis() + 10 * 1000 // 10 segundos

// Para versiones de Android mayores a API 19, usa setExact()
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
```

```
    alarmManager.setExact(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent)
} else {
    alarmManager.set(AlarmManager.RTC_WAKEUP, triggerTime, pendingIntent)
}
```

96.4 Corrutinas y Flow

En este apartado se presenta una introducción a corrutinas y Flow. Podéis seguir con ellos documentos de Corrutinas y de Flow

Los `Flow` son **secuencias asíncronas** con inicialización diferida (Lazy cold stream) `Lazy` significa que no se generan datos si no se necesitan. Cuando se conecta un `recolector` (consumidor de los datos), recibe los datos generados. Si se conecta otro recolector al mismo Flow, vuelve a recibir los mismos datos generados desde el principio. Este es el comportamiento por defecto, que en algunas especializaciones cambia a `hot Flow`

Asíncrono porque los datos no están generados desde el principio y los valores se generean en cualquier momento (ejemplo peticiones web).
Los Flow tienen contexto de corrutina.

Ejemplo

```
// En un programa Kotlin

fun main() = runBlocking{

    makeFlow2()
        .filter{ it%2}
        .collect{ println(it)}
}


// de coleccion a flow
fun makeFlow() = listOf(1,2,3).asFlow()

fun makeFlow2() = flow<Int>{
    for(i in 1..10){
         emit(i)
         delay(700) // de corrutinas
    }
}

suspend fun productor(){

}
```

(Observar que la función flow() crea un objeto Flow)
En el main `makeFlow.collect()` bloquea la corrutina mientras existan datos en el Flow.
Con Flow podemos usar filtros similares a las colecciones (.filter etc)

96.4.1 Especializaciones de Flow

- StateFlow
- SharedFlow

- Channel
- CallbackFlow

StateFlow

StateFlow almacena un único valor. Es una especialización de SharedFlow

Características:

- Flow hot (caliente)
- almacena un único valor
- Lo emite nada mas subscribirse
- Es derivado de SharedFlow

Uso:

Ejemplo:

```
class StateView{
    private val _miEstado = MutableStateFlow()
    val miEstado =  _miEsdtado.asStateFlow()

    suspend fun actualiza(){
        while(true){
            _miEstado.value = _miEstado.value + 1
        }
    }

}

fun main() = runBlocking{
    val miView = StateView()
    // emitimos valores
    miView.actualiza()

    // recolectamos
    miView.miEstado.collect{ println(it)}
}
```

En la clase StateFlow declaramos la variable privada __miEstado como un Flow mutable. Al ser privado no se puede modificar fuera de la clase. Y miEstado como un Flow inmutable por lo que solo se puede leer fuera de la clase.
Los estados de StateView se gestionan mediante métodos de la clase.

En el método actualiza() se emiten los valores. Como StateFlow tiene la propiedad `.value` que realiza el `emit()`

Tal y como está el codigo anterior , nunca se ejecuta la última línea del main() porque el Flow no devuelve el control (queda en suspensión)

Tenemos que usar una nueva corrutina:

```
fun main() = runBlocking{
    val miView = StateView()
    // emitimos valores
```

```
    launch{
       miView.actualiza()
    }

    // recolectamos
    miView.miEstado.collect{ println(it)}
}
```

Cuando lo usamos en Compose, en lugar de `collect()` usamos una función delegada (de observador)

Por ejemplo, AppBarViewModel es el viewModel que contiene uiState

```
    val uiState = appBarViewModel.uiState.collectAsState()

// y lo usamos en funciones composables:

   Text( text= uiState.value.titulo)
```

96.5 SERVICE

Seguir en UT8.6 Servicios

96.6 Receptores de Broadcast

El tema se desarrolla en UT8.7

Un BroadcastReceiver es un componente de Android que responde a los mensajes de broadcast. Estos mensajes, conocidos como Intents, pueden ser emitidos por el sistema Android o por aplicaciones.

Los BroadcastReceiver son útiles para monitorear cambios en el sistema o para la comunicación entre diferentes componentes de la aplicación o entre distintas aplicaciones.

Tipos de Broadcasts:

- **Sistema**: Emitidos por el sistema Android, como `BOOT_COMPLETED` (cuando el dispositivo termina de arrancar), `ACTION_BATTERY_LOW` (cuando la batería está baja), `ACTION_LOCALE_CHANGED` (cuando cambia la configuración regional), entre otros.
- **Locales**: Broadcasts dentro de una aplicación, usando LocalBroadcastManager.
- **Personalizados**: Emitidos por aplicaciones para comunicarse entre sí o dentro de la misma aplicación.
- Declaración

```
class MyBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        // Reaccionar al broadcast
    }
}
```

1. Registro
2. Estáticamente en AndroidManifest.xml

```
<receiver android:name=".MyBroadcastReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

a. Dinámicamente en el código: Para registros más flexibles y controlados, especialmente para broadcasts relacionados con el ciclo de vida de la aplicación o la interfaz de usuario

b.
```
val receiver = MyBroadcastReceiver()
val filter = IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION)
registerReceiver(receiver, filter)
```

Recomendaciones: * Android ha introducido **restricciones** en el uso de BroadcastReceiver para ciertas acciones del sistema para mejorar el rendimiento y la seguridad. Por ejemplo, algunos broadcasts del sistema ya no se envían a receptores estáticos declarados en el AndroidManifest. * **No Realizar Operaciones Largas** en onReceive(): `onReceive()` se ejecuta en el hilo principal y tiene un tiempo de ejecución limitado; realizar operaciones largas puede causar problemas de rendimiento. * **Desregistro**: Si registras un BroadcastReceiver dinámicamente, asegúrate de anular su registro adecuadamente cuando ya no sea necesario, generalmente en el onPause() o onDestroy() de tu Activity o Fragment

96.7 Apendice

Versión 0.5 12-12-23 Versión 0.9 8-1-23