Universidad politécnica de chiapas

Asignatura:
Programacion para moviles I

Profesor:
Jose Alonso Macias Montoya

Actividad:
C2 - A4 - Implementa el uso de Android Profiler y LeakCanary

Alumno:
Jose Alberto Morales Solorzano 223190

Grado: 8 **Grupo:** “B”

Introducción

Las fugas de memoria son uno de los problemas más comunes y sutiles en el desarrollo de aplicaciones Android. Ocurren cuando objetos en memoria ya no son necesarios por la aplicación, pero el recolector de basura (Garbage Collector o GC) no puede liberarlos porque todavía existen referencias a ellos. Si no se abordan, las fugas de memoria pueden llevar a un consumo excesivo de memoria, degradación del rendimiento, e incluso cierres inesperados de la aplicación (Out of Memory Errors).

Este reporte explora dos herramientas esenciales para identificar y resolver fugas de memoria en Android:

- **Android Profiler:** Una herramienta integrada en Android Studio que proporciona información detallada sobre el uso de la CPU, memoria, red y batería de la aplicación. Permite inspeccionar el heap de la aplicación, rastrear asignaciones de memoria y detectar patrones sospechosos.
- **LeakCanary:** Una biblioteca de código abierto creada por Square que detecta automáticamente fugas de memoria en tiempo real durante el desarrollo. Proporciona notificaciones y un análisis detallado de la causa raíz de la fuga, facilitando la identificación y corrección del problema.

Este documento presentará ejemplos prácticos de cómo usar ambas herramientas para diagnosticar y solucionar diferentes tipos de fugas de memoria comunes en aplicaciones Android.

1. Android Profiler

El Android Profiler es una herramienta de rendimiento integrada en Android Studio que te permite medir el uso de recursos de tu aplicación. Es particularmente útil para identificar fugas de memoria al permitirle inspeccionar el heap de la aplicación y ver qué objetos están ocupando memoria.

Android Profiler: Identificación y Corrección de Fugas

Ejemplo: Fuga por Contexto de Actividad en un Singleton

Bloque de código con el error:

```
object MiSingleton {
    var contexto: Context? = null // ¡Peligro! Retiene el contexto de la actividad.
}

class MiActividad : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        MiSingleton.contexto = this // Asigna el contexto de la actividad al singleton
    }
}
```

- **Proceso de detección con Android Profiler:**
    - Ejecutar la aplicación y navegar a la actividad MiActividad.
    - Regresar de la actividad (simulando su destrucción).
    - Abrir Android Profiler -> Memory.

    - Tomar un "Heap Dump".
    - En el Heap Dump, buscar instancias de MiActividad. Observar que la instancia de MiActividad sigue presente en memoria, incluso después de haber sido destruida.
    - Analizar la "referencia" a la instancia de MiActividad. Se verá que está siendo retenida por MiSingleton.contexto.
- **Corrección:**
    - Evitar guardar el contexto de la actividad directamente. En su lugar, guardar el contexto de la aplicación (que tiene un ciclo de vida más largo).

```
object MiSingleton {
    var contexto: Context? = null
}

class MiActividad : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        MiSingleton.contexto = applicationContext // Usa el contexto de la aplicación
    }
}
```

Ejemplo : Fuga por Listener No Removido

Bloque de código con el error:

```
class MiActividad : AppCompatActivity() {

    private val miListener = object : MiObjeto.MiListener {
        override fun onCambio() {
            // Actualizar UI
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        MiObjeto.agregarListener(miListener) // Agrega el listener
    }
}

object MiObjeto {
    interface MiListener {
        fun onCambio()
    }

    private val listeners = mutableListOf<MiListener>()

    fun agregarListener(listener: MiListener) {
        listeners.add(listener)
    }
}
```

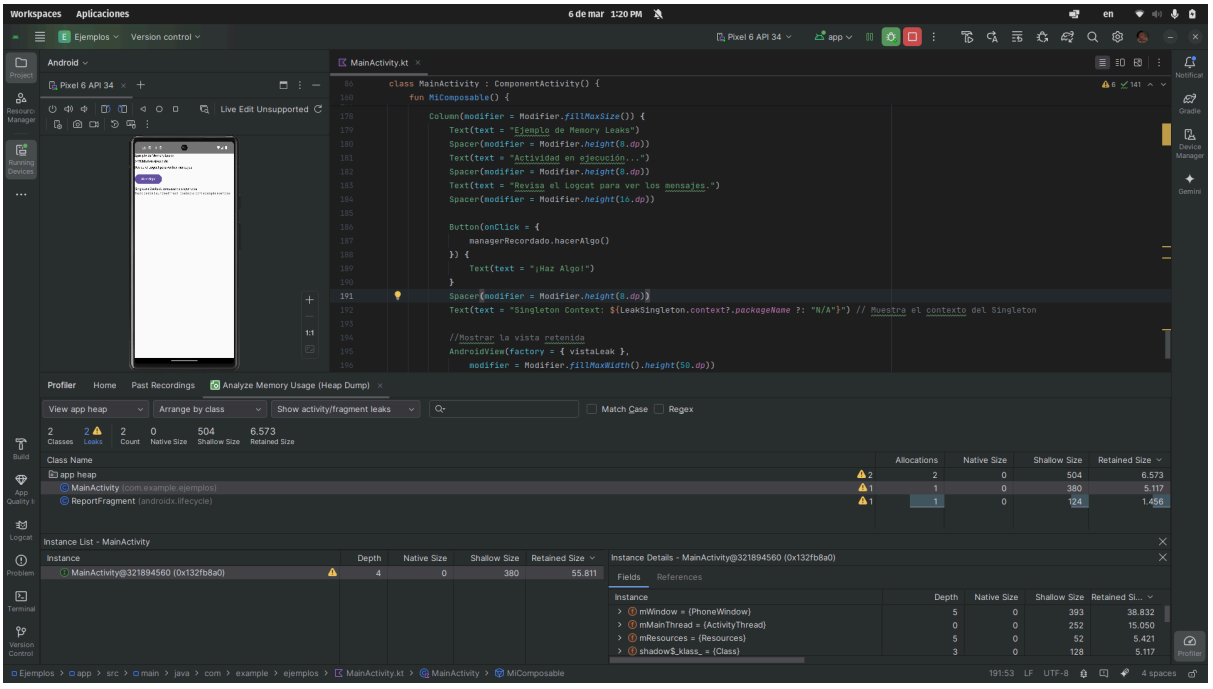
- **Proceso de detección con Android Profiler:**
    - Ejecutar la aplicación y abrir MiActividad varias veces, cerrándola después de cada vez.

- Tomar un Heap Dump en Android Profiler.
- Buscar múltiples instancias de MiActividad. Si el número de instancias sigue creciendo con cada apertura/cierre, hay una fuga.
- Analizar la referencia. Se observará que las instancias antiguas están siendo retenidas por la lista listeners en MiObjeto.

- **Corrección:**
  - Remover el listener en el método onDestroy de la actividad.

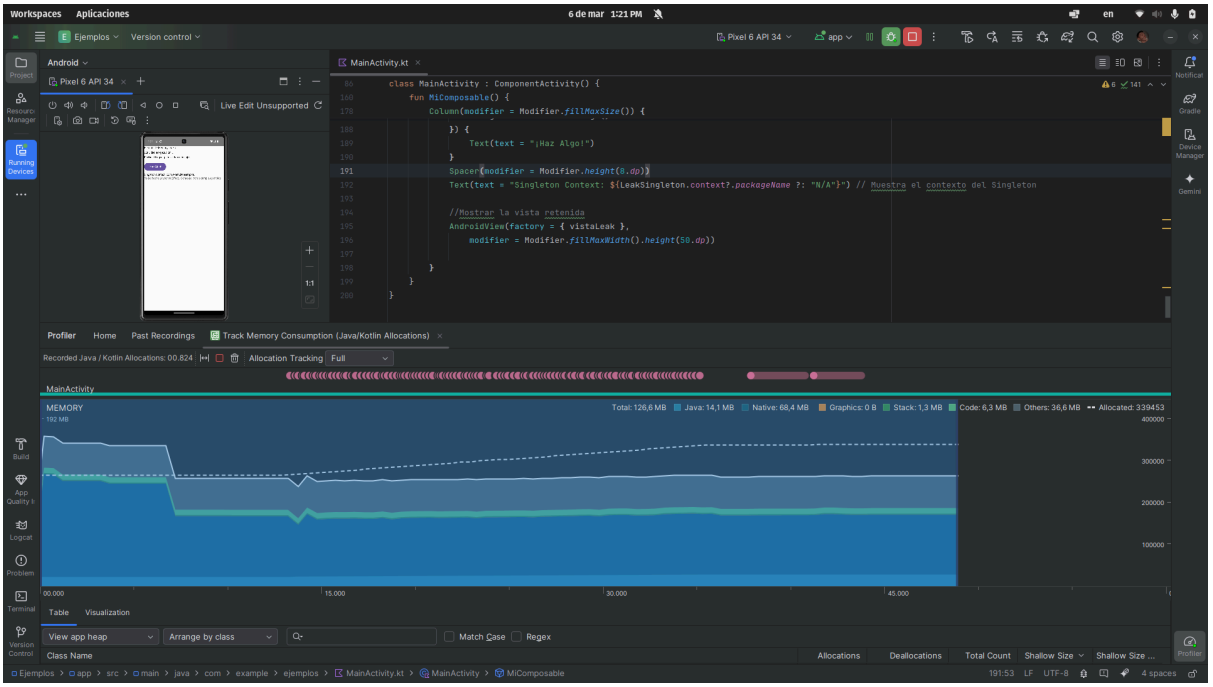
```
class MiActividad : AppCompatActivity() {
    // ... (Código anterior) ...

    override fun onDestroy() {
        super.onDestroy()
        MiObjeto.removerListener(miListener) // Remueve el listener al destruir la actividad
    }
}

object MiObjeto {
    // ... (Código anterior) ...

    fun removerListener(listener: MiListener) {
        listeners.remove(listener)
    }
}
```

Ejemplo práctico 1.1:



Aquí, en el Android Profiler, estoy analizando el Heap Dump que tomé. Puedo ver claramente una instancia de MainActivity, identificada como MainActivity@321894560(0x132fba0), que sigue en memoria a pesar de que ya debería haber sido destruida. Al examinar los 'Instance Details', encuentro que el campo mWindow mantiene una referencia a la Activity. Esto me dice que hay algo impidiendo que el recolector de basura la libere. Sospecho que el

problema está relacionado con el uso del Singleton (LeakySingleton) donde guardé el contexto de la Activity.



En esta vista del Android Profiler, estoy usando la función 'Track Memory Consumption'. El gráfico de memoria muestra cómo la cantidad de memoria asignada por mi aplicación está aumentando con el tiempo. Este patrón ascendente me preocupa, ya que sugiere que podría haber una acumulación de memoria, lo que es un indicio de una fuga. Las marcas rojas en el gráfico (aunque no se ven bien en esta captura) me indicarían los momentos en que el recolector de basura intentó liberar memoria sin éxito. Para profundizar en esto, planeo usar la función 'Allocation Tracking' para identificar qué objetos son los responsables de este aumento en el uso de memoria. También voy a tomar un Heap Dump para analizar las referencias y determinar la causa raíz de esta fuga.

2. LeakCanary

LeakCanary es una potente biblioteca que simplifica la detección de fugas de memoria en aplicaciones Android. Se integra fácilmente en tu proyecto y detecta automáticamente cuando una actividad o fragmento ha sido destruido pero aún está en memoria.

LeakCanary: Identificación y Corrección de Fugas

Ejemplo: Fuga por Referencia en un Hilo (Thread) en Ejecución

Bloque de código con el error:

```
class MiActividad : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        Thread {
            while (true) {
                // Hacer algo
                Thread.sleep(1000)
                Log.d("Fuga", "Actividad: ${this}") // ¡Referencia a la actividad!
            }
        }.start()
    }
}
```

- **Proceso de detección con LeakCanary:**
    - Agregar LeakCanary al proyecto (como lo has hecho en tu ejemplo).
    - Ejecutar la aplicación y abrir/cerrar MiActividad.
    - LeakCanary detectará la fuga y mostrará una notificación.
    - La notificación te llevará a un reporte detallado de la fuga, indicando que el Thread está reteniendo una referencia a la instancia de MiActividad.
- **Corrección:**
    - Evitar referenciar la actividad directamente dentro del hilo. Utilizar un WeakReference o un mecanismo similar para evitar que el hilo impida la liberación de la memoria de la actividad.

```
class MiActividad : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val referenciaDebil = WeakReference(this)

        Thread {
            while (true) {
                val actividad = referenciaDebil.get()
                if (actividad != null) {
                    Log.d("Fuga", "Actividad (Weak): ${actividad}")
                } else {
                    break // Salir del hilo si la actividad ya no existe
                }
                Thread.sleep(1000)
            }
        }.start()
    }
}
```

Ejemplo: Fuga por View Adjunta a una Actividad Destruida

Bloque de código con el error:

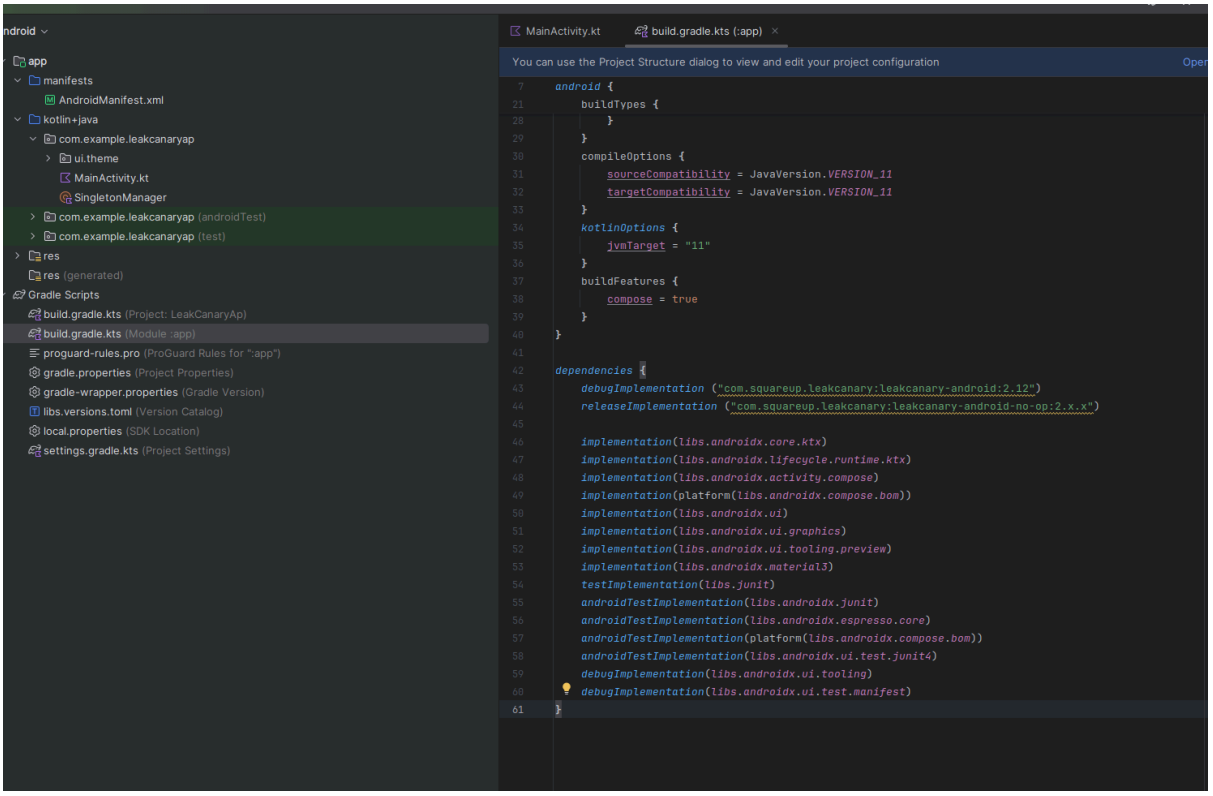
```
class MiActividad : AppCompatActivity() {

    private lateinit var miVista: View

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        miVista = View(this) // Crear la vista con el contexto de la actividad
        window.decorView.postDelayed({
            window.decorView.addView(miVista)
        }, 5000)
    }
}
```

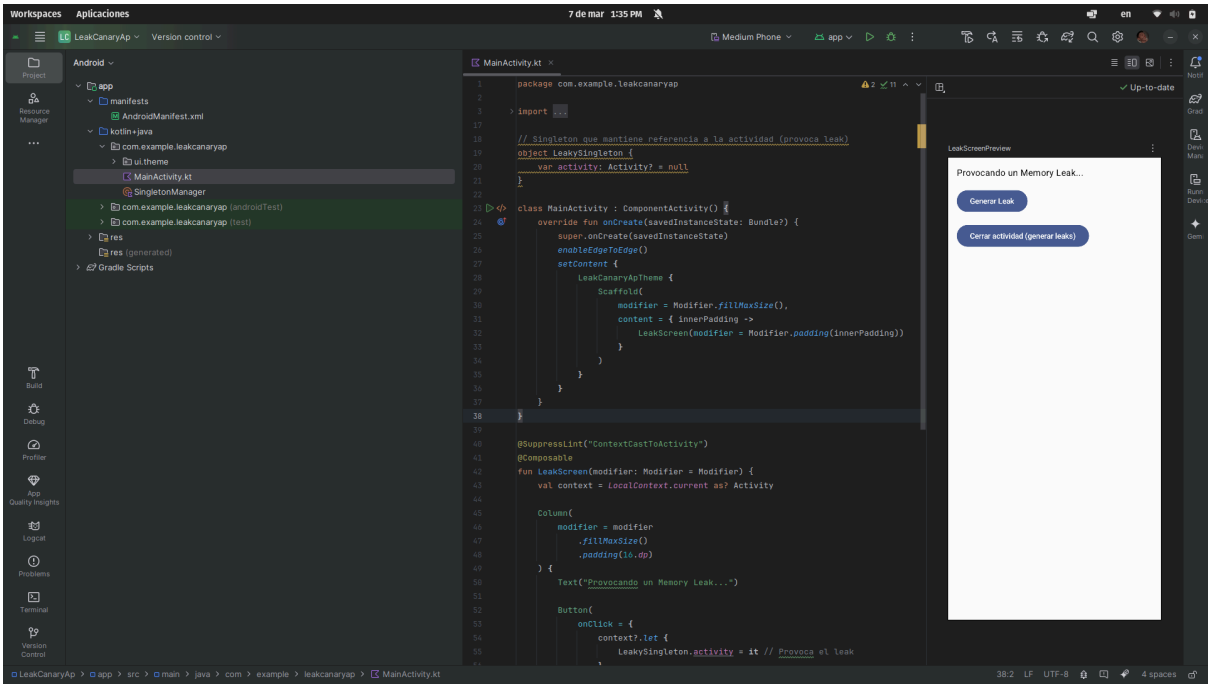
- **Proceso de detección con LeakCanary:**
  - Ejecutar la aplicación y abrir MiActividad.
  - Cerrar MiActividad antes de que el postDelayed se ejecute.
  - LeakCanary detectará que la vista (miVista) está siendo retenida por la vista raíz de la actividad (incluso después de que la actividad ha sido destruida).
- **Corrección:**
  - Remover la vista del decorView en el método onDestroy de la actividad.

```
class MiActividad : AppCompatActivity() {

    private lateinit var miVista: View

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        miVista = View(this) // Crear la vista con el contexto de la actividad
        window.decorView.postDelayed({
            window.decorView.addView(miVista)
        }, 5000)
    }

    override fun onDestroy() {
        super.onDestroy()
        window.decorView.removeView(miVista)
    }
}
```

Ejemplo práctico 2.1:

Configuracion previa ,en el gradle app:





Explicación:

Diseñé este pequeño proyecto en Kotlin para mostrar cómo LeakCanary ayuda a detectar fugas de memoria. Creé un singleton llamado LeakySingleton donde guardo una referencia a la Activity. La idea es simple: la interfaz, hecha con Compose, tiene un botón que 'atrapa' la Activity en el singleton y otro que la cierra. Al cerrar la Activity después de 'atraparla', LeakCanary debería saltar y avisar que la memoria no se libera porque el singleton sigue apuntando a ella. Es una forma sencilla de simular un memory leak y ver LeakCanary en acción.

Ejecucion en Telefono movil: