Universidad Politécnica de Chiapas

Ingeniería en Tecnologías de la Información e Innovación Digital

[Programacion Para moviles]

[C2 - A6 - Implementa el uso de Android Profiler o LeakCanary]

[Nomenclatura del nombre de archivo C2 - A6 - Implementa el uso de Android Profiler o LeakCanary-223216-Daniel Peregrino Perez.pdf]

[Alumno -Daniel Peregrino Perez] - [223216]

Docente: [José Alonso Macias Montoya]

Fecha de entrega: [27/06/2025]

1. DESCRIPCIÓN DE LA ACTIVIDAD

1.1. Enunciado del problema

Desarrollar dos aplicaciones móviles nativas para Android utilizando Android Studio y el lenguaje Kotlin para demostrar la detección y corrección de fugas de memoria (Memory Leaks).

1. **Primer Ejemplo:** Crear una aplicación con un Memory Leak causado por una referencia implícita en un Handler anónimo. El objetivo es utilizar el **Android Profiler** para detectar la fuga, analizar su causa y corregirla.

2. **Segundo Ejemplo:** Crear una segunda aplicación con un Memory Leak causado por una referencia estática a una Activity. El objetivo es integrar la librería **LeakCanary** para detectar automáticamente la fuga, interpretar su reporte y aplicar la corrección necesaria.

1.2. Objetivos de aprendizaje

- Comprender el concepto de Memory Leak y su impacto en el rendimiento de una aplicación Android.
- Aprender a utilizar el **Android Profiler** para analizar el uso de memoria, forzar la recolección de basura y generar *heap dumps*.
- Analizar un *heap dump* para identificar objetos que no han sido liberados y rastrear la cadena de referencias que los retiene.
- Implementar y configurar la librería **LeakCanary** para la detección automática de fugas de memoria.
- Interpretar los reportes generados por LeakCanary para identificar la causa raíz de una fuga.
- Aplicar soluciones comunes para corregir Memory Leaks, como el uso de clases anidadas estáticas, WeakReference y la limpieza de referencias en los métodos del ciclo de vida.

2. FUNDAMENTOS TEÓRICOS

- **Memory Leak (Fuga de Memoria):** Ocurre cuando un objeto ya no es necesario para la aplicación, pero el Recolector de Basura (Garbage Collector) no puede eliminarlo de la memoria porque todavía existen referencias fuertes hacia él desde otros objetos activos. Esto provoca un consumo de memoria innecesario que puede degradar el rendimiento y, en casos graves, causar un error de OutOfMemoryError.
- **Android Profiler:** Es un conjunto de herramientas dentro de Android Studio que proporciona datos en tiempo real sobre el uso de CPU, memoria, red y energía de una aplicación. El Memory Profiler, en particular, permite visualizar la asignación de objetos, detectar fugas de memoria, forzar la recolección de basura y capturar instantáneas del heap (memoria de Java) para un análisis detallado.
- **LeakCanary:** Es una librería de detección de fugas de memoria para Android. Se integra en la aplicación (generalmente en builds de depuración) y monitorea automáticamente los objetos que deberían ser destruidos (como Activities y Fragments). Si un objeto no es recolectado por el Garbage Collector después de un tiempo prudencial, LeakCanary genera un *heap dump*, lo analiza y muestra una notificación con la "traza de la fuga", que es la cadena de referencias que impide que el objeto sea liberado, facilitando enormemente su corrección.

3. DESARROLLO DE LA ACTIVIDAD

3.1. Desarrollo

Se crearon dos proyectos independientes para aislar cada caso de estudio.

3.1. Ejemplo 1: Detección con Android Profiler

1. **Configuración y Código con Error:**
    - Se creó un proyecto llamado ProfilerLeakDemo.
    - En MainActivity, se implementó un Handler como una clase anónima para ejecutar una tarea con un retraso de 30 segundos. Al ser una clase interna no estática, este Handler mantiene una referencia implícita a MainActivity. Si la actividad se destruye (ej. al rotar la pantalla) antes de que se complete el retraso, el Handler sigue en la cola de mensajes, reteniendo la instancia de MainActivity en memoria y causando una fuga.

// BLOQUE DE CÓDIGO CON EL ERROR

```
pr
ivate val leakyHandler = object : Handler(Looper.getMainLooper()) { // Mantiene referencia a MainActivity
    override fun handleMessage(msg: Message) {
        // ...
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // ...
    leakyHandler.postDelayed({ /* ... */ }, 30000) // Tarea de larga duración
}
```

2. **Corrección del Código:**
    - La solución fue convertir el Handler en una clase anidada estática (private class en Kotlin) para eliminar la referencia implícita.
    - Se utilizó una WeakReference para permitir el acceso a MainActivity de forma segura, sin impedir que sea recolectada por el Garbage Collector.
    - Adicionalmente, se añadió removeCallbacksAndMessages(null) en el método onDestroy() de la actividad para limpiar cualquier tarea pendiente del Handler, una práctica recomendada para prevenir ejecuciones no deseadas y ayudar a liberar recursos.

Generated kotlin

```
// BLOQUE DE CÓDIGO CORREGIDO
private class SafeHandler(activity: MainActivity) : Handler(Looper.getMainLooper()) {
    private val activityReference: WeakReference<MainActivity> = WeakReference(activity)

    override fun handleMessage(msg: Message) {
        val activity = activityReference.get()
        if (activity != null && !activity.isFinishing) {
            // ... interactuar con la actividad de forma segura
        }
    }
}

override fun onDestroy() {
    super.onDestroy()
    safeHandler.removeCallbacksAndMessages(null) // Limpiar tareas pendientes
}
```

3.2. Ejemplo 2: Detección con LeakCanary

1. **Configuración y Código con Error:**
    - Se creó un proyecto llamado LeakCanaryDemo y se añadió la dependencia de LeakCanary en el archivo build.gradle.
    - Se creó una LeakyActivity que, en su companion object (equivalente a estático en Java), tenía una variable que almacenaba la propia instancia de la actividad.

- Esta referencia estática nunca se limpiaba, por lo que incluso después de que LeakyActivity se cerrara con finish(), la referencia estática la mantenía viva en memoria.

```
// BLOQUE DE CÓDIGO CON EL ERROR
class LeakyActivity : AppCompatActivity() {
    companion object {
        var leakyActivityInstance: LeakyActivity? = null // Referencia estática
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        leakyActivityInstance = this // Se asigna la instancia, pero nunca se limpia
    }
}
```

2. **Corrección del Código:**
    - La corrección fue muy sencilla: consistió en anular la referencia estática en el momento en que la actividad se destruye.
    - En el método onDestroy() de LeakyActivity, se estableció la variable activityInstance a null, rompiendo la referencia estática y permitiendo que el Garbage Collector liberara la instancia de la actividad.

Generated kotlin

```
// BLOQUE DE CÓDIGO CORREGIDO
override fun onDestroy() {
    super.onDestroy()
    // Limpiar la referencia estática para permitir la recolección de basura
    if (activityInstance == this) {
        activityInstance = null
    }
}
```

ProfilerLeakActivity.kt | MainActivity.kt | activity_main.xml | build.gradle.kts (:app)

```kotlin
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    private lateinit var leakyTextView: TextView

    // INCORRECTO: Handler como clase interna (o anónima)
    // Mantiene una referencia implícita a MainActivity
    private val leakyHandler = object : Handler(Looper.getMainLooper()) {
        override fun handleMessage(msg: Message) {
            super.handleMessage(msg)
            // Imaginemos que aquí actualizamos la UI
            if (::leakyTextView.isInitialized) { // Solo para evitar crash si ya se destruyó la vista
                leakyTextView.text = "Handler ejecutado! Leak!"
                Log.d( tag: "ProfilerLeak",  msg: "Handler ejecutado en MainActivity")
            }
        }
    }

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

        leakyTextView = findViewById(R.id.textViewLeak)
        leakyTextView.text = "Esperando al Handler..."

        // Enviar un mensaje con un retraso largo
        // Si rotamos o cerramos la actividad antes de 30s, habrá un leak
        leakyHandler.postDelayed({
            leakyHandler.sendEmptyMessage( what: 0)
            Log.d( tag: "ProfilerLeak",  msg: "Mensaje enviado por el Handler")
        }, delayMillis: 30000) // 30 segundos de retraso
        Log.d( tag: "ProfilerLeak",  msg: "MainActivity onCreate")
    }
```

motorola moto g31 | app

ProfilerLeakActivity.kt | MainActivity.kt | activity_main.xml | build.gradle.kts (:app)

```kotlin
import android.os.Bundle
import android.os.Handler
import android.os.Looper
import android.os.Message
import android.util.Log
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity
import com.example.memoryleakdemo.R
import java.lang.ref.WeakReference // Importar WeakReference

class MainActivity : AppCompatActivity() {

    private lateinit var currentTextView: TextView // Renombrado para claridad

    // CORRECTO: Handler como clase anidada estática
    private class SafeHandler(activity: MainActivity) : Handler(Looper.getMainLooper()) {
        // Usar WeakReference para evitar leaks de Activity
        private val activityReference: WeakReference<MainActivity> = WeakReference(activity)

        override fun handleMessage(msg: Message) {
            super.handleMessage(msg)
            val activity = activityReference.get() // Obtener la referencia a la Activity
            if (activity != null && !activity.isFinishing) {
                // Solo interactuar si la Activity aún existe y no se está finalizando
                activity.currentTextView.text = "Handler ejecutado de forma segura!"
                Log.d( tag: "ProfilerLeak",  msg: "Handler ejecutado de forma segura en MainActivity")
            } else {
                Log.d( tag: "ProfilerLeak",  msg: "Handler ejecutado, pero Activity ya no está disponible.")
            }
        }
    }

    private lateinit var safeHandler: SafeHandler // Declarar la instancia del handler
```

```kotlin
import androidx.appcompat.app.AppCompatActivity

class LeakyActivity : AppCompatActivity() {

    companion object {
        // INCORRECTO: Referencia estática a la Activity
        var leakyActivityInstance: LeakyActivity? = null
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_leaky)
        Log.d( tag: "LeakCanaryDemo",  msg: "LeakyActivity onCreate")

        // Guardar la instancia en la variable estática
        leakyActivityInstance = this

        findViewById<Button>(R.id.button_finish_leaky).setOnClickListener {
            Log.d( tag: "LeakCanaryDemo",  msg: "Finalizando LeakyActivity...")
            finish() // Esto debería permitir que la Activity sea recolectada
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d( tag: "LeakCanaryDemo",  msg: "LeakyActivity onDestroy. Leaky instance: $leakyActivityInstance")
        // NO estamos limpiando leakyActivityInstance = null aquí, causando el leak.
    }
}
```

4. RESULTADOS

4.1. Resultados obtenidos

- **Proceso de Detección:**

Se ejecutó la aplicación ProfilerLeakDemo, se abrió el Memory Profiler y se rotó el dispositivo varias veces para crear y destruir instancias de MainActivity. Después de forzar la recolección de basura, se capturó un *heap dump*. El análisis del dump mostró claramente múltiples instancias de MainActivity que persistían en memoria, confirmando la fuga.

MemoryLeakDemo · Version control · motorola moto g31 · app

Android

app
manifests
AndroidManifest.xml
kotlin+java
com.example.memoryleakdemo
ui.theme
CanaryLeakActivity
MainActivity

ProfilerLeakActivity.kt · MainActivity.kt · activity_main.xml · build.gradle.kts (:app)

```
7   android {
40  }
41
42  dependencies {
43
44      implementation(libs.androidx.core.ktx)
45      implementation(libs.androidx.lifecycle.runtime.ktx)
46      implementation(libs.androidx.activity.compose)
47      implementation(platform(libs.androidx.compose.bom))
```

Profiler · Home · Past Recordings · View Live Telemetry

MainActivity - stopped - saved - destroyed · MainActivity

LIVE_CPU — App: 0 % · Others: 7 % · Threads: 44
100 % · 50 % · 50 · 40 · 30 · 20 · 10

THREADS (44)
.memoryleakdemo
RenderThread

MEMORY — Total: 79 MB · Java: 10.2 MB · Native: 9.1 MB · Graphics: 15.8 MB · Stack: 1.1 MB · Code: 30.7 MB · Others: 12.1 MB · Allocated: N/A
96 MB · 64 MB · 32 MB

02:40.000 · 02:45.000 · 02:50.000 · 02:55.000 · 03:00.000 · 03:05.000

MemoryLeakDemo > app > build.gradle.kts > android · 39:6 · LF · UTF-8 · 4 spaces

Profiler · Home · Past Recordings · View Live Telemetry

MainActivity

LIVE_CPU — App: 0 % · Others: 10 % · Threads: 43
100 % · 50 % · 30 · 20 · 10

THREADS (43)
.memoryleakdemo

MEMORY — Total: 102.2 MB · Java: 10.8 MB · Native: 10.4 MB · Graphics: 15.8 MB · Stack: 1020 KB · Code: 49.5 MB · Others: 14.7 MB · Allocated: N/A
128 MB

00:43.649
Others: 14.7 MB
Code: 49.5 MB
Stack: 1008 KB
Graphics: 22.4 MB
Native: 10.3 MB
Java: 10.8 MB
Allocated: N/A
Total: 108.7 MB

40.000 · 50.000 · 55.000 · 01:00.000 · 01:05.000

MemoryLeakDemo > app > src > main > java > com > example > memoryleakdemo > MainActivity · 5:26 · LF · UTF-8 · 4 spaces

themes.xml

Profiler · Home · Past Recordings · View Live Telemetry

LeakyActivity - ... · MainActivity - stopped - saved · LeakyAct... · LeakyActivit... · LeakyActivity - st... · MainActivity

LIVE_CPU — App: 2 % · Others: 57 % · Threads: 46
100 % · 40 · 30

MEMORY — Total: 127.4 MB · Java: 33.6 MB · Native: 11.7 MB · Graphics: 9.1 MB · Stack: 1.1 MB · Code: 46.6 MB · Others: 25.3 MB · Allocated: N/A
96 MB

05.000 · 10.000 · 15.000 · 20.000 · 25.000 · 30.000

- **Análisis de la Causa:**
  Al seleccionar una de las instancias "fugadas" de MainActivity, el panel "References" mostró la cadena de retención. Se pudo observar que la instancia era retenida por la variable this$0 dentro de una instancia del leakyHandler.

- **Verificación de la Corrección:**
  Tras aplicar el código corregido, se repitió el proceso. El nuevo *heap dump* mostró que, después de rotar y forzar la recolección de basura, solo quedaba una instancia activa de MainActivity (la visible en pantalla), demostrando que la fuga había sido solucionada.

4.2. Resultados con LeakCanary

- **Proceso de Detección:**
  Se ejecutó la aplicación LeakCanaryDemo. Se navegó a LeakyActivity y luego se presionó el botón para cerrarla. Pocos segundos después, LeakCanary mostró una notificación en el dispositivo indicando que se había detectado una fuga de memoria.

6:38

just now

Explore Heap Dump

Share Heap Dump analysis

Print analysis to Logcat (tag: LeakCanary)

Share Heap Dump file

See Metadata

1 Distinct Leak

1 LeakyActivity.leakyActivityIns...
just now
New

6:39

just now

Explore Heap Dump

Metadata

Build.VERSION.SDK_INT: 31
Build.MANUFACTURER: motorola
LeakCanary version: 2.12
App process name: com.example.leakcanarydemo
Class count: 25590
Instance count: 204659
Primitive array count: 146740
Object array count: 29884
Thread count: 22
Heap total bytes: 26151602
Bitmap count: 2
Bitmap total bytes: 129782
Large bitmap count: 0
Large bitmap total bytes: 0
Stats: LruCache[maxSize=3000,hits=114808,misses=196552,hitRate=36%] RandomAccess[bytes=9663996,reads=196552,travel=65013463879,range=32222849,size=39827886]
Analysis duration: 43379 ms
Heap dump file path: /storage/emulated/0/Download/leakcanary-com.example.leakcanarydemo/2025-06-25_18-37-07_454.hprof
Heap dump timestamp: 1750898280509
...on: Unknown

ACEPTAR

6:38

1 leak at LeakyActivity.leakyActivi...

LeakyActivity Leaked just now

New

Share leak trace as text or on Stack Overflow

Print leak trace to Logcat (tag: LeakCanary)

Share Heap Dump file

References **underlined** are the likely causes of the leak. Learn more at https://squ.re/leaks

GC Root: System class

android.os.StrictMode$InstanceTracker class
Leaking: NO (LeakyActivity↓ is not leaking and a class is never leaking)
static StrictMode$InstanceTracker.*sInstanceCounts*

java.util.HashMap instance
Leaking: NO (LeakyActivity↓ is not leaking)
HashMap[key()]

com.example.leakcanarydemo.LeakyActivity class
Leaking: NO (a class is never leaking)
static LeakyActivity.*leakyActivityInstance*

com.example.leakcanarydemo.LeakyActivity instance
Leaking: YES (ObjectWatcher was watching this because com.example.leakcanarydemo.LeakyActivity received Activity#onDestroy() callback and Activity#mDestroyed is true)
Retaining 181.2 kB in 1668 objects
key = 541fbcfb-9338-4e8d-a344-606b9cad427d
watchDurationMillis = 5710

- **Análisis de la Causa:**
  Al abrir la notificación, LeakCanary presentó una pantalla con la traza de la fuga. El reporte indicó inequívocamente que LeakyActivity estaba siendo retenida por la referencia estática LeakyActivity.Companion.leakyActivityInstance.

[Espacio para captura de pantalla de la traza de fuga de LeakCanary mostrando la referencia estática]

- **Verificación de la Corrección:**
  Con el código corregido (limpiando la referencia en onDestroy()), se repitió el proceso. Esta vez, después de cerrar LeakyActivity, LeakCanary no generó ninguna notificación, confirmando que la fuga había sido eliminada exitosamente.

---

5. CONCLUSIONES

A través de esta actividad práctica, se logró un entendimiento profundo sobre la detección y corrección de Memory Leaks en Android.

- **El Android Profiler se demostró como una herramienta potente y esencial para el análisis manual y detallado de la memoria, aunque requiere un proceso proactivo por parte del desarrollador.**

- **LeakCanary probó ser una herramienta invaluable para el desarrollo diario, ya que automatiza la detección de fugas comunes con una configuración mínima, permitiendo identificar y corregir problemas de manera temprana y eficiente.**

Se consolidaron buenas prácticas de programación, como la importancia de gestionar correctamente el ciclo de vida de los componentes, evitar referencias fuertes a Contexts o Activities desde objetos con un ciclo de vida más largo, y el uso de patrones como WeakReference para prevenir fugas. La actividad reafirmó que mantener una aplicación libre de fugas de memoria es fundamental para garantizar su estabilidad y una buena experiencia de usuario.