*Học viện Kỹ Thuật Mật Mã*

**-----\*\*\*-----**



**-----\*\*\*-----**

**BÁO CÁO BÀI TẬP LỚN TỐI ƯU PHẦN MỀM DI ĐỘNG**

**Đề tài: Nghiên cứu về việc tối ưu bộ nhớ**

|  |  |
| --- | --- |
| GIÁO VIÊN HƯỚNG DẪN: | **Th.S Lê Bá Cường** |

Sinh viên thực hiện: Mai Quang Vinh – CT050154

**MỤC LỤC**

[**CHƯƠNG 1: TỔNG QUAN TỐI ƯU PHẦN DI ĐỘNG** 6](#_Toc187587893)

[**1.1. Hệ điều hành Android**  6](#_Toc187587894)

[**1.2. Tổng quan về tối ưu hóa trong lập trình di động** 7](#_Toc187587895)

[**1.3. Một vài nguyên nhân ảnh hưởng đến hiệu xuất của ứng dụng** 9](#_Toc187587896)

[**1.3.1 Slow Rendering** 9](#_Toc187587897)

[**1.3.2 Layout** 10](#_Toc187587898)

[**1.3.3 Power Usage** 12](#_Toc187587899)

[**1.3.4 Multithreading** 12](#_Toc187587900)

[**1.3.5 Memory** 13](#_Toc187587901)

[**CHƯƠNG 2: TỐI ƯU BỘ NHỚ TRONG LẬP TRÌNH ỨNG DỤNG ANDROID** 15](#_Toc187587902)

[**2.1. Cơ chế cấp phát và giải phóng bộ nhớ trong Android** 15](#_Toc187587903)

[**2.1.1 Memory Mapping, Dalvik Virtual Machine, Garbage Collection** 16](#_Toc187587904)

[**2.1.2 Cơ chế cấp phát và giải phóng bộ nhớ** 22](#_Toc187587905)

[**2.1.3 Hạn chế bộ nhớ ứng dụng** 24](#_Toc187587906)

[**2.1.4 Quản lý chuyển đổi ứng dụng** 25](#_Toc187587907)

[**2.2. Các phương pháp tối ưu bộ nhớ** 26](#_Toc187587908)

[**2.2.1 Sử dụng services đúng cách** 27](#_Toc187587909)

[**2.2.2 Giải phóng bộ nhớ khi giao diện bị ẩn** 28](#_Toc187587910)

[**2.2.3 Xử lí bộ nhớ khi thiếu tài nguyên** 29](#_Toc187587911)

[**2.2.4 Kiểm soát sử dụng bộ nhớ tối ưu** 30](#_Toc187587912)

[**2.2.5 Tối ưu hóa Bitmap** 31](#_Toc187587913)

[**2.2.6 Sử dụng cấu trúc dữ liệu hiệu quả** 31](#_Toc187587914)

[**2.2.7 Hạn chế chi phí bộ nhớ không cần thiết** 32](#_Toc187587915)

[**2.2.8 Lưu ý khi trừu tượng mã hóa nguồn** 33](#_Toc187587916)

[**2.2.9 Quản lí khung phụ thuộc và thư viện bên ngoài** 34](#_Toc187587917)

[**2.2.10 Sử dụng ProGuard để tối ưu mã nguồn** 35](#_Toc187587918)

[**2.3. Vai trò proguard trong tối ưu bộ nhớ** 35](#_Toc187587919)

[**2.3.1 Giới thiệu về ProGuard** 35](#_Toc187587920)

[**2.3.2 Lợi ích khi sử dụng ProGuard** 36](#_Toc187587921)

[**2.3.3 Cách tích hợp ProGuard trong ứng dụng** 37](#_Toc187587922)

[**2.4. Các công cụ tối ưu bộ nhớ** 39](#_Toc187587923)

[**2.4.1 MAT ( Memory Analyzer Tool)** 39](#_Toc187587924)

[**2.4.2 JetBrains JVM Debugger Memory View** 40](#_Toc187587925)

[**2.4.3. Netbeans Profiler** 41](#_Toc187587926)

[**2.5. Tìm hiểu công cụ Leak Canary** 42](#_Toc187587927)

[**2.5.1 Giới thiệu Leak Canary** 42](#_Toc187587928)

[**2.5.2 Cơ chế hoạt động Leak Canary** 46](#_Toc187587929)

[**2.5.3 Cấu hình LeakCanary trong ứng dụng Android** 47](#_Toc187587930)

[**2.5.4 Phát hiện và xử lí lỗi rò rỉ bộ nhớ** 49](#_Toc187587931)

[**2.5.5 Hướng dẫn sử dụng Leak Canary** 49](#_Toc187587932)

[**CHƯƠNG 3: PHÂN TÍCH VÀ TIẾN HÀNH THỰC NGHIỆM CÁC TRƯỜNG HỢP LEAK MEMORY** 54](#_Toc187587933)

[**3.1. AsyncTaskActivity** 54](#_Toc187587934)

[**3.2. HandlerLeak** 55](#_Toc187587935)

[**3.3. StaticAsyncTaskActivity** 56](#_Toc187587936)

[**3.4. ThreadActivity** 59](#_Toc187587937)

[**TÀI LIỆU THAM KHẢO** 62](#_Toc187587938)

**DANH MỤC HÌNH ẢNH**

[*Hình 1. 1 Kiến trúc hệ điều hành android*](about:blank) 6

[*Hình 1. 2 Slow Rendering* 9](#_heading=h.4d34og8)

[*Hình 1. 3 Dropped frame* 9](#_heading=h.2s8eyo1)

[*Hình 1. 4 Xử lý luồng trong ứng dụng* 12](#_heading=h.lnxbz9)

[*Hình 2. 1 Cơ chế memory mapping* 15](#_heading=h.z337ya)

[*Hình 2. 2 Cấu trúc Dalvik virtual machine* 16](#_heading=h.3j2qqm3)

[*Hình 2. 3 Sơ đồ hoạt động của Garbage Collection* 17](#_heading=h.1y810tw)

[*Hình 2. 4 Cơ chế cấp phát và giải phóng bộ nhớ* 19](#_heading=h.2xcytpi)

[*Hình 2. 5 Cấu hình build grandle* 28](#_heading=h.1v1yuxt)

[*Hình 2. 6 Ví dụ Static Activities* 30](#_heading=h.37m2jsg)

[*Hình 2. 7 Ví dụ Static Views* 31](#_heading=h.1mrcu09)

[*Hình 2. 8 Ví dụ Non static Inner Classes* 31](#_heading=h.46r0co2)

[*Hình 2. 9 Ví dụ Handler và Asynctask* 32](#_heading=h.2lwamvv)

[*Hình 3. 1 Memory leack khi dùng Async Task* 39](#_heading=h.3ygebqi)

[*Hình 3. 2 Khắc phục memory leack khi dùng AsyncTask* 40](#_heading=h.2dlolyb)

[*Hình 3.3 Memory leack khi dùng Handler* 40](#_heading=h.3cqmetx)

[*Hình 3. 4 Khắc phục memory leack khi dùng Handler* 41](#_heading=h.1rvwp1q)

[*Hình 3. 5 Memory leack khi sử dụng Static Async Task Activity* 42](#_heading=h.2r0uhxc)

[*Hình 3. 6 Khác phục memory leack khi sử dụng Static Async Task Activity* 43](#_heading=h.1664s55)

[*Hình 3. 7 Memory leack khi sử dụng Thread Activity* 44](#_heading=h.25b2l0r)

[*Hình 3. 8 Khắc phục memory leack khi sử dụng Thread Activity* 45](#_heading=h.kgcv8k)

**LỜI MỞ ĐẦU**

Hiện nay Công nghệ thông tin vô cùng phát triển thì mọi người đều sử dụng máy tính vi tính hoặc điện thoại di động để làm việc và việc cập nhật thông tin. Do đó việc xây dựng các ứng dụng cho điện thoại di động đang là ngành công nghiệp mới đầy tiềm năng và hứa hẹn nhiều sự phát triển vượt bậc của ngành khoa học kỹ thuật.

Phần mềm, ứng dụng cho điện thoại di động hiện nay rất đa dạng và phong phú trên các hệ điều hành di động. Các hệ điều hành J2ME, Android, IOS, Web bases Mobile Application đã rất phát triển trên thị trường truyền thông di động.

Trong vài năm trở lại đây, hệ điều hành android ra đời với sự kế thừa những ưu việt của các hệ điều hành ra đời trước và sự kết hợp của nhiều công nghệ tiên tiến nhất hiện nay. Android đã nhanh chóng là đối thủ cạnh tranh mạnh mẽ với các hệ điều hành trước đó và đang là hệ điều hành di động của tương lai và được nhiều người ưa chuộng.

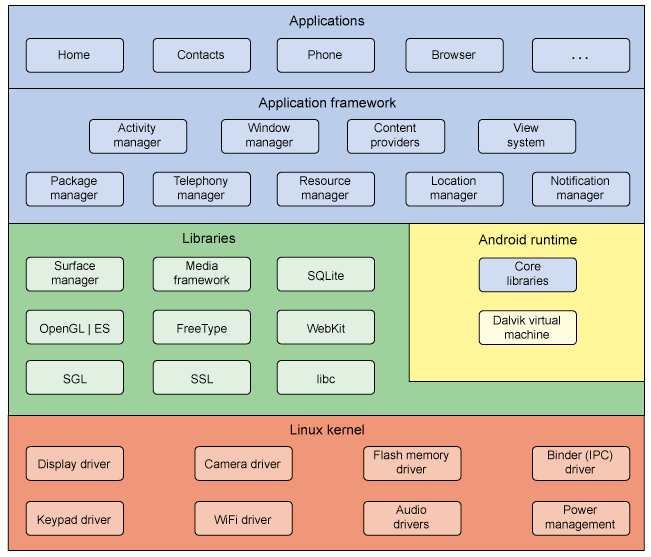
Việc sử dụng các thiết bị di động tinh vi như điện thoại thông minh và máy tính bảng đã tăng lên nhanh chóng trong vài năm qua. Các nhà phân tích tin rằng sự gia tăng này sẽ tiếp tục và các thiết bị di động sẽ lấy PC là thiết bị truy cập web phổ biến nhất vào năm 2013. Các nhà phát triển phần mềm đang nhắm mục tiêu các nền tảng di động đang phải đối mặt với một số thách thức mới. Các khía cạnh như hiệu suất và cuộc sống khó khăn phải được tính đến để một ứng dụng thành công về mặt thương mại.

Vì vậy, câu hỏi được đặt ra. Làm thế nào để bạn phát triển các ứng dụng hiệu suất cao mà là năng lượng tiết kiệm? Mục tiêu của bài báo cáo này là cung cấp cái nhìn sâu sắc có giá trị về nền tảng Android và cho phép các nhà phát triển application đưa ra quyết định đủ điều kiện để tối đa hóa hiệu suất và hiệu quả năng lượng của các ứng dụng của họ. Nền tảng Android đã được chọn vì nó hiện có thị phần lớn nhất và dự kiến sẽ tiếp tục phát triển trong những năm tới.

# **CHƯƠNG 1: TỔNG QUAN TỐI ƯU** **PHẦN DI ĐỘNG**

## **1.1. Hệ điều hành Android**

*Hình 1. SEQ Hình\_1. \\* ARABIC 1 Kiến trúc hệ điều hành android*

Android là một hệ điều hành được phát triển trên nền tảng Linux, thiết kế dành riêng cho các thiết bị di động có màn hình cảm ứng như điện thoại thông minh và máy tính bảng. Ban đầu, hệ điều hành này được phát triển bởi công ty Android Inc., dưới sự hỗ trợ tài chính từ Google. Đến năm 2005, Google đã chính thức mua lại Android và tiếp tục phát triển hệ điều hành này.

Android được xây dựng dựa trên nhân Linux, tích hợp thêm một máy ảo tùy chỉnh nhằm tối ưu hóa việc sử dụng bộ nhớ và tài nguyên phần cứng trong môi trường di động. Với mã nguồn mở, Android cho phép cộng đồng các nhà phát triển tự do mở rộng và sáng tạo, giúp nền tảng này không ngừng phát triển qua các ứng dụng di động mới mẻ và độc đáo.

Điểm đặc biệt của Android là không có sự phân biệt giữa các ứng dụng hệ thống cơ bản và ứng dụng của bên thứ ba. Tất cả đều được cung cấp quyền truy cập tương đương vào các tính năng và dịch vụ của thiết bị, tạo điều kiện thuận lợi để phát triển các ứng dụng đa dạng.

Trên các thiết bị chạy Android, người dùng được tự do tùy chỉnh theo nhu cầu cá nhân. Họ có thể thay đổi giao diện màn hình, kiểu gọi điện, hay thậm chí định cấu hình để thiết bị chỉ hiển thị những nội dung mà họ mong muốn. Android mang đến sự linh hoạt và cá nhân hóa cao, đáp ứng tối đa nhu cầu của người dùng.

## **1.2. Tổng quan về tối ưu hóa trong lập trình di động**

Hiện nay, sự phát triển nhanh chóng các công cụ miễn phí giúp bất kỳ ai cũng có thể viết ứng dụng tiếp cận hàng triệu khách hàng. Nền tảng Android, hiện diện trên nhiều thiết bị từ smartphone, máy tính bảng cho đến TV, tạo ra cơ hội nhưng cũng đặt ra thách thức lớn cho nhà phát triển trong việc tối ưu ứng dụng của mình.

#### **Thách Thức Tối ưu Hóa Ứng Dụng Android**

Mỗi trường thiết bị di động có những hạn chế nhất định so với các thiết bị PC hay laptop:

* Bộ nhớ giới hạn
* Vi xử lý tính toán chậm
* Truy xuất tài nguyên mạng hạn chế
* Dung lượng pin bị giới hạn

Những yếu tố này đặt ra bài toán cho nhà phát triển: làm thế nào để ứng dụng vừa chạy mượt, an toàn, vừa đáp ứng tối đa nhu cầu người dùng.

#### **Nguyên Tắc Tối ưu Hóa Ứng Dụng**

* Tối ưu năng lượng
  + Giảm thiểu các hoạt động tích cự đồng trên nền.
  + Tắt các broadcast receiver khi không cần thiết.
  + Theo dõi và đo lường thời gian sử dụng pin.
* Tối ưu bộ nhớ
  + Sử dụng các kiểu dữ liệu phù hợp.
  + Tối đa hoá truy cập bộ nhớ.
  + Sử dụng các bộ nhớ cache để lưu trữ kết quả tính toán.
* Tối ưu hiệu năng
  + Lựa chọn ngôn ngữ và framework hợp lý cho bài toán.
  + Cài đặt đa luồng để tối đa hiệu năng xử lý.
* Tối ưu trong truy cập mạng và dữ liệu
  + Giảm thiểu lưu lượng dữ liệu truyền tải.
  + Sử dụng các API có sẵn của Android để tối ưu tài nguyên.
  + Tối ưu hoá các truy vấn cơ sở dữ liệu SQLite.

#### **Quy Trình Tối ưu Dựa Trên Giai Đoạn Phát Triển**

* Giai Đoạn Thiết Kế
  + Xác định nghiệp vụ và yêu cầu của bài toán.
  + Lựa chọn mô hình thiết kế tối ưu cho hệ thống.
* Giai Đoạn Phát Triển (Mã Hóa)
  + Lựa chọn các mã nguồn hiệu quả.
  + Kết hợp nhiều thư viện một cách hợp lý.
* Giai Đoạn Thực Thi
  + Xác định môi trường chạy phù hợp.
  + Thiết kế cơ sở dữ liệu tối ưu.

#### **Kết Luận**

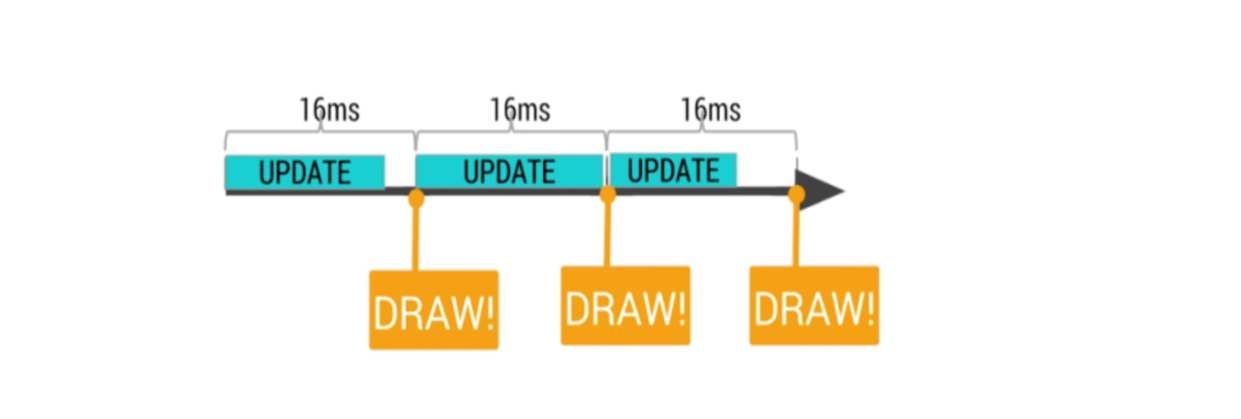
Việc tối ưu hóa ứng dụng Android là một quá trình không ngừng nghỉ trong bối cảnh nền tảng di động liên tục thay đổi. Nhờ việc áp dụng những nguyên tắc và quy trình tối ưu một cách linh hoạt, nhà phát triển có thể đảm bảo ứng dụng vừa đáp ứng tối đa nhu cầu khách hàng, vừa duy trì sự cạnh tranh trên thị trường.

## **1.3. Một vài nguyên nhân ảnh hưởng đến hiệu xuất của ứng dụng**

### **1.3.1 Slow Rendering**

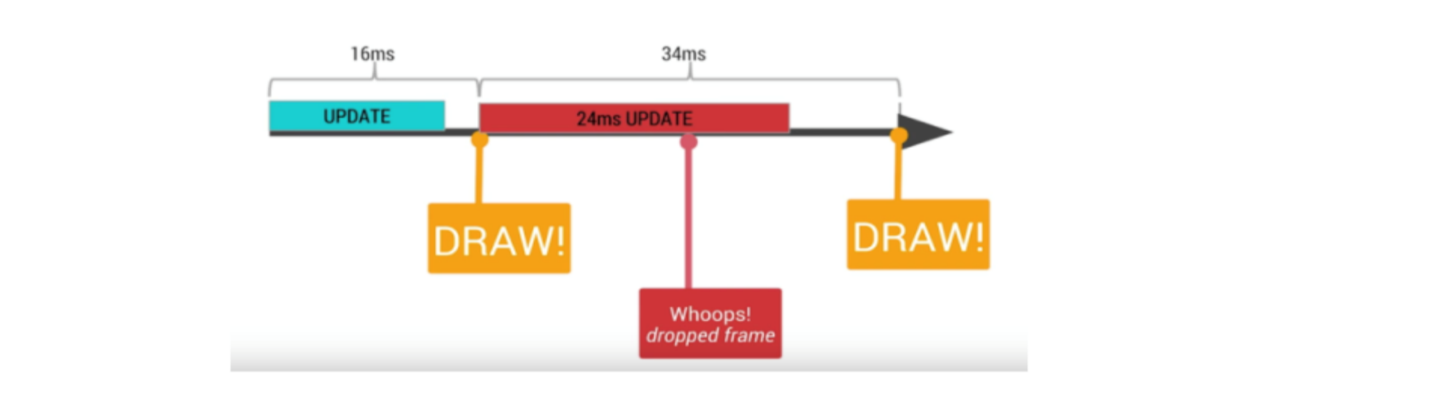
Slow Rendering là vấn đề hiệu suất cơ bản nhất. Bởi vì những gì những designer muốn chúng ta làm và những gì chúng ta thực sự làm, có thể không giống nhau, và sự cố gắng làm cho chúng trực quan khiến chúng ta bị thất bại trong việc phát triển. Rendering xác định theo thời gian, đảm bảo ứng dụng đang chạy mượt mà ở 60 FPS mà không có bất kỳ khung hình nào bị mất hoặc bị delay.

* Nguyên nhân:
* Hệ thống cố gắng để vẽ lại activity của chúng ta sau mỗi 16ms. Điều này có nghĩa là ứng dụng của chúng ta phải làm tất cả các logic cần thực hiện cho việc update screen phải được thực hiện trong 16ms.



*Hình 1. 2 Slow Rendering*

* Đây là những gì xảy ra khi ứng dụng không thể hoàn thành logic trong 16ms:



*Hình 1. 3 Dropped frame*

* Điều này được gọi là **dropped frame**. Ví dụ, Nếu xảy ra trường hợp tính toán của bạn mất 24ms, hệ thống cố gắng vẽ một hình mới trên màn hình, nhưng nó không sẵn sàng. Do đó nó không làm mới bất kì điều gì. Và đây là lý do, người dùng thấy việc làm mới bức ảnh xảy ra sau 32ms thay vì 16ms. Nếu có một frame bị bỏ qua, animation sẽ không mượt mà.
* Công cụ giúp cải thiện:
* Hierarchy Viewer:
  + Hierarchy Viewer là một tool xây dựng trong Android Device Monitor mà cho phép bạn kiểm tra thuộc tính và layout speed cho mỗi view trong layout hierarchy. Nó có thể giúp bạn tìm performance gây ra bởi cấu trúc của view hierarchy của bạn, sau đó giúp bạn đơn giản hoá hierarchy và giảm overdraw.
* Profile GPU Rendering:
  + Profile GPU Rendering cung cấp một biểu diễn trực quan nhanh về thời gian tiêu tốn để render các frame của một UI window so với tiêu chuẩn 16ms.

### **1.3.2 Layout**

Layout là một phần quan trọng của ứng dụng Android mà ảnh hưởng trực tiếp đến trải nghiệm người dùng. Nếu thực hiện kém, layout của bạn có thể khiến cho ứng dụng đầy bộ nhớ với giao diện người dùng chậm. Mỗi widget và layout bạn thêm vào ứng dụng của bạn yêu cầu khởi tạo, layout và vẽ. Ví dụ sử dụng nested instance của LinearLayout có thể dẫn tới cấu trúc view quá sâu.

* Cách tăng hiệu suất layout:
* Tối ưu hóa layout hierarchy

Sử dụng basic layout structure dẫn tới các layout hiệu quả nhất. Tuy nhiên, mỗi widget và layout thêm vào ứng dụng của bạn yêu cầu khởi tạo, layout và vẽ. Ví dụ, sử dụng nested instance của LinearLayout có thể dẫn tới cấu trúc view quá sâu. Hơn nữa, nested một số instance của LinearLayout mà sử dụng layout\_weight parameter có thể đặc biệt tiêu tốn tài nguyên do cần được measure hai lần. Điều này đặc biệt quan trọng khi layout được inflate liên tục, khi được sử dụng trong một ListView hoặc GridView.

* Tái sử dụng layout với <layout>

Tái sử dụng layout đặc biệt mạnh mẽ vì nó cho phép bạn tạo các layout phức tạp có thể sử dụng lại. Ví dụ, một yes/no button panel, hoặc một custom progress bar với một description text. Nó cũng có nghĩa là bất cứ thành phần nào trong ứng dụng của bạn sử dụng chung trong nhiều layout đều có thể được extract, quản lý riêng rẽ sau đó include trong mỗi layout. Vì vật, trong khi bạn có thể tạo các UI component riêng rẽ bằng cách viết một custom View, bạn có thể làm điều đó dễ dàng hơn bằng cách sử dụng một một layout file.

* Loading Views on Demand

Thỉnh thoảng, layout của bạn có thể cần các view phức tạp mà ít khi được sử dụng. Cho dù chúng là item detail, progress indicator, hoặc undo message, bạn có thể giảm memory sử dụng và tăng tốc render bằng cách loading view chỉ khi chúng cần.

### **1.3.3 Power Usage**

**Battery Usage Reduction**cũng là một phần quan trọng của một android development vì vậy việc tối ưu hoá này cuối cùng sẽ dẫn đến việc giữ chân người dùng, vì nhiều lần người dùng gỡ cài đặt vì hao pin.

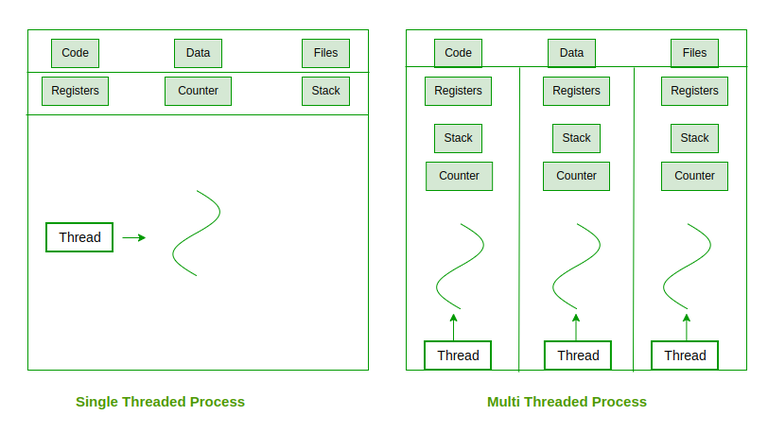
* Khắc phục:
* Giảm tối đa các network call.
* Tránh wake lock càng nhiều càng tốt.
* Sử dụng GPS cẩn thận.
* Sử dụng Alarm manager cẩn thận.

### **1.3.4 Multithreading**

Thread (luồng) là một phần thực thi độc lập bên trong một chương trình. Tất cả các chương trình java đều có ít nhất 1 thread là *main* *thread* được khởi tạo với [JVM](https://shareprogramming.net/jvm-va-su-khac-biet-giua-jvm-jdk-jre/) lúc khởi chạy chương trình, hàm *main()* sẽ được gọi bởi*[main thread](https://shareprogramming.net/main-thread-trong-java/)*.

Multithreading (đa luồng) là chúng ta có nhiều hơn 1 *thread* (main thread) thực thi bên trong chương trình. Hãy tưởng tượng 1 *thread* giống như 1 CPU riêng biệt trong máy tính, các CPU này sẽ thực hiện các nhiệm vụ được giao cùng lúc làm tăng tốc độ xử lý cho chương trình.

Trong thực tế, một CPU máy tính sẽ chia sẽ thời gian thực thi giữa các *thread*, luân chuyển qua lại giữa các *thread* của chương trình trong một khoảng thời gian nhất định. Nếu máy tính có nhiều CPU thì các *thread* trong chương trình có thể thực thi trên nhiều CPU khác nhau. Với CPU chỉ được giao 1 *thread* thì nó chỉ thực thi *thread* này, nếu được giao nhiều hơn 1 *thread* thì chúng tiến hành phân bổ thời gian chạy giữa các *thread*trong  một khoảng thời gian nhất định trước khi chuyển sang thực thi *thread* khác.

**

*Hình 1. 4 Xử lý luồng trong ứng dụng*

Tuy nhiên tồn tại rất nhiều vấn đề trong *multithreading* mà các lập trình viên phải đối mặt. Các *thread* thực thi trong cùng một chương trình nghĩa là đọc ghi cùng một vùng nhớ.

### **1.3.5 Memory**

Nếu thiết bị Android của bạn chỉ sở hữu lượng RAM "khiêm tốn" nhưng lại bị bắt chạy quá nhiều ứng dụng cùng lúc hay kích hoạt các chương trình chạy nền thì có thể dẫn đến tình trạng máy bị treo hoặc vận hành chậm chạp. Đó là lúc mà bạn cần đến những ứng dụng tối ưu hóa bộ nhớ RAM như Smart RAM Booster.

* Nguyên nhân:
* Sử dụng một đối tượng quá lớn
* Sử dụng sai context
* Không hủy dialog trong fragment
* Không hủy dialog trong fragment

Trong những nguyên nhân đã nêu ở trên. Nguyên nhân phổ biến nhất cũng như đã tồn tại từ lâu đó là việc liên quan đến memory. Đây là vấn đề được rất nhiều người quan tâm. Thay vì việc mở rộng thêm bộ nhớ như hiện người dùng muốn tối ưu memory trong khi kích thước bộ nhớ không đổi sao cho sử dụng có hiệu năng tốt nhất.

# **CHƯƠNG 2: TỐI ƯU BỘ NHỚ TRONG LẬP TRÌNH ỨNG DỤNG ANDROID**

## **2.1. Cơ chế cấp phát và giải phóng bộ nhớ trong Android**

Android sử dụng nhân Linux và áp dụng bộ nhớ ảo cho phép phân trang (paging) và ánh xạ (mapping) bộ nhớ. Tuy nhiên, không giống như các hệ điều hành khác sử dụng nhân Linux, Android không hỗ trợ bộ nhớ trao đổi (swap). Điều này có nghĩa là mọi đối tượng được tạo ra hoặc vùng dữ liệu đã phân trang đều nằm trong RAM, và không có thao tác page-in/page-out khi cần swapping bộ nhớ từ RAM ra đĩa. Vì lý do đó, cách duy nhất để giải phóng bộ nhớ trên Android là loại bỏ tham chiếu (strong/hard-reference) tới các đối tượng không cần thiết, từ đó cho phép trình dọn rác (Garbage Collector) thực hiện việc giải phóng bộ nhớ.

Bộ nhớ trong Android được chia thành hai loại chính là bộ nhớ riêng (Private Memory) và bộ nhớ chung (Shared Memory). Bộ nhớ riêng chỉ được sử dụng bởi ứng dụng và không chia sẻ với các ứng dụng khác, toàn bộ vùng nhớ này được cấp phát vào tiến trình của ứng dụng. Trong khi đó, bộ nhớ chung bao gồm các thành phần như framework classes, assets, và thư viện native được chia sẻ giữa các ứng dụng, giúp tiết kiệm tài nguyên hệ thống.

Ngoài ra, bộ nhớ trong Android còn được phân loại thành "Dirty Memory" và "Clean Memory." Dirty Memory là bộ nhớ chỉ được lưu trữ trong RAM và không thể khôi phục từ bộ nhớ đĩa nếu bị loại bỏ, trong khi Clean Memory bao gồm các mục có thể lưu trữ đồng thời trên RAM và đĩa, cho phép hệ thống khôi phục khi cần thiết.

Với việc không hỗ trợ swap và sự phụ thuộc vào RAM, quản lý bộ nhớ hiệu quả là yếu tố then chốt để tối ưu hóa hiệu năng ứng dụng Android, đồng thời giảm thiểu nguy cơ rò rỉ bộ nhớ và cải thiện trải nghiệm người dùng.

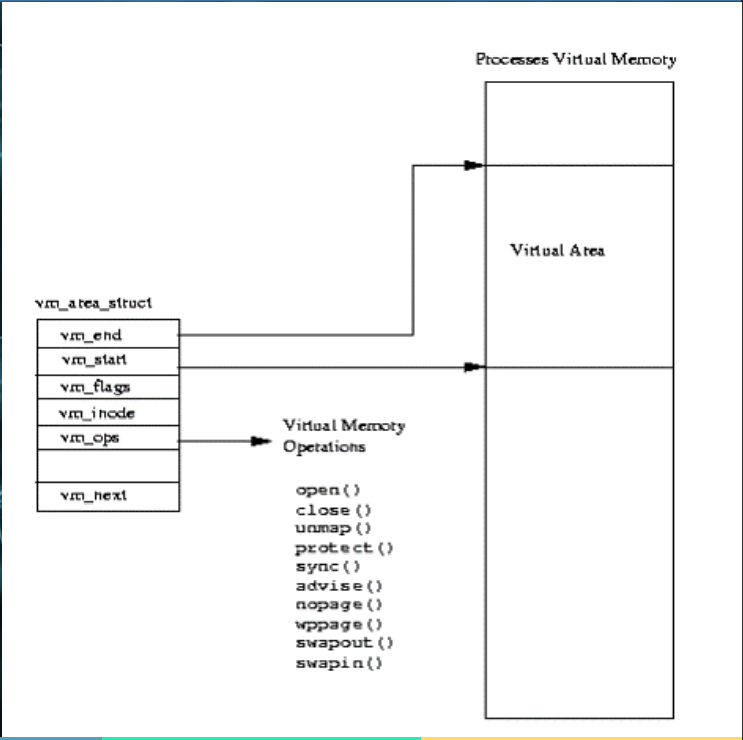
### **2.1.1 Memory Mapping, Dalvik Virtual Machine, Garbage Collection**

#### **Memory mapping**

Memory Mapping (ánh xạ bộ nhớ) là quá trình ánh xạ nội dung của một tệp hoặc hình ảnh vào không gian địa chỉ ảo của một tiến trình. Điều này cho phép tiến trình truy cập trực tiếp đến dữ liệu của tệp như thể đó là một phần của bộ nhớ chính, giúp cải thiện hiệu suất đọc/ghi và quản lý dữ liệu. Khi sử dụng memory mapping, thay vì đọc hoặc ghi toàn bộ tệp vào bộ nhớ, hệ thống chỉ nạp các phần cần thiết vào RAM khi tiến trình yêu cầu, từ đó tối ưu việc sử dụng bộ nhớ và giảm thiểu thời gian truy xuất.

Trong Android, memory mapping được áp dụng chủ yếu cho việc xử lý các tệp dữ liệu lớn hoặc tệp hình ảnh cần truy cập thường xuyên, chẳng hạn như tệp lưu trữ tài nguyên đa phương tiện hoặc thư viện native. Cơ chế này không chỉ giúp tiết kiệm tài nguyên hệ thống mà còn giảm bớt chi phí về hiệu năng khi xử lý các tệp lớn.

Việc sử dụng memory mapping hiệu quả là yếu tố quan trọng để tối ưu hóa bộ nhớ và tăng cường hiệu suất của ứng dụng Android, đặc biệt khi các ứng dụng cần truy cập nhiều dữ liệu hoặc xử lý các tệp có dung lượng lớn.



*Hình 2. 1 Cơ chế memory mapping*

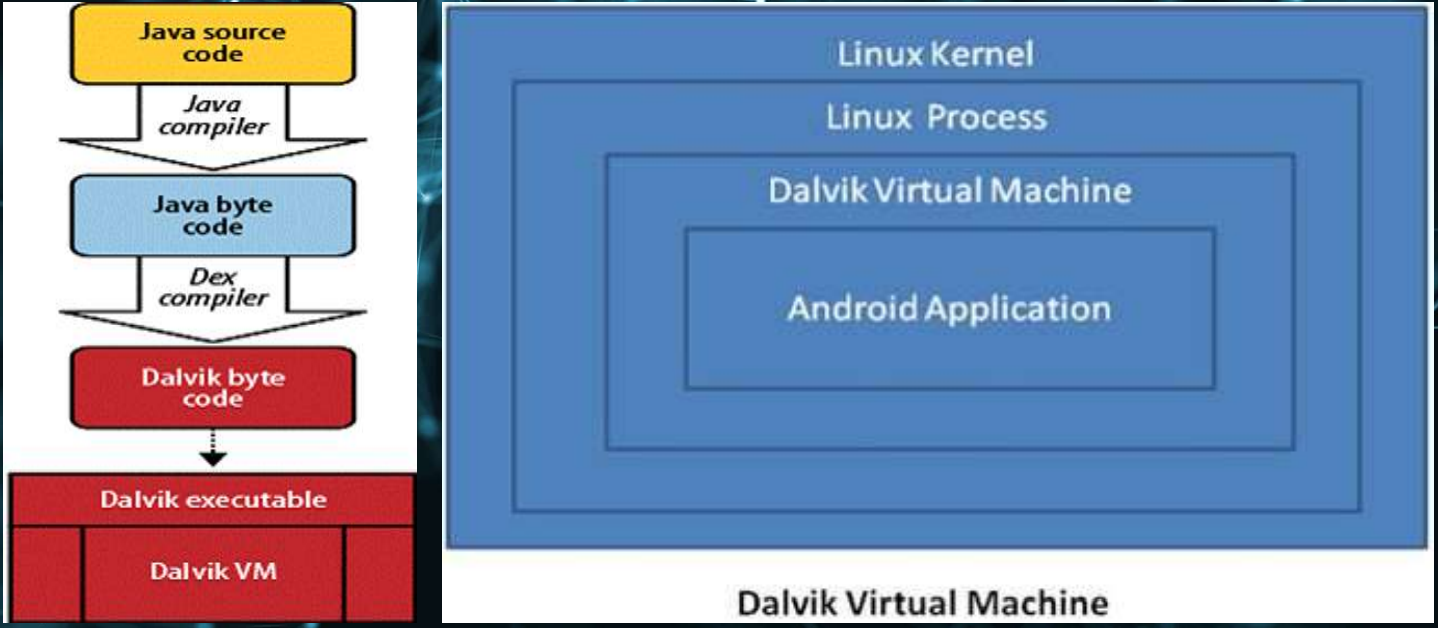
***Dalvik virtual machine***

Dalvik là máy ảo được Google phát triển cho hệ điều hành Android nhằm thực thi các ứng dụng viết bằng Java hoặc ngôn ngữ tương thích. Dalvik VM đóng vai trò quan trọng trong việc biên dịch và thực thi mã nguồn ứng dụng thành mã máy có thể chạy trực tiếp trên thiết bị Android.

Khác với các máy ảo Java (Java Virtual Machine - JVM) truyền thống, Dalvik được thiết kế để tối ưu hóa hiệu suất và tiết kiệm bộ nhớ cho các thiết bị di động có tài nguyên hạn chế. Một điểm đặc biệt của Dalvik là khả năng chạy nhiều phiên bản máy ảo đồng thời, giúp các ứng dụng Android hoạt động độc lập và hiệu quả hơn.

Dalvik sử dụng định dạng tệp đặc biệt là .dex (Dalvik Executable), được tối ưu hóa cho không gian lưu trữ và tốc độ thực thi. Điều này giúp giảm thiểu kích thước tệp và cải thiện hiệu suất bộ nhớ so với định dạng tệp .class truyền thống của JVM.

Mặc dù Dalvik đã được thay thế bởi Android Runtime (ART) trong các phiên bản Android mới hơn, nhưng vai trò của nó trong việc phát triển nền tảng Android ban đầu là không thể phủ nhận. ART tiếp nối thành công của Dalvik với những cải tiến về hiệu suất và quản lý bộ nhớ tốt hơn.



*Hình 2. 2 Cấu trúc Dalvik virtual machine*

#### **Garbage collection**

Garbage Collection (GC) là quá trình tự động dọn dẹp và giải phóng bộ nhớ của các đối tượng dữ liệu không còn được sử dụng, từ đó giúp bộ nhớ có thể phân bổ lại cho các đối tượng mới. Khi một đối tượng không còn tham chiếu hoạt động nào trong ứng dụng, hệ thống sẽ đánh dấu đối tượng đó để thu gom và giải phóng tài nguyên bộ nhớ mà nó chiếm giữ.

#### Mục Tiêu Chính của Garbage Collection:

* Xác định các đối tượng không còn được truy cập: Hệ thống tìm và đánh dấu những đối tượng dữ liệu trong chương trình mà ứng dụng không thể truy cập đến trong tương lai.
* Thu hồi tài nguyên bộ nhớ: Giải phóng bộ nhớ được chiếm giữ bởi các đối tượng đã được xác định là không còn cần thiết.

#### Các Loại Garbage Collection Phổ Biến:

* Mark and Sweep GC:
  + Đánh dấu (Mark) tất cả các đối tượng đang được tham chiếu.
  + Quét (Sweep) và giải phóng bộ nhớ của các đối tượng không được đánh dấu.
* Mark-Sweep-Compact GC:
  + Thực hiện quá trình Mark và Sweep.
  + Sắp xếp lại (Compact) bộ nhớ để giảm phân mảnh sau khi giải phóng.
* Mark and Copy GC:
  + Đánh dấu và sao chép các đối tượng còn hoạt động sang một vùng bộ nhớ mới.
  + Giải phóng toàn bộ vùng bộ nhớ cũ.
* Generational Garbage Collector:
  + Chia bộ nhớ thành các thế hệ (young, old) để quản lý hiệu quả các đối tượng có vòng đời ngắn hoặc dài.

Trong Android, việc tối ưu hóa GC là rất quan trọng để duy trì hiệu suất mượt mà và giảm tình trạng giật hình (jank) trong ứng dụng, đặc biệt là khi xử lý khối lượng lớn dữ liệu hoặc tạo nhiều đối tượng trong thời gian ngắn.

* Hoạt động của Garbage collection trong Android:





*Hình 2. 3 Sơ đồ hoạt động của Garbage Collection*

Trong Android, cơ chế Generational Garbage Collector (GC) được áp dụng nhằm tối ưu hóa việc quản lý và giải phóng bộ nhớ. Không gian bộ nhớ được chia thành các thế hệ khác nhau, chẳng hạn như Young Generation và Old Generation, mỗi thế hệ có cách quản lý và tần suất thu dọn rác khác nhau.

#### **Quy Trình Hoạt Động:**

* **Young Generation:**
  + Tất cả các đối tượng mới được cấp phát bộ nhớ tại Young Generation.
  + Khi một chu kỳ thu dọn rác diễn ra, những đối tượng vẫn còn tham chiếu (được gọi là "đối tượng sống sót") sẽ được chuyển lên Old Generation.
  + Các đối tượng còn lại trong Young Generation, không còn tham chiếu, sẽ bị giải phóng để nhường chỗ cho các đối tượng mới.
* **Old Generation:**
  + Các đối tượng đã sống sót qua nhiều chu kỳ thu dọn rác được lưu trữ tại Old Generation.
  + Chu kỳ thu dọn rác ở Old Generation diễn ra ít thường xuyên hơn do giả định rằng các đối tượng này có xu hướng sống lâu hơn.
  + Việc giảm tần suất thu gom rác cho Old Generation giúp cải thiện hiệu suất tổng thể và giảm gián đoạn trong hoạt động của ứng dụng.

#### **Ý Tưởng Cơ Bản:**

* Các đối tượng mới tạo thường có vòng đời ngắn, do đó Young Generation được thu gom rác thường xuyên để nhanh chóng giải phóng bộ nhớ.
* Những đối tượng tồn tại qua nhiều lần thu gom rác thường có xu hướng tồn tại lâu dài, do đó Old Generation không cần thu gom thường xuyên.

#### **Lợi Ích:**

* Giảm thiểu tình trạng phân mảnh bộ nhớ.
* Tối ưu hiệu năng nhờ phân loại và quản lý hợp lý các đối tượng có vòng đời ngắn và dài.
* Hạn chế tình trạng tạm dừng ứng dụng do chu kỳ GC kéo dài.

#### **Tùy Biến Theo Ngôn Ngữ:**

Số lượng thế hệ và cách quản lý có thể khác nhau tùy theo ngôn ngữ lập trình. Ví dụ, trong Java thường có ba thế hệ: Young Generation, Old Generation, và Permanent Generation (trong các phiên bản cũ của JVM). Tuy nhiên, Android đã điều chỉnh cơ chế này để phù hợp hơn với môi trường di động, nơi tài nguyên bộ nhớ có hạn và cần tối ưu hóa hiệu năng mạnh mẽ.

### **2.1.2 Cơ chế cấp phát và giải phóng bộ nhớ**

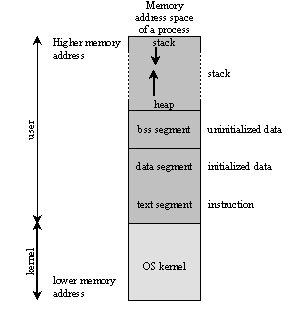
Cơ chế cấp phát và giải phóng bộ nhớ của ứng dụng trên Android, đặc biệt là trong Dalvik VM (máy ảo Dalvik), bao gồm các bước quản lý bộ nhớ hiệu quả và giảm thiểu rủi ro phân mảnh bộ nhớ. Dalvik sử dụng Dalvik heap cho mỗi tiến trình, với một phạm vi bộ nhớ ảo giới hạn.

#### **Cấp Phát Bộ Nhớ:**

* Dalvik Heap:  
  Mỗi ứng dụng chạy trên Android có một Dalvik heap riêng biệt, và heap này được giới hạn trong một phạm vi bộ nhớ ảo duy nhất. Kích thước của heap này có thể phát triển tùy theo nhu cầu, nhưng sẽ bị giới hạn bởi một giá trị mà hệ thống định nghĩa cho mỗi ứng dụng.
* Kích Thước Heap Hợp Lý:  
  Kích thước hợp lý của heap không giống như số bộ nhớ vật lý mà heap này sử dụng. Android tính toán tỉ lệ Set Kích thước (PSS - Proportional Set Size), một giá trị đại diện cho bộ nhớ mà ứng dụng sử dụng, bao gồm cả các trang bộ nhớ sạch và bẩn được chia sẻ với các tiến trình khác. Tuy nhiên, PSS chỉ tính tỷ lệ bộ nhớ mà ứng dụng thực sự sử dụng từ bộ nhớ chung được chia sẻ.

#### **Giải Phóng Bộ Nhớ:**

* Thu Gom Rác (Garbage Collection):  
  Dalvik sử dụng cơ chế garbage collection để giải phóng bộ nhớ khi các đối tượng không còn được tham chiếu. Sau mỗi chu kỳ thu gom rác, Dalvik sẽ kiểm tra và tìm các trang bộ nhớ không sử dụng, rồi trả lại chúng cho hệ điều hành thông qua lệnh madvise.
* Không Phân Mảnh Heap:  
  Android không thực hiện việc chống phân mảnh bộ nhớ bằng cách sắp xếp lại không gian heap. Điều này có nghĩa là khi có bộ nhớ chưa sử dụng, heap không được thu nhỏ lại để dồn các không gian trống lại với nhau. Tuy nhiên, sau khi thu gom rác, Dalvik có thể thu hồi các trang không sử dụng và trả lại chúng cho hệ điều hành.
* Cải Tạo Bộ Nhớ:  
  Việc cải tạo bộ nhớ từ các phân bổ nhỏ có thể hiệu quả, vì các trang bộ nhớ sử dụng cho các phân bổ nhỏ vẫn có thể được chia sẻ với các trang khác mà chưa bị giải phóng. Điều này giúp tiết kiệm bộ nhớ và giảm thiểu lãng phí tài nguyên.



*Hình 2. 4 Cơ chế cấp phát và giải phóng bộ nhớ*

### **2.1.3 Hạn chế bộ nhớ ứng dụng**

Trong môi trường Android, để duy trì một hệ thống đa tác vụ hiệu quả, hệ điều hành giới hạn dung lượng bộ nhớ heap mà mỗi ứng dụng có thể sử dụng. Điều này giúp ngăn chặn các ứng dụng chiếm dụng quá nhiều tài nguyên bộ nhớ, ảnh hưởng đến hiệu suất chung của hệ thống.

#### **Giới Hạn Bộ Nhớ Heap:**

* **Giới Hạn Bộ Nhớ Heap:** Android đặt một giới hạn cứng về kích thước bộ nhớ heap cho mỗi ứng dụng. Giới hạn này có sự khác biệt giữa các thiết bị, phụ thuộc vào tổng bộ nhớ RAM mà thiết bị đó có. Cụ thể, các thiết bị có bộ nhớ RAM lớn hơn sẽ có giới hạn heap cao hơn.
* **OutOfMemoryError:** Nếu ứng dụng vượt quá giới hạn bộ nhớ heap và cố gắng phân bổ thêm bộ nhớ, hệ thống sẽ trả về lỗi **OutOfMemoryError**. Đây là một lỗi phổ biến khi ứng dụng cố gắng lưu trữ hoặc xử lý dữ liệu quá lớn so với bộ nhớ khả dụng.

#### **Truy Vấn Bộ Nhớ Ứng Dụng:**

Để giúp ứng dụng tự quản lý bộ nhớ và tránh gặp lỗi **OutOfMemoryError**, ta có thể truy vấn hệ thống để xác định chính xác dung lượng bộ nhớ heap có sẵn. Điều này đặc biệt hữu ích khi chúng ta muốn lưu trữ dữ liệu tạm thời hoặc tối ưu bộ nhớ sử dụng.

#### **Lợi Ích:**

* Quản lý bộ nhớ hiệu quả: Việc biết được lượng bộ nhớ khả dụng giúp bạn tối ưu hóa việc sử dụng bộ nhớ trong ứng dụng, tránh tình trạng hết bộ nhớ.
* Dự đoán và xử lý lỗi: Việc truy vấn bộ nhớ giúp ứng dụng chủ động dự đoán được khả năng xảy ra lỗi OutOfMemoryError và có các biện pháp xử lý trước khi lỗi xảy ra.

Thông qua việc giám sát và giới hạn bộ nhớ sử dụng, Android giúp các ứng dụng hoạt động hiệu quả hơn trên nhiều loại thiết bị khác nhau, đảm bảo rằng một ứng dụng không chiếm quá nhiều tài nguyên và gây ảnh hưởng đến các ứng dụng khác hoặc toàn bộ hệ thống.

### **2.1.4 Quản lý chuyển đổi ứng dụng**

Trong môi trường Android, việc chuyển đổi giữa các ứng dụng thường xuyên là một phần không thể thiếu trong thói quen sử dụng của người dùng. Tuy nhiên, việc chuyển đổi này đôi khi gây phiền phức, bởi vì người dùng phải mở từng ứng dụng một và thao tác trở lại để tiếp tục công việc. Để giải quyết vấn đề này, các ứng dụng như **Switchr – App Switcher** đã ra đời, giúp người dùng tận dụng tối đa không gian màn hình của thiết bị Android để dễ dàng chuyển đổi giữa các ứng dụng mà không cần phải thực hiện các thao tác phức tạp.

#### **Ứng Dụng Switchr – App Switcher:**

* Ứng dụng này giúp người dùng có thể chuyển đổi giữa các ứng dụng đang mở ngay trên màn hình mà không cần phải quay lại màn hình chính hay dùng nút quay lại mặc định.
* Nó sử dụng các cử chỉ vuốt trực tiếp trên màn hình để chuyển đổi ứng dụng, giúp tăng tốc độ thao tác và tối ưu hóa trải nghiệm người dùng.

#### **Quản Lý Quy Trình và Bộ Nhớ trong Android:**

Trên Android, hệ điều hành không sử dụng bộ nhớ hoán đổi (swap memory) như một số hệ điều hành khác mà thay vào đó, quản lý bộ nhớ ứng dụng thông qua bộ nhớ đệm gần đây nhất (LRU – Least Recently Used). Khi người dùng chuyển đổi ứng dụng, các quy trình không còn được sử dụng sẽ được lưu trữ trong bộ nhớ cache để hệ thống có thể sử dụng lại khi người dùng quay lại ứng dụng đó.

* Quản Lý Quy Trình:  
  Khi người dùng khởi chạy một ứng dụng lần đầu, một quy trình được tạo ra cho ứng dụng đó. Tuy nhiên, khi người dùng rời khỏi ứng dụng, quy trình đó không bị đóng mà sẽ được lưu trữ trong bộ nhớ cache. Điều này giúp khi người dùng quay lại ứng dụng, quy trình cũ có thể được tái sử dụng, giúp giảm thời gian chuyển đổi giữa các ứng dụng.
* Vấn Đề với Bộ Nhớ Cache:  
  Nếu ứng dụng của bạn lưu trữ quy trình trong bộ nhớ cache nhưng không sử dụng bộ nhớ đó hiệu quả, ngay cả khi ứng dụng không được người dùng sử dụng, nó có thể làm giảm hiệu suất tổng thể của hệ thống. Quá trình này có thể gây tiêu tốn bộ nhớ và ảnh hưởng đến các ứng dụng khác.
* Giải Quyết Khi Hệ Thống Cạn Bộ Nhớ:  
  Khi bộ nhớ của hệ thống gần hết, Android sẽ bắt đầu giết các quy trình lưu trữ trong bộ nhớ cache, ưu tiên các quy trình ít được sử dụng gần đây nhất. Tuy nhiên, hệ thống sẽ không chỉ giết chết các quy trình trong bộ nhớ cache mà còn xem xét việc giải phóng bộ nhớ cho các quy trình quan trọng hơn, như các ứng dụng đang sử dụng bộ nhớ một cách nặng nề.

#### **Tối Ưu Hóa Quản Lý Bộ Nhớ:**

Để tránh tình trạng giảm hiệu suất, các nhà phát triển cần lưu ý quản lý bộ nhớ hiệu quả trong ứng dụng của mình, đảm bảo không giữ lại quy trình không cần thiết trong bộ nhớ cache quá lâu. Một số cách để tối ưu hóa bộ nhớ bao gồm:

* Giải phóng tài nguyên không cần thiết khi ứng dụng không còn hoạt động.
* Sử dụng Garbage Collection để dọn dẹp bộ nhớ không sử dụng.
* Giảm thiểu việc giữ các quy trình không cần thiết trong bộ nhớ, giúp hệ thống có thể tái sử dụng tài nguyên bộ nhớ một cách hiệu quả.

## **2.2. Các phương pháp tối ưu bộ nhớ**

Trong quá trình xây dựng ứng dụng nên xem xét các ràng buộc về RAM xuyên suốt tất cả các giai đoạn phát triển, bao gồm cả trong quá trình thiết kế ứng dụng (trước khi bạn bắt đầu phát triển). Có nhiều cách bạn có thể thiết kế và viết mã dẫn đến kết quả hiệu quả hơn, thông qua việc tổng hợp các kỹ thuật giống nhau được áp dụng lặp đi lặp lại.

### **2.2.1 Sử dụng services đúng cách**

Trong phát triển ứng dụng Android, việc sử dụng **services** đúng cách là một yếu tố quan trọng giúp tối ưu bộ nhớ và hiệu suất của ứng dụng. Dưới đây là những hướng dẫn và lưu ý để tránh các vấn đề về bộ nhớ khi sử dụng services trong ứng dụng Android.

Trong phát triển ứng dụng Android, việc sử dụng services đúng cách là yếu tố quan trọng giúp tối ưu bộ nhớ và nâng cao hiệu suất của ứng dụng. Một trong những lỗi phổ biến trong quản lý bộ nhớ là để service chạy khi không còn cần thiết. Khi một service được khởi động, hệ thống ưu tiên duy trì tiến trình của nó, chiếm dụng một lượng bộ nhớ RAM đáng kể. Nếu service không được dừng sau khi hoàn thành nhiệm vụ, nó sẽ gây lãng phí tài nguyên và có thể dẫn đến rò rỉ bộ nhớ. Do đó, cần đảm bảo dừng service ngay sau khi nhiệm vụ của nó hoàn tất để giảm thiểu tác động đến bộ nhớ và hiệu suất của hệ thống.

Việc khởi chạy quá nhiều services mà không dừng đúng lúc có thể gây ra tình trạng quá tải bộ nhớ, làm giảm hiệu quả trong việc chuyển đổi giữa các ứng dụng và gây sự cố hệ thống. Điều này bởi vì hệ thống Android ưu tiên duy trì các tiến trình của service, chiếm dụng bộ nhớ RAM và giảm bộ nhớ khả dụng cho các quy trình khác. Để tránh tình trạng này, việc sử dụng JobScheduler thay vì các services chạy liên tục là một giải pháp hiệu quả. JobScheduler giúp lên lịch và quản lý các tác vụ nền như tải dữ liệu, đồng bộ hóa mà không cần phải chạy liên tục, nhờ vậy giúp tiết kiệm tài nguyên hệ thống và giảm tải bộ nhớ.

Khi cần sử dụng service cho các tác vụ nền ngắn hạn, việc sử dụng IntentService là một lựa chọn hợp lý. IntentService tự động kết thúc sau khi hoàn thành nhiệm vụ, giúp quản lý vòng đời của service dễ dàng hơn và đảm bảo bộ nhớ được giải phóng ngay lập tức sau khi tác vụ kết thúc. Đồng thời, IntentService chạy trên một luồng riêng biệt, giảm thiểu tác động đến giao diện người dùng và không làm giảm hiệu suất hệ thống. Việc sử dụng đúng loại service giúp tiết kiệm tài nguyên, tối ưu bộ nhớ và cải thiện trải nghiệm người dùng trong ứng dụng.

### **2.2.2 Giải phóng bộ nhớ khi giao diện bị ẩn**

Khi người dùng điều hướng đến một ứng dụng khác và giao diện người dùng của ứng dụng không còn hiển thị nữa, việc giải phóng các tài nguyên chỉ được giao diện người dùng sử dụng là rất quan trọng để tối ưu hóa bộ nhớ và cải thiện hiệu suất hệ thống. Giải phóng tài nguyên vào thời điểm này có thể giúp tăng dung lượng bộ nhớ của hệ thống cho các quy trình khác, đặc biệt là các quy trình đã được lưu trong bộ nhớ cache, điều này sẽ có tác động trực tiếp đến chất lượng trải nghiệm người dùng. Để làm điều này, một cách hiệu quả là triển khai phương thức onTrimMemory() trong các Activity và sử dụng hằng số TRIM\_MEMORY\_UI\_HIDDEN. Phương thức này thông báo rằng giao diện người dùng của ứng dụng đã bị ẩn và tài nguyên giao diện người dùng cần được giải phóng.

Mặc dù onStop() cũng được gọi khi một Activity bị ẩn (chẳng hạn khi người dùng chuyển sang một Activity khác trong ứng dụng), phương thức này không phải lúc nào cũng phù hợp để giải phóng tài nguyên giao diện người dùng. Trong khi onStop() thích hợp để giải phóng tài nguyên liên quan đến hoạt động như kết nối mạng hoặc hủy đăng ký bộ thu phát sóng, onTrimMemory() với TRIM\_MEMORY\_UI\_HIDDEN mới là phương thức thích hợp để giải phóng tài nguyên giao diện người dùng, khi mà toàn bộ giao diện người dùng đã bị ẩn khỏi chế độ xem.

Việc sử dụng onTrimMemory() giúp đảm bảo rằng các tài nguyên giao diện người dùng sẽ chỉ được giải phóng khi thực sự cần thiết, và nếu người dùng điều hướng trở lại từ một Activity khác trong ứng dụng, các tài nguyên giao diện người dùng vẫn sẽ được giữ lại, giúp việc chuyển đổi giữa các Activity trong ứng dụng diễn ra nhanh chóng và mượt mà. Khi áp dụng các phương pháp này, ứng dụng sẽ sử dụng tài nguyên hệ thống hiệu quả hơn, tránh lãng phí bộ nhớ và cải thiện trải nghiệm người dùng.

### **2.2.3 Xử lí bộ nhớ khi thiếu tài nguyên**

Khi hệ thống Android gặp phải tình trạng thiếu tài nguyên bộ nhớ, phương thức onTrimMemory() được gọi để thông báo về mức độ bộ nhớ còn lại và yêu cầu ứng dụng giải phóng tài nguyên. Các mức độ bộ nhớ khác nhau mà onTrimMemory() cung cấp sẽ giúp ứng dụng phản hồi một cách phù hợp, từ đó tối ưu hóa việc quản lý tài nguyên và đảm bảo hiệu suất của hệ thống.

* TRIM\_MEMORY\_RUNNING\_MODERATE: Khi hệ thống sắp hết bộ nhớ nhưng ứng dụng không bị đánh giá là có nguy cơ bị kill. Tuy nhiên, các quy trình trong bộ đệm LRU đang bị hệ thống tích cực giết đi, do đó ứng dụng nên giải phóng thêm tài nguyên để giảm thiểu tác động lên hệ thống.
* TRIM\_MEMORY\_RUNNING\_LOW: Ứng dụng vẫn đang chạy nhưng bộ nhớ hệ thống đang ở mức thấp. Lúc này, ứng dụng cần giải phóng các tài nguyên không cần thiết để cải thiện hiệu suất của hệ thống. Việc giải phóng tài nguyên không sử dụng sẽ giúp tối ưu bộ nhớ và cải thiện hiệu suất ứng dụng.
* TRIM\_MEMORY\_RUNNING\_CRITICAL: Khi bộ nhớ hệ thống đang ở mức rất thấp và hầu hết các quy trình trong bộ đệm LRU đã bị giết, ứng dụng cần giải phóng mọi tài nguyên không quan trọng ngay lập tức. Nếu bộ nhớ không được khôi phục, hệ thống sẽ bắt đầu giết các quy trình đang chạy, bao gồm cả các quy trình quan trọng như các dịch vụ đang hoạt động.

Ngoài các mức trên, có một số mức độ bộ nhớ khác khi ứng dụng được lưu vào bộ nhớ cache:

* TRIM\_MEMORY\_BACKGROUND: Khi hệ thống sắp hết bộ nhớ và ứng dụng của bạn ở gần đầu danh sách LRU. Mặc dù ứng dụng không có nguy cơ bị kill ngay lập tức, nhưng cần giải phóng các tài nguyên dễ khôi phục để ứng dụng có thể tiếp tục hoạt động mượt mà khi người dùng quay lại.
* TRIM\_MEMORY\_MODERATE: Ứng dụng nằm ở vị trí giữa danh sách LRU và có thể bị kill nếu hệ thống tiếp tục thiếu bộ nhớ. Lúc này, cần giải phóng các tài nguyên không cần thiết để giảm thiểu nguy cơ bị kill.
* TRIM\_MEMORY\_COMPLETE: Đây là mức độ bộ nhớ thấp nhất, ứng dụng của bạn có thể là một trong những quy trình đầu tiên bị giết nếu hệ thống không thể khôi phục bộ nhớ ngay lập tức. Trong trường hợp này, mọi tài nguyên không quan trọng cần được giải phóng ngay lập tức để giữ lại trạng thái hoạt động của ứng dụng.

Các mức độ này cung cấp thông tin quan trọng giúp ứng dụng xử lý việc giải phóng tài nguyên một cách hợp lý, bảo vệ bộ nhớ của hệ thống và duy trì hiệu suất ổn định cho cả hệ thống và ứng dụng.

### **2.2.4 Kiểm soát sử dụng bộ nhớ tối ưu**

Để tối ưu việc sử dụng bộ nhớ trong ứng dụng Android, cần chú ý đến các yếu tố liên quan đến dung lượng RAM và kích thước heap mà hệ thống cung cấp cho mỗi ứng dụng. Mỗi thiết bị có dung lượng RAM khác nhau, do đó, giới hạn heap cũng sẽ khác nhau. Bạn có thể sử dụng phương thức getMemoryClass() để ước tính dung lượng heap khả dụng cho ứng dụng tính bằng megabyte. Nếu ứng dụng cố gắng phân bổ bộ nhớ vượt quá dung lượng heap này, một lỗi OutOfMemoryError sẽ xảy ra.

Trong những tình huống đặc biệt, bạn có thể yêu cầu kích thước heap lớn hơn bằng cách đặt thuộc tính bigHeap thành "true" trong tệp khai báo <application> của ứng dụng. Nếu thực hiện điều này, bạn có thể gọi getLargeMemoryClass() để ước tính kích thước heap lớn hơn. Tuy nhiên, yêu cầu heap lớn chỉ nên áp dụng cho một nhóm ứng dụng nhỏ mà việc sử dụng nhiều RAM là hợp lý và cần thiết.

Điều quan trọng là không nên yêu cầu một heap lớn chỉ vì ứng dụng hết bộ nhớ. Việc yêu cầu thêm bộ nhớ chỉ nên được thực hiện khi bạn chắc chắn về việc phân bổ bộ nhớ và lý do tại sao bộ nhớ đó phải được giữ lại. Hơn nữa, ngay cả khi bạn tin rằng ứng dụng có thể sử dụng heap lớn, vẫn nên tránh yêu cầu nó một cách không cần thiết. Việc sử dụng thêm bộ nhớ sẽ gây ảnh hưởng tiêu cực đến hiệu suất hệ thống, vì thu gom rác sẽ mất nhiều thời gian hơn và quá trình chuyển đổi giữa các tác vụ hoặc thực hiện các hoạt động thông thường có thể trở nên chậm hơn.

### **2.2.5 Tối ưu hóa Bitmap**

Tối ưu hóa Bitmap trong Android là một bước quan trọng để giảm thiểu việc sử dụng bộ nhớ và cải thiện hiệu suất của ứng dụng, đặc biệt khi làm việc với hình ảnh. Một trong những kỹ thuật tối ưu hóa chính là chỉ giữ bitmap trong RAM ở độ phân giải phù hợp với màn hình của thiết bị hiện tại. Nếu bitmap ban đầu có độ phân giải quá cao, cần thu nhỏ nó để tiết kiệm bộ nhớ và đảm bảo hình ảnh không bị phóng đại hoặc làm giảm hiệu suất.

Để thực hiện điều này, có thể sử dụng các phương thức như BitmapFactory.Options để điều chỉnh kích thước của bitmap khi tải nó từ bộ nhớ hoặc từ đĩa. Cụ thể, thuộc tính inSampleSize của BitmapFactory.Options có thể được sử dụng để chỉ định một tỷ lệ giảm kích thước (ví dụ, giảm một nửa, giảm một phần tư) trước khi tải bitmap vào bộ nhớ. Điều này giúp tải bitmap với độ phân giải nhỏ hơn mà không gây lãng phí tài nguyên bộ nhớ.

Ngoài ra, cần chú ý đến các giải pháp như tái sử dụng bitmap (sử dụng Bitmap.recycle() khi không cần thiết) và tối ưu hóa việc tải và giải phóng bitmap để tránh gây ra tình trạng rò rỉ bộ nhớ.

### **2.2.6 Sử dụng cấu trúc dữ liệu hiệu quả**

Khi phát triển ứng dụng Android, việc sử dụng các cấu trúc dữ liệu như SparseArray, SparseBooleanArray và LongSparseArray có thể mang lại nhiều lợi ích trong việc tối ưu hóa bộ nhớ và hiệu suất. Những cấu trúc dữ liệu này đặc biệt hữu ích khi ánh xạ dữ liệu theo chỉ số hoặc khóa mà không cần phải lưu trữ các khóa và giá trị thừa. Điều này giúp giảm thiểu việc sử dụng bộ nhớ và giảm overhead so với các cấu trúc dữ liệu truyền thống như HashMap.

Một ví dụ là SparseArray, được sử dụng thay thế cho HashMap trong trường hợp ánh xạ các khóa nguyên (int) tới các giá trị đối tượng. Việc sử dụng SparseArray giúp giảm kích thước bộ nhớ vì nó không yêu cầu mỗi mục trong bộ sưu tập phải lưu trữ khóa dưới dạng đối tượng. Tương tự, SparseBooleanArray tối ưu hóa việc lưu trữ các giá trị boolean, và LongSparseArray giúp tối ưu hóa việc ánh xạ các giá trị dài.

Tuy nhiên, mặc dù các cấu trúc này giúp tiết kiệm bộ nhớ, chúng lại không sử dụng phương pháp băm như HashMap mà thay vào đó là tìm kiếm nhị phân, do đó, có thể chậm hơn khi xử lý một số lượng lớn các mục nhập. Do đó, khi ứng dụng cần tra cứu nhanh với số lượng lớn các mục, HashMap có thể là lựa chọn tối ưu hơn.

Việc lựa chọn giữa các cấu trúc dữ liệu này phụ thuộc vào yêu cầu cụ thể của ứng dụng, bao gồm số lượng mục nhập, loại dữ liệu được ánh xạ, và yêu cầu về hiệu suất trong quá trình tra cứu.

### **2.2.7 Hạn chế chi phí bộ nhớ không cần thiết**

Khi phát triển ứng dụng, cần có sự hiểu biết rõ ràng về chi phí tài nguyên của ngôn ngữ và các thư viện đang sử dụng, đồng thời luôn giữ ý thức về việc tiết kiệm bộ nhớ ngay từ giai đoạn lập trình đầu tiên. Một số ví dụ về việc tiêu tốn bộ nhớ khi sử dụng các cấu trúc dữ liệu và thành phần trong Java và Android:

* **Enums**: Trong Java, enums thường tiêu tốn bộ nhớ gấp đôi so với các biến static thông thường. Vì vậy, cần tránh sử dụng enums trong Android trừ khi thực sự cần thiết, đặc biệt là khi ứng dụng yêu cầu tối ưu hóa bộ nhớ.
* **Các lớp trong Java**: Mỗi lớp trong Java (bao gồm cả các inner class) tiêu tốn khoảng 500 bytes bộ nhớ. Điều này có thể tích tụ nhanh chóng nếu ứng dụng sử dụng quá nhiều lớp không cần thiết.
* **Các đối tượng (Instances)**: Mỗi instance (đối tượng) của một lớp trong Java tiêu tốn khoảng 12-16 bytes bộ nhớ, tùy thuộc vào lớp và các thuộc tính của nó.
* **HashMap**: Mỗi phần tử được thêm vào một HashMap sẽ tiêu tốn thêm khoảng 32 bytes bộ nhớ, bao gồm bộ nhớ cho khóa, giá trị, và các cấu trúc hỗ trợ như bảng băm.
* **Abstractions**: Việc sử dụng các lớp abstract, interface là một thực tiễn lập trình tốt, giúp cải thiện tính linh hoạt và khả năng bảo trì của mã nguồn. Tuy nhiên, các trừu tượng này có thể làm tăng độ phức tạp trong cấu trúc chương trình, dẫn đến việc tốn bộ nhớ do các liên kết giữa các lớp và các đối tượng được tạo ra từ các lớp trừu tượng.

Chi phí bộ nhớ có thể gia tăng nhanh chóng khi ứng dụng có cấu trúc lớp phức tạp, hoặc chứa các đối tượng với nhiều thuộc tính tiêu tốn bộ nhớ như bitmap và các collection. Mặc dù tiết kiệm bộ nhớ từng chút một có thể không mang lại sự cải thiện tức thời trong hiệu năng, nhưng việc quản lý tài nguyên bộ nhớ hợp lý sẽ đảm bảo tính bền vững cho ứng dụng, giúp nó hoạt động ổn định trên nhiều thiết bị khác nhau, đặc biệt là với các thiết bị có tài nguyên hạn chế.

### **2.2.8 Lưu ý khi trừu tượng mã hóa nguồn**

Trừu tượng hóa mã nguồn, mặc dù mang lại lợi ích trong việc cải thiện tính linh hoạt và khả năng bảo trì của code, nhưng cũng đi kèm với chi phí về hiệu suất và tài nguyên hệ thống. Khi sử dụng các lớp abstract, một số vấn đề có thể phát sinh:

* **Tăng thời gian thực thi**: Trừu tượng hóa đòi hỏi việc xử lý thông qua các lớp cơ sở, phương thức ảo và các quá trình phản chiếu, dẫn đến việc mã cần phải được thực thi thêm, gây tăng thời gian thực thi so với các phương thức trực tiếp.
* **Tăng tiêu tốn bộ nhớ**: Sử dụng các lớp abstract và các lớp con của chúng tạo ra thêm lớp và đối tượng trong bộ nhớ. Điều này có thể làm tăng đáng kể việc sử dụng bộ nhớ, đặc biệt là khi ứng dụng có một cấu trúc phức tạp với nhiều lớp abstract không thực sự cần thiết.

Vì vậy, khi không có lợi ích rõ ràng từ việc sử dụng trừu tượng, nên tránh sử dụng chúng. Điều này giúp giảm chi phí tài nguyên, đặc biệt là khi ứng dụng yêu cầu tối ưu hóa hiệu suất và tiết kiệm bộ nhớ. Khi quyết định sử dụng trừu tượng, cần cân nhắc kỹ về chi phí tài nguyên mà nó mang lại và lợi ích thực tế mà nó mang đến cho việc phát triển và bảo trì mã nguồn.

### **2.2.9 Quản lí khung phụ thuộc và thư viện bên ngoài**

Quản lý khung phụ thuộc và thư viện bên ngoài là một phần quan trọng trong việc phát triển ứng dụng Android. Khung phụ thuộc như Guice hoặc RoboGuice có thể giúp đơn giản hóa mã nguồn và hỗ trợ kiểm thử, nhưng cũng đi kèm nhiều hạn chế cần được cân nhắc kỹ lưỡng khi phát triển ứng dụng.

Các khung phụ thuộc giúp đơn giản hóa mã nguồn bằng cách tự động quản lý các đối tượng và phụ thuộc thông qua chú thích (@Inject). Điều này giảm thiểu số lượng mã phải viết cho các tác vụ phức tạp trong quản lý phụ thuộc và tạo ra một môi trường thử nghiệm linh hoạt. Tuy nhiên, quá trình này có thể dẫn đến hiệu suất thấp do việc quét mã (classpath scanning) để tìm kiếm các chú thích và đối tượng cần khởi tạo, gây tiêu tốn tài nguyên không cần thiết. Điều này có thể làm tăng kích thước ứng dụng và gây chậm trễ khi khởi động ứng dụng, đặc biệt nếu ứng dụng có nhiều phụ thuộc.

Khung phụ thuộc thường không phù hợp với môi trường di động do các thiết bị di động có tài nguyên hạn chế. Giải pháp thay thế là khởi tạo thủ công các đối tượng hoặc sử dụng các công cụ Dependency Injection nhẹ hơn như Dagger 2 hoặc Hilt, được thiết kế đặc biệt cho môi trường Android và giúp tăng hiệu suất nhờ annotation processing thay vì classpath scanning.

Việc sử dụng thư viện bên ngoài cũng có thể mang lại nhiều lợi ích nhưng cần kiểm tra kỹ lưỡng. Các thư viện không tối ưu hóa cho môi trường di động có thể làm giảm hiệu suất hoặc tiêu tốn tài nguyên hệ thống. Ngoài ra, có thể xảy ra xung đột giữa các thư viện sử dụng các công nghệ khác nhau, chẳng hạn như các thư viện sử dụng protobuf nhẹ hoặc vi mô không tương thích với nhau. Thư viện có thể làm tăng kích thước ứng dụng hoặc gặp khó khăn khi cấu hình ProGuard, đặc biệt là những thư viện sử dụng reflection.

Khi sử dụng thư viện với Activity, các phụ thuộc bổ sung có thể gây ra xung đột và làm phình to kích thước ứng dụng. Các thư viện này có thể không hoạt động tốt nếu không có cấu hình ProGuard chính xác, đòi hỏi lập trình viên phải tinh chỉnh thủ công để ProGuard hoạt động hiệu quả.

### **2.2.10 Sử dụng ProGuard để tối ưu mã nguồn**

Sử dụng ProGuard để tối ưu mã nguồn trong ứng dụng Android là một kỹ thuật quan trọng giúp cải thiện hiệu suất và giảm kích thước của ứng dụng. ProGuard là một công cụ đi kèm với SDK Android, có khả năng cắt bớt, tối ưu hóa và mã hóa giải code bằng cách loại bỏ mã không sử dụng, đổi tên các lớp, trường và phương thức thành các tên ngắn gọn, tối nghĩa về mặt ngữ nghĩa nhưng tối ưu về mặt tài nguyên. Điều này giúp giảm kích thước tệp APK và lượng bộ nhớ cần thiết để tải mã nguồn vào bộ nhớ, từ đó giúp ứng dụng hoạt động hiệu quả hơn.

ProGuard giúp giảm số lượng trang bộ nhớ (memory pages) mà mã nguồn chiếm dụng trong RAM, từ đó giảm bớt các tài nguyên hệ thống không cần thiết. Công cụ này sẽ tự động loại bỏ những đoạn mã không bao giờ được sử dụng trong quá trình biên dịch, giúp tối ưu hóa mã nguồn bằng cách chỉ giữ lại những phần thực sự cần thiết.

## **2.3. Vai trò proguard trong tối ưu bộ nhớ**

### **2.3.1 Giới thiệu về ProGuard**

ProGuard là một công cụ mã nguồn mở miễn phí đi kèm với Android SDK, được sử dụng để tối ưu hóa, thu nhỏ kích thước và bảo mật mã nguồn của các ứng dụng Java hoặc Android. Nó giúp cải thiện hiệu suất của ứng dụng, giảm kích thước tệp APK và tăng cường bảo mật thông qua việc làm khó hiểu mã nguồn.

* Chức năng chính của ProGuard:
* Giảm kích thước ứng dụng: ProGuard giúp loại bỏ các lớp, phương thức, biến, và tài nguyên không sử dụng trong ứng dụng. Quá trình này giúp thu nhỏ kích thước tệp APK, làm cho ứng dụng nhẹ hơn và dễ dàng phân phối hơn.
* Tối ưu hóa mã nguồn: ProGuard có khả năng chuyển đổi một số hàm thành các hàm nội tuyến, tối ưu hóa mã nguồn để tăng hiệu suất ứng dụng. Nó cũng loại bỏ mã dư thừa hoặc không cần thiết, giúp ứng dụng chạy mượt mà và tiết kiệm tài nguyên hơn.
* Làm khó hiểu mã nguồn (Obfuscation): ProGuard thay đổi tên của các lớp, phương thức, biến và hàm thành các chuỗi ngắn và khó hiểu. Điều này giúp bảo vệ mã nguồn của ứng dụng khỏi việc dịch ngược (reverse engineering) và phân tích mã trái phép. Việc làm khó hiểu mã nguồn giúp bảo vệ quyền sở hữu trí tuệ của nhà phát triển và ngăn chặn việc sao chép trái phép.

Nhờ vào những tính năng này, ProGuard không chỉ giúp tối ưu hóa tài nguyên và cải thiện hiệu suất mà còn giúp bảo vệ mã nguồn, tăng cường bảo mật cho ứng dụng Android.

### **2.3.2 Lợi ích khi sử dụng ProGuard**

Proguard là một công cụ rất hữu ích cho việc tạo một ứng dụng tối ưu. Nó giúp ứng dụng của ta giảm được một lượng mã lệnh thừa (unusing code), ứng dụng được tối ưu hoá rất đáng kể vì vậy sẽ giúp cho ứng dụng chạy mượt mà hơn.

Proguard được tạo mặc định khi tạo một ứng dụng Android từ Android Studio, đồng thời chúng ta có thể tùy chỉnh một số cấu hình cho phù hợp với dự án. Chúng đem lại rất nhiều lợi ích, như là:

* **Giảm kích thước ứng dụng**: ProGuard giúp loại bỏ mã lệnh thừa và tài nguyên không sử dụng trong ứng dụng, như hình ảnh trong thư mục drawable hoặc các lớp, phương thức không cần thiết. Điều này giúp giảm kích thước của tệp APK, làm cho ứng dụng nhẹ hơn và nhanh chóng hơn.
* **Tối ưu hóa hiệu suất**: Bằng cách loại bỏ mã không cần thiết và tối ưu hóa mã nguồn, ProGuard giúp ứng dụng chạy mượt mà hơn, giảm thiểu sự lãng phí tài nguyên hệ thống (bộ nhớ, CPU) và cải thiện tốc độ xử lý.
* **Bảo vệ mã nguồn**: Một trong những tính năng quan trọng của ProGuard là khả năng làm khó hiểu mã nguồn. Bằng cách đổi tên các lớp, phương thức và biến thành những chuỗi ký tự ngắn gọn và khó hiểu, ProGuard giúp bảo vệ mã nguồn khỏi các hành động dịch ngược (reverse engineering). Điều này giúp bảo vệ quyền sở hữu trí tuệ và ngăn chặn việc sao chép trái phép mã nguồn ứng dụng.
* **Tăng tính bảo mật**: Việc làm khó hiểu mã nguồn và loại bỏ các phần mã không cần thiết giúp làm giảm khả năng tấn công hoặc khai thác ứng dụng, đặc biệt trong các tình huống khi ứng dụng chứa thông tin nhạy cảm hoặc logic quan trọng.
* **Tích hợp dễ dàng trong Android Studio**: ProGuard được tích hợp sẵn trong Android Studio và có thể được cấu hình theo nhu cầu của dự án. Điều này giúp việc tối ưu hóa ứng dụng trở nên dễ dàng hơn, chỉ cần thiết lập một vài cấu hình trong file build.gradle.
* **Bảo vệ cấu trúc ứng dụng**: ProGuard cũng giúp bảo vệ cấu trúc của ứng dụng bằng cách ngăn chặn việc người khác dễ dàng hiểu và sao chép các lớp và chức năng của ứng dụng.

### **2.3.3 Cách tích hợp ProGuard trong ứng dụng**

#### **2.3.3.1 Cấu hình build gradle của app**



*Hình 2. 5 Cấu hình build gradle*

* **minifyEnabled true** : Khi gắn giá trị cho nó bằng **true** thì mã lệnh của giúp ta lúc này khi build bản release version sẽ được tối ưu, chống dịch ngược, ...

#### **2.3.3.2 Một số custom rules hữu ích trong proguard**

Ta có một số custom rules hữu ích:

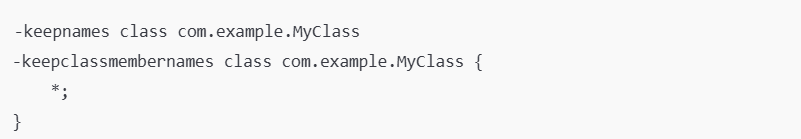
* Keep class files: Đảm bảo rằng các lớp cụ thể không bị xóa trong quá trình tối ưu hóa hoặc làm khó hiểu.



* Keep member for a class: Chỉ giữ lại các thành viên cụ thể của một lớp.



* Keeping names of the class and members: Giữ lại tên lớp và thành viên để dễ dàng gỡ lỗi hoặc tích hợp với các công cụ bên ngoài.



* Using any Library in Android: Để tránh xung đột khi sử dụng thư viện bên thứ ba, cần giữ nguyên tên lớp của thư viện đó



* Only Obfuscate your project: Giữ nguyên tất cả các lớp và thành viên của thư viện bên thứ ba.



* Maintaining annotations: Giữ lại thông tin chú thích cho lớp hoặc phương thức.



* Optimization: Cho phép tối ưu hóa mã bằng cách loại bỏ mã không sử dụng.



## **2.4. Các công cụ tối ưu bộ nhớ**

### **2.4.1 MAT ( Memory Analyzer Tool)**

**MAT (Memory Analyzer Tool)** là một công cụ phân tích bộ nhớ mạnh mẽ, đặc biệt hữu ích trong việc phát hiện và giải quyết các vấn đề liên quan đến việc sử dụng bộ nhớ và rò rỉ bộ nhớ trong các ứng dụng Java, bao gồm cả ứng dụng Android.

Dưới đây là các tính năng nổi bật của **Eclipse Memory Analyzer Tool (MAT)**:

1. **Phân tích chi tiết heap dump**:
   * MAT cung cấp thông tin chi tiết về số lượng và kích thước của các đối tượng trong bộ nhớ.
   * Công cụ giúp lập trình viên xác định các đối tượng đang chiếm dụng nhiều bộ nhớ nhất trong ứng dụng.
2. **Xác định rò rỉ bộ nhớ**:
   * MAT có khả năng phát hiện các đối tượng không thể được thu hồi bởi Garbage Collector (GC).
   * Công cụ giúp lập trình viên xác định nguyên nhân khiến GC không thể giải phóng các đối tượng bị giữ lại không cần thiết, gây rò rỉ bộ nhớ.
3. **Phân tích và tối ưu hóa bộ nhớ**:
   * MAT cung cấp các báo cáo chi tiết về các đối tượng tiêu tốn nhiều tài nguyên, giúp xác định các khu vực cần tối ưu hóa trong ứng dụng.
   * Công cụ hỗ trợ tự động cách ly và gắn cờ các dấu hiệu nghi ngờ rò rỉ bộ nhớ, giúp dễ dàng giải quyết vấn đề.
4. **Báo cáo trực quan**:
   * MAT hiển thị đồ thị về mối quan hệ giữa các đối tượng trong heap, giúp lập trình viên dễ dàng xác định điểm rò rỉ bộ nhớ.
   * Cung cấp các báo cáo định lượng chi tiết về việc sử dụng bộ nhớ, từ đó giúp lập trình viên tối ưu hóa hiệu suất của ứng dụng.

**Cách sử dụng MAT với ứng dụng Android**:

* MAT ban đầu được phát triển như một phần của Eclipse IDE, nhưng lập trình viên Android có thể tải và sử dụng công cụ này dưới dạng ứng dụng độc lập từ Eclipse.org.
* Để sử dụng MAT với Android Studio, bạn cần tạo heap dump từ ứng dụng Android của mình (thường là qua Android Device Monitor hoặc sử dụng các lệnh gdb), sau đó phân tích heap dump này trong MAT để tìm ra các vấn đề về bộ nhớ.

### **2.4.2 JetBrains JVM Debugger Memory View**

**JetBrains JVM Debugger Memory View** là một plugin mạnh mẽ dành cho **IntelliJ IDEA** và **Android Studio**, mở rộng khả năng gỡ lỗi (debug) của JVM, giúp lập trình viên theo dõi và phân tích bộ nhớ heap trong khi debug ứng dụng. Công cụ này hỗ trợ phát hiện và xử lý các vấn đề liên quan đến quản lý bộ nhớ, rò rỉ bộ nhớ, và tối ưu hóa hiệu suất ứng dụng.

**Chức năng chính của JVM Debugger Memory View:**

* **Hiển thị thông tin về heap**:
  + Hiển thị tổng số lượng đối tượng trong heap và nhóm chúng theo tên class.
  + Giúp dễ dàng xác định đối tượng nào đang chiếm dụng nhiều bộ nhớ nhất.
* **Theo dõi đối tượng thời gian thực**:
  + Quan sát số lượng đối tượng cụ thể khi bước qua từng điểm dừng (breakpoint) trong quá trình debug.
  + Cho phép so sánh số lượng đối tượng giữa các thời điểm khác nhau, giúp phát hiện tình trạng cấp phát bộ nhớ không mong muốn hoặc rò rỉ bộ nhớ.
* **Chức năng Garbage Collection (GC)**:
  + Cho phép gọi thủ công Garbage Collector trong quá trình debug để kiểm tra xem đối tượng có thể được thu hồi hay không.
  + Giúp phát hiện các đối tượng không cần thiết mà vẫn giữ lại trong bộ nhớ.
* **Chức năng filter và tìm kiếm**:
  + Lọc và tìm kiếm nhanh các đối tượng trong heap.
  + Tính năng này giúp lập trình viên dễ dàng tìm ra đối tượng cần thiết để tối ưu hóa quá trình phân tích bộ nhớ.

**Lợi ích của JVM Debugger Memory View:**

* **Hiểu rõ hơn về sử dụng bộ nhớ**: Giúp lập trình viên theo dõi chính xác việc sử dụng bộ nhớ trong ứng dụng.
* **Phát hiện rò rỉ bộ nhớ**: Giúp nhận diện các đối tượng bị rò rỉ hoặc không được thu hồi bởi Garbage Collector, làm tăng độ ổn định và hiệu suất của ứng dụng.
* **Cải thiện hiệu suất**: Bằng cách tối ưu hóa việc cấp phát bộ nhớ, có thể giảm thiểu mức tiêu thụ bộ nhớ và tăng tốc độ hoạt động của ứng dụng.
* **Tăng khả năng gỡ lỗi**: Công cụ giúp theo dõi bộ nhớ và các đối tượng trong thời gian thực trong quá trình debug, cho phép lập trình viên tìm ra vấn đề nhanh chóng.

### **2.4.3. Netbeans Profiler**

Netbeans IDE đã tích hợp sẵn profiler với cái tên đơn giản là Profiler. Nó theo dõi các thông tin của CPU, bộ nhớ, luồng, khóa (lock) và các truy vấn SQL, cũng như giám sát JVM cơ bản.

## **2.5. Tìm hiểu công cụ Leak Canary**

### **2.5.1 Giới thiệu Leak Canary**

**Leak Canary** là một công cụ mạnh mẽ và phổ biến được sử dụng để phát hiện **rò rỉ bộ nhớ** trong các ứng dụng Android. Đây là một thư viện mã nguồn mở giúp lập trình viên phát hiện các đối tượng không được giải phóng bộ nhớ khi không còn sử dụng nữa, qua đó giúp giải quyết các vấn đề liên quan đến hiệu suất ứng dụng và tối ưu hóa bộ nhớ.

Rò rỉ bộ nhớ xảy ra khi một ứng dụng cấp phát bộ nhớ cho một đối tượng, nhưng không giải phóng bộ nhớ khi đối tượng không còn được sử dụng nữa. Việc này dẫn đến **tiêu tốn tài nguyên không cần thiết** và ảnh hưởng đến hiệu suất của ứng dụng. Các vấn đề đi kèm với rò rỉ bộ nhớ bao gồm:

* **Hiệu suất kém**: Khi bộ nhớ bị chiếm dụng không đúng cách, ứng dụng sẽ ngày càng chậm và giật lag khi thao tác.
* **Nguy cơ ứng dụng bị treo**: Nếu bộ nhớ tiếp tục bị rò rỉ, ứng dụng có thể gặp lỗi nghiêm trọng hoặc treo, thậm chí dẫn đến việc ứng dụng không thể chạy được nữa.

Trong Android, **rò rỉ bộ nhớ** là một vấn đề phổ biến do sự phức tạp trong việc quản lý vòng đời của **Activity** và **Fragment**. Các đối tượng có thể bị tham chiếu một cách sai cách hoặc không được giải phóng đúng lúc khi các Activity và Fragment không còn hoạt động. Điều này làm cho bộ nhớ không thể được thu hồi, gây ảnh hưởng nghiêm trọng đến hiệu suất của ứng dụng.

**Leak Canary** giúp phát hiện và chẩn đoán rò rỉ bộ nhớ bằng cách theo dõi sự thay đổi trong bộ nhớ của ứng dụng trong thời gian thực. Nó cung cấp các tính năng sau:

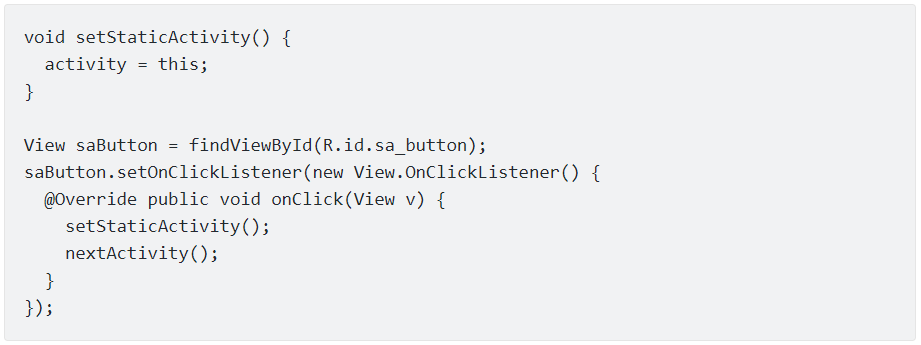
* **Phát hiện tự động**: Leak Canary sẽ tự động phát hiện các đối tượng bị rò rỉ khi một Activity hoặc Fragment bị hủy nhưng đối tượng vẫn giữ tham chiếu.
* **Báo cáo trực quan**: Khi phát hiện rò rỉ bộ nhớ, Leak Canary sẽ tạo ra một báo cáo chi tiết, giúp lập trình viên dễ dàng xác định nguyên nhân của vấn đề và giải quyết chúng.
* **Dễ tích hợp**: Leak Canary dễ dàng tích hợp vào bất kỳ ứng dụng Android nào mà không làm ảnh hưởng nhiều đến mã nguồn.
* **Hỗ trợ phân tích**: Công cụ này cung cấp các chi tiết về các đối tượng gây rò rỉ, bao gồm các đối tượng đã bị giữ tham chiếu, các nguyên nhân chính và sự tương tác giữa các đối tượng.

**Lợi ích khi sử dụng Leak Canary:**

* **Phát hiện rò rỉ bộ nhớ sớm**: Leak Canary giúp phát hiện các vấn đề rò rỉ bộ nhớ trong quá trình phát triển, từ đó có thể xử lý chúng kịp thời trước khi ứng dụng được phát hành.
* **Giảm thiểu các vấn đề hiệu suất**: Bằng cách giải quyết các rò rỉ bộ nhớ, ứng dụng sẽ mượt mà hơn và hoạt động hiệu quả trên các thiết bị có tài nguyên hạn chế.
* **Giảm thiểu nguy cơ treo ứng dụng**: Leak Canary giúp ngăn ngừa việc ứng dụng bị treo hoặc gặp lỗi nghiêm trọng do bộ nhớ bị rò rỉ.

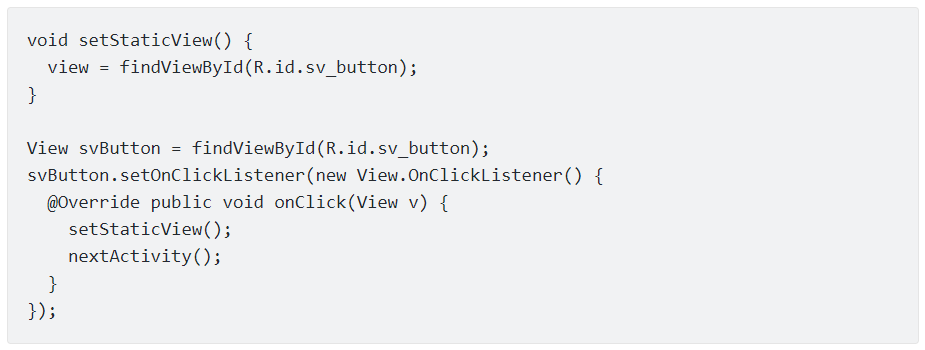
Một số trường hợp bị memory leak:

* Static Activities là trường hợp xảy ra khi bạn định nghĩa 1 biến variable là instance trong Activity sau đó setting và run Activity đó. Đây là trường hợp gây Memory leak gây tốn bộ nhớ nhất của điện thoại.



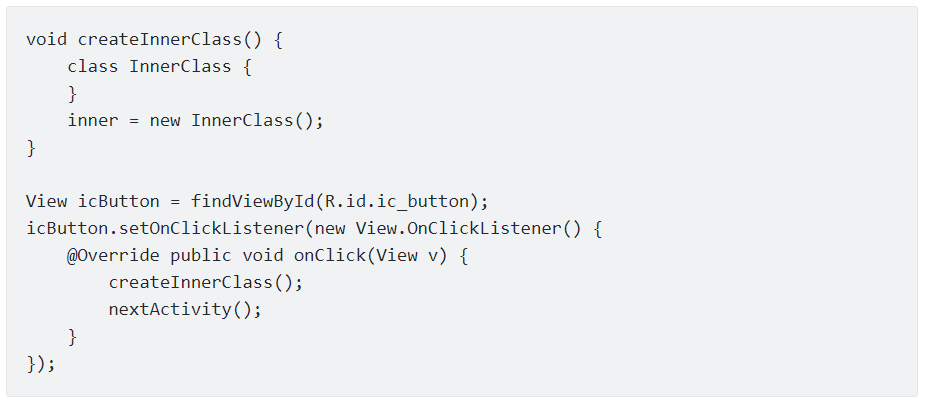
*Hình 2. 6 Ví dụ Static Activities*

* Static Views Là một vấn đề thực tế bạn thường gặp khi thực hiện singleton pattern



*Hình 2. 7 Ví dụ Static Views*

* Non static Inner Classes Định nghĩa inner class trong activity. Nên thay thế non static inner class bằng static inner class



*Hình 2. 8 Ví dụ Non static Inner Classes*

* Handler và Asynctask Sử dụng Handler và asynctask không đúng cách cũng gây ra memory leak rất nhiều

**

*Hình 2. 9 Ví dụ Handler và Asynctask*

### **2.5.2 Cơ chế hoạt động Leak Canary**

#### **1. Giám sát đối tượng với RefWatcher.watch()**

* Khi bạn gọi RefWatcher.watch(), LeakCanary tạo một instance của KeyedWeakReference để tham chiếu yếu (weak reference) đến đối tượng cần theo dõi.
* Mỗi tham chiếu được gán một ID duy nhất giúp nhận diện đối tượng đó trong quá trình phân tích heap sau này.

#### **2. Kiểm tra tham chiếu trong nền**

* Sau một khoảng thời gian nhất định, LeakCanary kiểm tra xem tham chiếu yếu đến đối tượng có bị thu hồi (cleared) bởi Garbage Collector (GC) hay không.
* Nếu tham chiếu vẫn tồn tại, LeakCanary có thể gọi Runtime.gc() để kích hoạt GC thu gom bộ nhớ một cách chủ động.

#### **3. Tạo Heap Dump khi phát hiện nghi vấn rò rỉ**

* Nếu sau khi gọi GC, tham chiếu yếu vẫn chưa bị thu hồi, LeakCanary lưu lại một bản ghi của heap vào file có định dạng .hprof.

#### **4. Phân tích Heap Dump với HeapAnalyzer**

* Một luồng xử lý nền riêng sẽ được khởi động để phân tích file heap dump.
* HeapAnalyzer tìm kiếm các đối tượng KeyedWeakReference thông qua ID đã lưu trước đó.
* Tiến hành xác định đường dẫn ngắn nhất từ đối tượng bị nghi rò rỉ đến các GC Roots.

#### **5. Xác nhận và báo cáo rò rỉ bộ nhớ**

* Nếu phát hiện một đường dẫn mạnh (strong reference) từ đối tượng đến GC roots, LeakCanary xác nhận rằng có rò rỉ bộ nhớ.
* Một chuỗi các tham chiếu dẫn đến rò rỉ sẽ được xây dựng và thông báo qua giao diện người dùng hoặc thông qua logcat.

#### **6. Thông báo đến lập trình viên**

* Thông tin chi tiết về rò rỉ bộ nhớ bao gồm đường dẫn tham chiếu gây ra rò rỉ được hiển thị trong thông báo hoặc được lưu vào báo cáo để lập trình viên khắc phục.

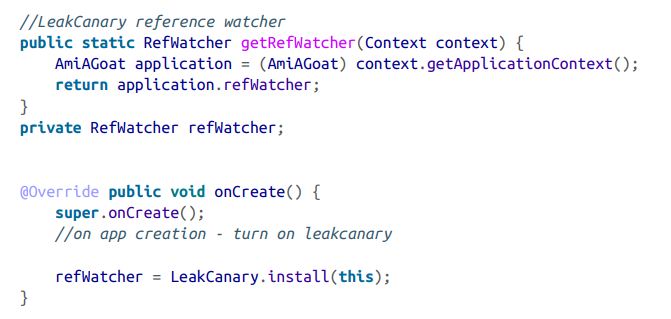
### **2.5.3 Cấu hình LeakCanary trong ứng dụng Android**

Bước 1: Trong build.gradle, thêm 2 dependencies

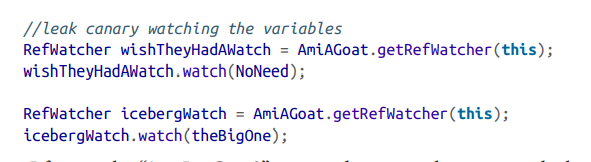


Bước 2: Trong lớp ứng dụng của AmIAGoat, thêm mã nguồn như sau:

* Có chức năng lăng nghe các thay đổi khi ứng dụng được khởi tạo



Bước 3: Để theo rỗi được sự thay đổi đó. Ta sẽ gán cho một tham chiếu cụ thể. Ở đây sử dụng 2 biến CancelTheWatch và Iceberg



### **2.5.4 Phát hiện và xử lí lỗi rò rỉ bộ nhớ**

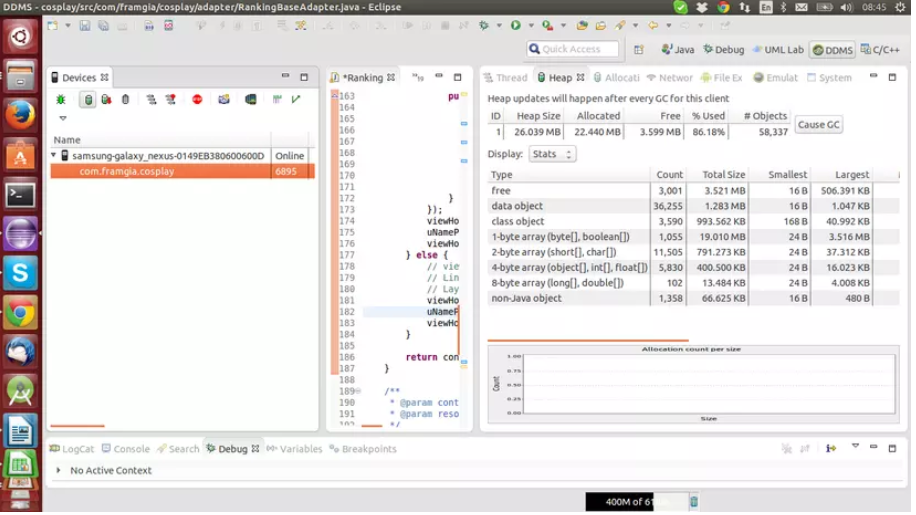
* Hạn chế giữ reference đến activity context (những object tham chiếu đến activity nên có lifecycle giống như activity).
* Dùng application context ở những chỗ có thể.
* Tránh sử dụng inner class trong activity.
* Tránh sử dụng anonymous class trong activity.
* Unregister listener trong **onDestroyed()** của activity.
* Nếu bắt buộc sử dụng inner class, hãy cho nó làm static nested class và sử dụng **WeakReference** để giữ ref đến outer class.

### **2.5.5 Hướng dẫn sử dụng Leak Canary**

#### **1. Android MAT memory leak analytics**

Trong quá trình xây dựng ứng dụng Android, đặc biệt là các ứng dụng có sử dụng nhiều ảnh, 1 trong những lỗi thường xuyên xuất hiện và gây rất nhiều khó khăn cho các developer khi muốn debug và fix bug. Memory Analyzer tool (sau đây sẽ gọi tắt là MAT) sẽ cho phép chúng ta tracking memory và tìm ra những điểm phát sinh lỗi OutOfMemory trong application

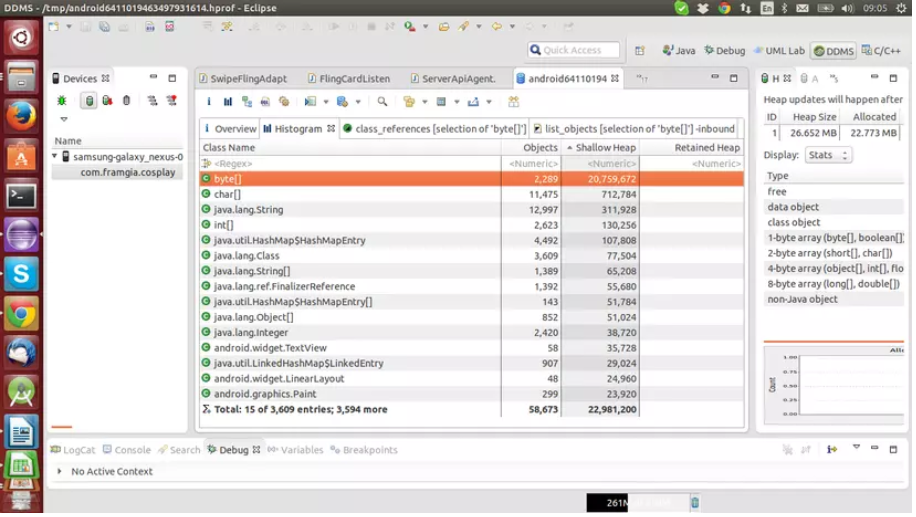
Tạo 1 ứng dụng Android đơn giản để sử dụng MAT thống kê bộ nhớ. Build ứng dụng, mở DDMS trong Android. Bạn sẽ nhìn thấy list các ứng dụng đang chạy. Bạn chọn ứng dụng để track Memory. Click Update heap button trong device session. Click Cause GC button. Tại đây có thể nhìn thấy 1 vài thông tin cơ bản vể bộ nhớ đang được sử dụng. Nó được update sau mỗi lần GC.

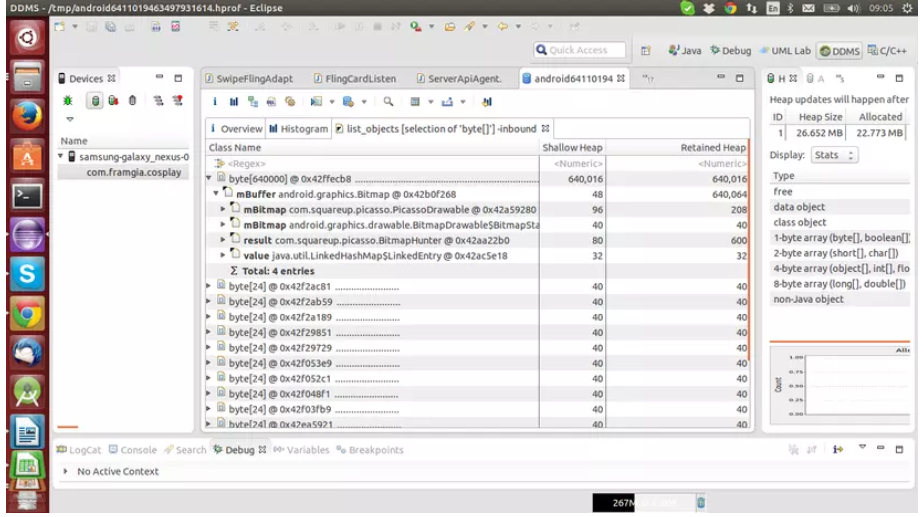


Đóng ứng dụng. Hoặc sử dụng 1 vài chức năng … sau đó click button DUMP HPROF file trong device session và lưu lại file hprof. Có thể lấy heap dump (hprof) bằng cách khác với android.os.Debug.dumpHprofData () .



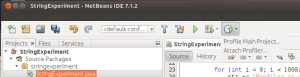
**HPROF file overview:** Tại màn hình show HPROF data, chọn Histogram, Bạn sẽ có thể tìm được nhưng object,class dựa trên package name đang chiếm dụng bao nhiêu bộ nhớ. Sau đó chọn show object by class:



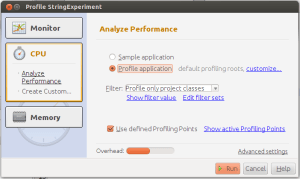


#### **2. Netbeans Profiler**

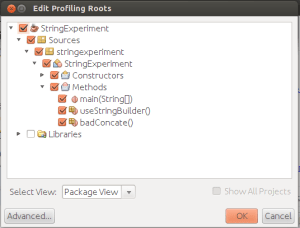
* Bước 1: Nhấn nút Profile Main Project



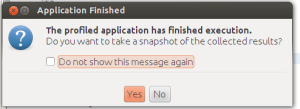
* **Bước 2: Chọn CPU** để bắt đầu phân tích hiệu năng về CPU, tức thời gian thực thi (hai lựa chọn khác là Monitor – theo dõi chi tiết hơn, và Memory – đo đạc các thông số liên quan đến bộ nhớ trong khi ứng dụng thực thi).



Nhấn vào customize, ta có thể tùy chọn việc đo đạc trên lớp nào, hàm nào. Trong trường hợp của ta, cứ để nguyên các lựa chọn mặc định.

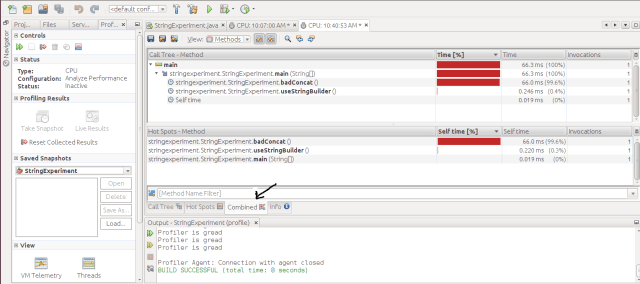


Sau khi một số hộp thoại hiện ra (do bạn chạy Profiler lần đầu nên nó cần phải căn chỉnh để kết quả profiling được chính xác, cứ nhân OK nếu bạn không muốn quan tâm gì thêm), bạn có thể đọc kết quả để phân tích.



* Bước 3: Đọc kết quả và phân tích

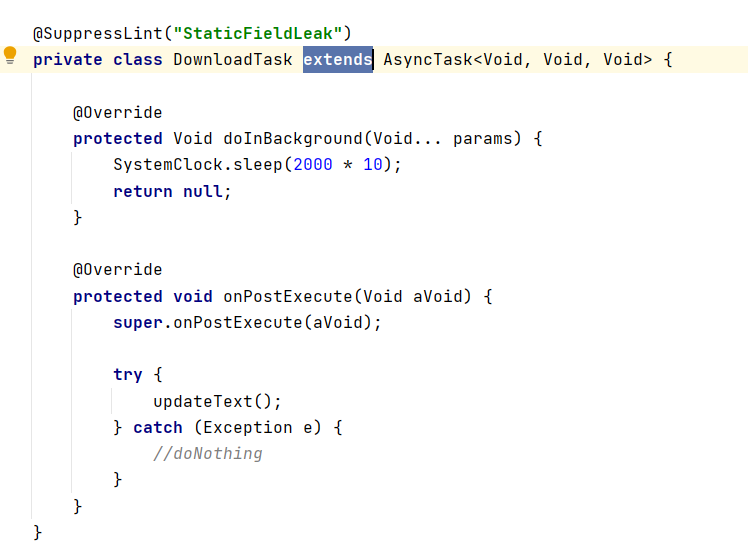
Sau khi profiling xong, bảng kết quả hiện ra với các dữ liệu về thời gian thực thi của từng hàm. Bạn có thể chọn các tab CallTree, Hot Spots và Combined để xem kết quả.  Do chương trình của ta ngắn và đơn giản nên danh sách các hàm được gọi rất đơn giản, ta chọn view Combined để xem:



# **CHƯƠNG 3: PHÂN TÍCH VÀ TIẾN HÀNH THỰC NGHIỆM CÁC TRƯỜNG HỢP LEAK MEMORY**

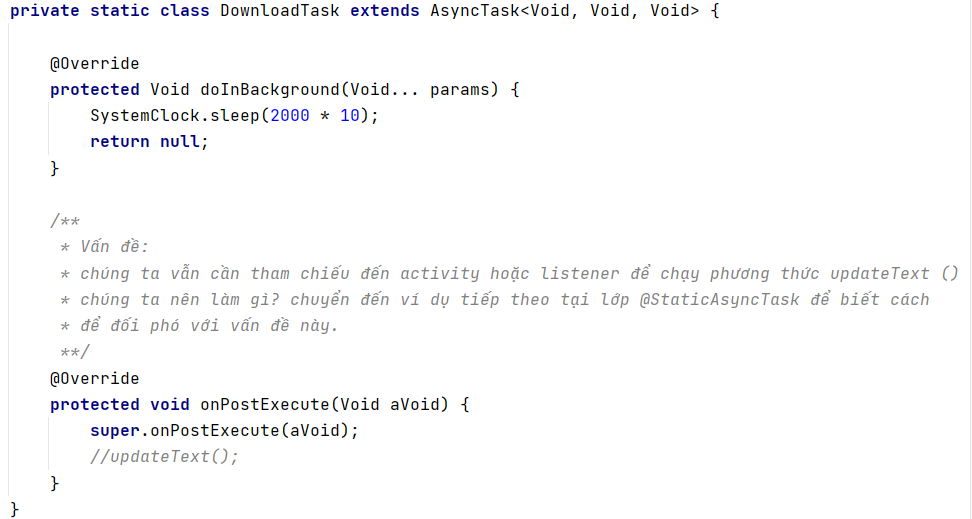
## **3.1. AsyncTaskActivity**

* Chúng ta sẽ bị rò rỉ bộ nhớ khi chúng ta rotate/close activity trong vòng 20 giây sau khi nó được tạo.
* Vì AsyncTask được khai báo là inner class nên nó sẽ chứa tham chiếu của activity khiến activity không đủ điều kiện để thu gom rác.
* LƯU Ý: nếu nhiệm vụ được thực hiện trước khi rotate/close activity, mọi thứ sẽ ổn mà không bị rò rỉ.



*Hình 3. 1 Memory leak khi dùng Async Task*

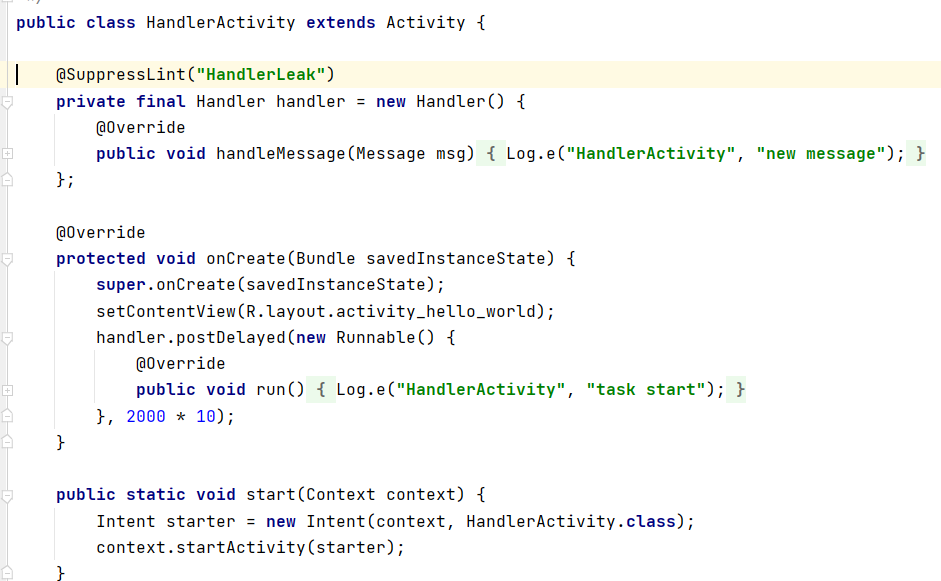
* Để khắc phục sự cố leak này, chúng ta sử dụng static class thay vì inner class. Static class không có tham chiếu đến lớp hoạt động chứa



*Hình 3. 2 Khắc phục memory leak khi dùng AsyncTask*

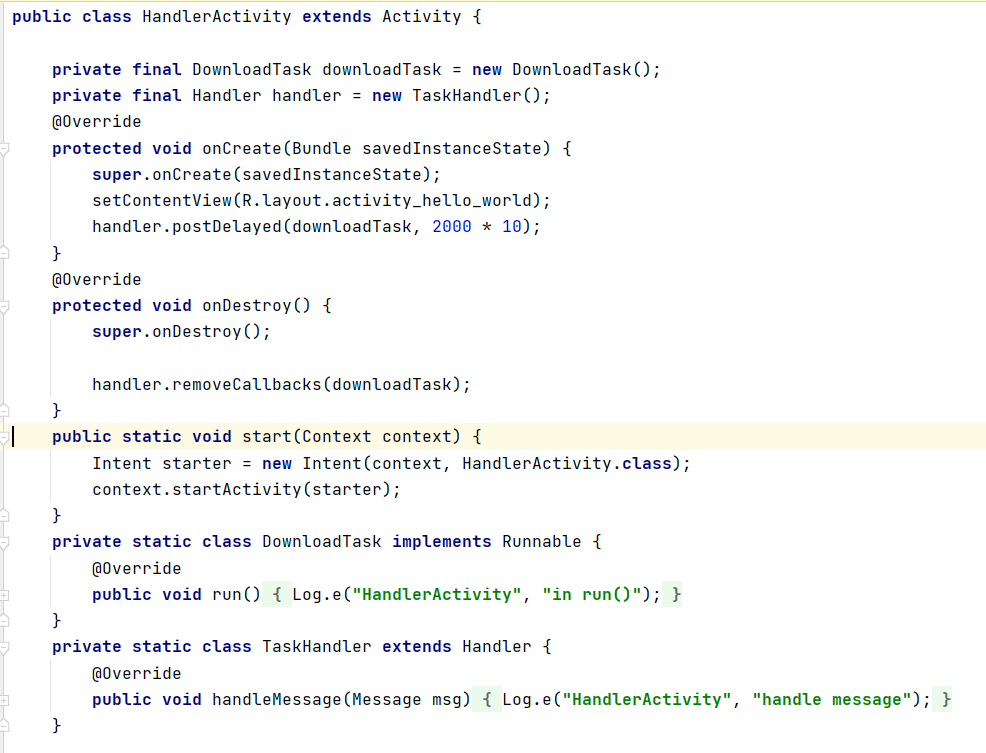
## **3.2. HandlerLeak**

* Đây là trường hợp handler này được khai báo là một lớp bên trong, nó có thể ngăn lớp bên ngoài được thu gom rác



*Hình 3.3 Memory leak khi dùng Handler*

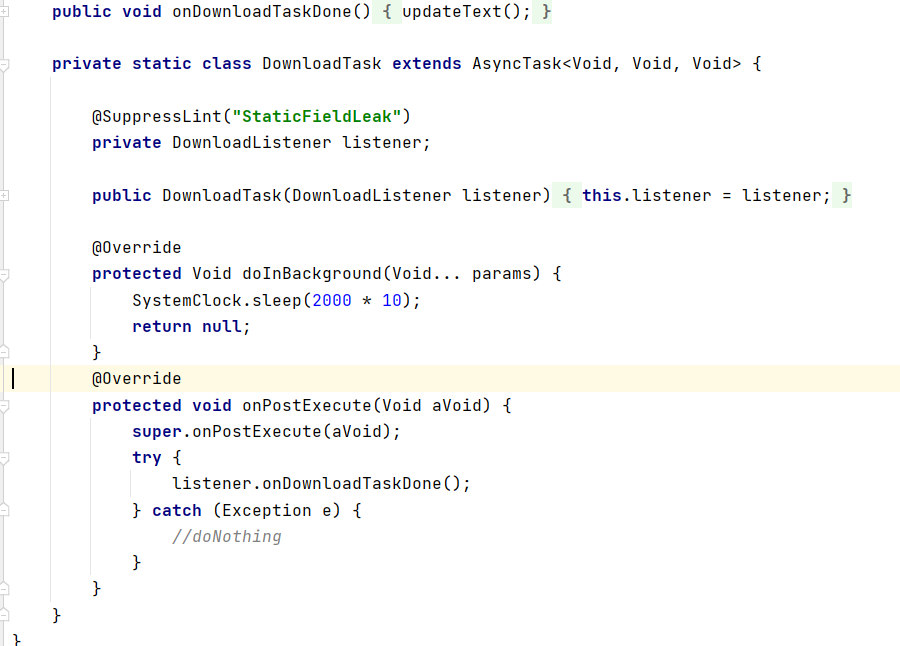
* Để khắc phục vấn để này ta sẽ add hander vào main. Đồng thời Xóa mọi post đang chờ xử lý của Runnable @downloadTask nằm trong message queue, tránh bị rò rỉ.



*Hình 3. 4 Khắc phục memory leak khi dùng Handler*

## **3.3. StaticAsyncTaskActivity**

* Là trường hợp lưu một strong reference của listener, điều này làm cho listener không đủ điều kiện để thu gom rác.



*Hình 3. 5 Memory leak khi sử dụng Static Async Task Activity*

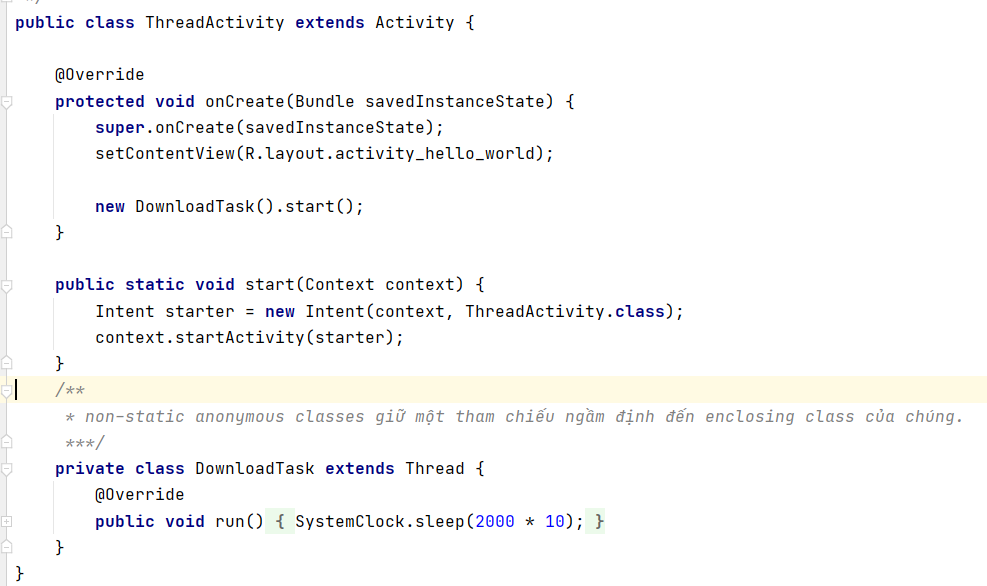
* Để khắc phục sử dụng Weak Reference cho phép Activity được thu thập rác. Rác được thu thập không bảo vệ Weak Reference bắt đầu được thu hồi.
* Mô tả Weak Reference:
  + Weak reference objects, không ngăn cản đối tượng tham chiếu của chúng được hoàn thiện, hoàn thiện và sau đó thu hồi.
  + Giả sử rằng bộ thu gom rác xác định tại một thời điểm nhất định rằng một đối tượng có thể tiếp cận yếu
  + Tại thời điểm đó, nó sẽ tự động xóa tất cả các weak references đến đối tượng đó và tất cả các Weak Reference đến bất kỳ đối tượng nào có khả năng tiếp cận yếu khác mà từ đó đối tượng đó có thể truy cập được thông qua một chuỗi các tham chiếu mạnh và mềm.
  + Đồng thời, nó sẽ khai báo tất cả các đối tượng có thể truy cập yếu trước đây là có thể hoàn thiện.Đồng thời hoặc một thời gian sau, nó sẽ xếp hàng đợi những tham chiếu yếu mới được xóa đã được đăng ký với hàng đợi tham chiếu.



*Hình 3. 6 Khắc phục memory leak khi sử dụng Static Async Task Activity*

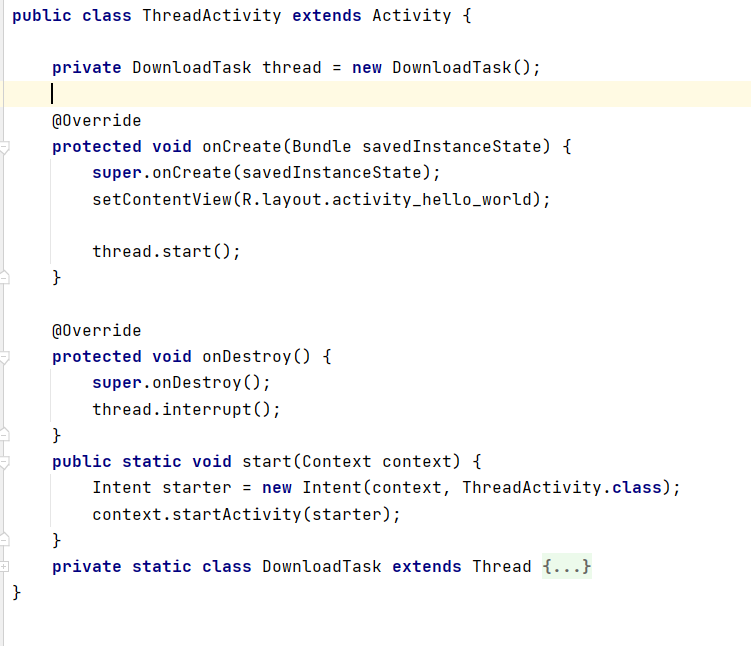
## **3.4. ThreadActivity**

* Là việc Task được khai báo là as non-static class nên nó sẽ giữ tham chiếu của activity khiến activity không đủ điều kiện để thu gom rác. Nếu Task được thực hiện trước khi rotate/close Activity, mọi thứ sẽ ổn mà không bị leak.



*Hình 3. 7 Memory leak khi sử dụng Thread Activity*

* Để khắc phục trương hợp này ta cho task được thực hiện trước đó để chuyển sang Activity khác hoặc rotate thiết bị. Từ đó mọi thứ sẽ hoạt động tốt mà không bị leak



*Hình 3. 8 Khắc phục memory leak khi sử dụng Thread Activity*

# **TÀI LIỆU THAM KHẢO**

**Trang web tham khảo:**

http://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/

https://developer.android.com/…/perfor…/memory-overview.html <https://developer.android.com/topic/performance/memory.html>

https://www.androidpit.com/how-to-manage-memory-on-android

**Sách và tài liệu tham khảo:**

High Performance Android Apps

Android Memory Optimization

<https://www.researchgate.net/publication/331133864>