🟢 MỨC DỄ (FRESHER)

1. Memory leak là gì?

Trả lời: Memory leak là tình trạng mà một đối tượng (object) **không còn được sử dụng**, nhưng **vẫn được giữ tham chiếu (reference)** bởi một đối tượng khác có vòng đời dài hơn. Kết quả là Garbage Collector **không thể thu hồi bộ nhớ của đối tượng đó**, khiến ứng dụng tiêu tốn RAM, chậm dần và có thể bị **OutOfMemoryError**.

---

2. Những nguyên nhân phổ biến gây memory leak trong Android?

Trả lời: Các nguyên nhân thường gặp:

- **Context leak**: giữ Activity context trong Singleton hoặc static field.
- **Static reference leak**: giữ object có lifecycle ngắn trong biến static có lifecycle dài.
- **Handler/Runnable**: sử dụng postDelayed và không huỷ khi Activity bị destroy.
- **Listener/Callback không gỡ bỏ**: đăng ký listener nhưng không remove.
- **Fragment View leak**: không set `binding = null` trong `onDestroyView()`.
- **Thread hoặc Coroutine sống lâu hơn Activity** nhưng vẫn giữ reference của Activity.

---

3. Context nào dễ gây leak nhất?

Trả lời: Activity context dễ gây leak nhất vì vòng đời của Activity ngắn. Nếu giữ Activity context trong các object sống lâu hơn (Singleton, static, ViewModel, Thread), Activity sẽ không được thu hồi và gây memory leak. Ngược lại, **Application context** có vòng đời bằng vòng đời của ứng dụng → an toàn hơn.

---

4. Sự khác nhau giữa ApplicationContext và ActivityContext?

Trả lời:

Đặc điểm	ApplicationContext	ActivityContext
Vòng đời	Sống suốt vòng đời app	Sống theo Activity
Dùng cho	Singleton, Manager	UI, View, Dialog
Leak	An toàn	Dễ leak nếu dùng sai
Theme	Không hỗ trợ UI	Có hỗ trợ UI

→ Kết luận: dùng ActivityContext cho UI, ApplicationContext cho các đối tượng sống lâu.

---

5. Khi nào cần set binding = null trong Fragment?

Trả lời: Fragment có vòng đời view ngắn hơn vòng đời Fragment, vì vậy trong `onDestroyView()`, bạn cần set:

```
binding = null
```

để giải phóng toàn bộ view hierarchy. Nếu không, Fragment sẽ giữ các View cũ lại → gây **memory leak**, đặc biệt khi dùng Navigation Component và nhiều màn hình.

---

6. Công dụng của Android Profiler?

Trả lời: Android Profiler là công cụ trong Android Studio giúp theo dõi:

- **Memory**: lượng RAM app sử dụng, thấy object, heap dump, GC events.
- **CPU**: thread hoạt động, thời gian xử lý, trace call.
- **Network**: request/response, dung lượng, tốc độ.
- **Energy**: mức độ sử dụng battery.

Nó giúp phát hiện memory leak, bottleneck và tối ưu hiệu năng.

---

7. LeakCanary dùng để làm gì?

Trả lời: LeakCanary là thư viện phát hiện rò rỉ bộ nhớ tự động. Khi một Activity/Fragment bị leak, LeakCanary:

1. Chụp heap
2. Phân tích leak
3. Hiển thị thông báo
4. Cung cấp **leak chain** cho biết object nào giữ reference khiến leak xảy ra

Đây là công cụ gần như bắt buộc khi debug ứng dụng thật.

---

8. DiffUtil là gì?

Trả lời: DiffUtil là utility class dùng để:

- So sánh 2 list cũ và mới
- Tìm ra item nào thay đổi / thêm / xoá / di chuyển
- Cập nhật RecyclerView một cách tối ưu

→ Chỉ update item cần thiết, giúp **hiệu năng cao hơn notifyDataSetChanged()**.

---

9. ListAdapter khác gì RecyclerView.Adapter?

Trả lời: ListAdapter:

- Tích hợp **DiffUtil.Callback** tự động
- Chạy diff computation trên background thread
- Tối ưu hiệu năng khi cập nhật list lớn
- Đơn giản hoá code

RecyclerView.Adapter không có DiffUtil → phải tự viết hoặc dùng notifyDataSetChanged() → gây lag.

---

10. Pagination dùng để làm gì?

Trả lời: Pagination giúp:

- Tải dữ liệu từng trang thay vì toàn bộ cùng lúc
- Giảm sử dụng RAM
- Giảm chi phí network
- Giúp RecyclerView load mượt
- Trải nghiệm người dùng tốt hơn (scroll vô hạn)

Pagination rất quan trọng khi làm việc với API trả về danh sách lớn.

🟠 MỨC TRUNG BÌNH (JUNIOR)

---

11. Vì sao static reference dễ gây memory leak trong Android?

Trả lời: Biến static thuộc về **ClassLoader**, sống suốt vòng đời của ứng dụng. Nếu bạn lưu Activity hoặc Fragment trong biến static, Activity sẽ không bao giờ được xóa khỏi bộ nhớ, dù user đã quay lại hoặc thoát màn hình.

```
class MainActivity {
    companion object {
        var instance: MainActivity? = null   // ❌ leak
    }
}
```

Static giữ reference → Activity không được GC thu hồi → memory leak.

---

12. Làm sao để tránh Context leak trong Singleton?

Trả lời: Không được truyền Activity context vào Singleton. Chỉ truyền **ApplicationContext**:

```
object Manager {
    fun init(context: Context) {
        this.context = context.applicationContext // ✔ an toàn
    }
}
```

ActivityContext gắn với UI → dễ leak khi Activity bị destroy.

---

13. Vì sao Handler.postDelayed có thể gây leak? Cách xử lý?

Trả lời: postDelayed giữ Runnable → Runnable giữ Activity reference thông qua lambda → Handler còn sống → Activity không thể bị thu hồi.

Cách xử lý:

- Dùng `lifecycleScope` thay cho Handler
- Hoặc trong `onDestroy()` gọi `handler.removeCallbacksAndMessages(null)`
- Hoặc dùng WeakReference để tránh giữ mạnh tới Activity

---

14. Tại sao Fragment dễ leak hơn Activity?

Trả lời: Vì việc destroy View của Fragment (onDestroyView) xảy ra **sớm hơn** destroy chính Fragment (onDestroy). Nếu giữ binding hoặc view trong field:

```
private var binding: FragmentAabbBinding? = null
```

Và không set null → giữ nguyên DOM cũ → leak view tree.

---

15. Khi nào nên dùng WeakReference?

Trả lời: WeakReference dùng khi:

- Cần giữ reference tới Activity/Fragment nhưng **không muốn ngăn GC thu hồi**
- Khi truyền Activity vào nền tảng sống lâu hơn như Thread, Runnable, Background task.

WeakReference giúp tránh leak, nhưng không phải giải pháp cho mọi leak.

---

16. DiffUtil tính toán như thế nào?

Trả lời: DiffUtil so sánh:

1. `areItemsTheSame()` → kiểm tra ID (ví dụ: user.id)
2. `areContentsTheSame()` → kiểm tra nội dung (name, age, email)
3. Tạo danh sách cập nhật tối thiểu
4. Apply thay đổi (insert, remove, update, move)

Nhờ đó RecyclerView chỉ update phần tử thật sự thay đổi → rất hiệu quả.

---

17. Vì sao không nên dùng notifyDataSetChanged()?

Trả lời:

- Update toàn bộ list, kể cả item không đổi
- Tốn CPU để re-bind toàn bộ

- Mất animation
- Dễ gây lag khi list lớn (1000+ items)

DiffUtil và ListAdapter giải quyết các vấn đề này.

---

18. Vì sao ListAdapter giúp UI mượt hơn?

Trả lời: ListAdapter:

- Chạy diff calculation ở background thread
- Chỉ update item thay đổi
- Dispatch update dưới dạng animation
- Không block UI thread

Kết quả: scroll mượt, update list nhanh, không drop frame.

---

19. PagingSource trong Paging 3 có nhiệm vụ gì?

Trả lời: PagingSource mô tả:

- Cách load dữ liệu theo trang
- Từ đâu (API hoặc Room)
- Điều kiện nextKey, prevKey
- Xử lý lỗi mạng
- Trả về LoadResult.Page

PagingSource là "trái tim của Pagination".

---

20. cachedIn(viewModelScope) dùng để làm gì trong Paging 3?

Trả lời: cachedIn:

- Cache flow của Paging
- Giúp khi rotate màn hình hoặc quay lại list, không cần load lại từ trang 1
- Tiết kiệm network
- Tránh mất thời gian chờ đầu trang

Paging 3 + cachedIn → UI cực kỳ mượt.

🟠 MỨC KHÓ (MIDDLE – GHI ĐIỂM)

---

21. Giải thích rõ tại sao SharedPreferences có thể gây ANR?

Trả lời: `SharedPreferences.commit()` ghi dữ liệu **đồng bộ** vào file XML trên main thread. Nếu file lớn hoặc ghi nhiều lần liên tục → main thread bị block → app đứng → ANR.

Ngay cả `.apply()` cũng có thể gây delay vì commit chạy trên background nhưng write vẫn lock file → block thread khác.

Do đó Google khuyên dùng **DataStore** thay thế.

---

22. Vì sao DataStore không gây ANR như SharedPreferences?

Trả lời: DataStore:

- Dùng coroutine
- Thực thi I/O trên Dispatchers.IO
- Ghi/đọc bất đồng bộ
- Có cơ chế concurrency safe, không lock UI thread
- Xây dựng bằng Flow nên lazy và nhẹ

→ Không bao giờ block main thread → tránh ANR.

---

23. Giải thích leak chain mà LeakCanary cung cấp?

Trả lời: Leak chain là chuỗi tham chiếu giữ đối tượng bị leak. Ví dụ: `ViewModel → Repository → Singleton → Activity`

LeakCanary hiển thị từng bước trong chain:

- **Leaking object:** Activity
- **Held by:** inner class, anonymous class, lambda…
- **strong reference path:** từ Singleton đến Activity

Nó giúp developer biết chính xác đoạn code nào gây leak.

---

24. Memory leak có xảy ra trong ViewModel không? Ví dụ?

Trả lời: Có. Nếu ViewModel giữ ActivityContext hoặc Fragment reference.

```
class MyViewModel(val context: Context) : ViewModel() // ❌ leak
```

ViewModel sống lâu hơn Activity → Activity không được GC thu hồi.

Cách đúng: Dùng AndroidViewModel:

```
class MyViewModel(app: Application) : AndroidViewModel(app) // ✔ dùng ApplicationContext
```

---

25. Dùng Glide sai cách có thể gây memory leak như thế nào?

Trả lời: Glide request gắn với lifecycle của Context. Nếu dùng:

`Glide.with(context.applicationContext)` trong khi load vào ImageView của Activity → request không biết khi nào Activity destroy → có thể giữ view lâu → leak.

Hoặc không cancel request khi Fragment bị destroy.

Cách đúng:

- `Glide.with(this)` trong Activity
- `Glide.with(viewLifecycleOwner)` trong Fragment
- Không dùng applicationContext khi load vào UI component.

---

26. Khi nào WeakReference không giải quyết được memory leak?

Trả lời: Khi leak không nằm ở object bạn quấn WeakReference, mà nằm ở **strong reference phía trên** trong chain.

WeakReference chỉ giúp đối tượng không bị giữ mạnh, nhưng:

- Nếu một object khác (static, singleton) giữ Activity mạnh → leak vẫn xảy ra
- WeakReference không xoá chain leak

WeakReference chỉ tránh leak, không sửa leak gốc.

---

27. DiffUtil có thể gây lag không? Nếu có, khi nào?

Trả lời: Có. DiffUtil chạy thuật toán tính toán diff với độ phức tạp O(n). Nếu list quá lớn (10.000+ item) hoặc cập nhật list liên tục mỗi vài millisecond → DiffUtil chạy nhiều lần → chiếm CPU.

Giải pháp:

- Throttle update
- Sử dụng AsyncListDiffer
- Chunk update list thành phần nhỏ
- Dùng Room + Paging 3 để chỉ load phần cần thiết
