MEMORY MANAGEMENT IN JAVA

GC Capsule

SUBJECT:	Memory Management & Garbage Collection in java	DATE:	MM/DD/YYYY	

> **“Objects in memory are closer than they appear.”**
> — GC

1. What is Java Memory Management?

Java manages memory automatically using the JVM. It allocates memory to objects and reclaims it using **Garbage Collection (GC)**.

🧠 **Key Areas**:

- **Heap** (Objects)
- **Stack** (Method calls)
- **Metaspace** (Class metadata)

✅ *Recommended*: Let JVM handle memory; fine-tune GC only if needed.

2. What is Garbage Collection in Java?

GC is the process of **reclaiming memory** used by unreachable objects.

📌 JVM tracks object references. If no references → object is GC'ed.

🔄 GC is automatic, but you can **trigger manually**:

```
System.gc();      // Not recommended in production
```

3. What is Heap Memory?

📦 **Heap** stores dynamically created objects.

🧠 **Divided into**:

- **Young Generation (Eden + Survivor)**
- **Old Generation**
- **Metaspace** (Java 8+)

🟢 *Recommended*: Monitor with -Xmx, -Xms.

4. What is Stack Memory?

📌 Used for method execution, local variables, and references.

Each thread has its own stack.

```
void print() {
   int x = 10; // Stored in stack
}
```

❌ StackOverFlowError occurs if stack is too deep (e.g., infinite recursion).

5. Difference: Heap vs Stack?

Feature	Heap	Stack
Lifetime	Until GC	Until method ends
Scope	Global (across threads)	Per thread
Speed	Slower	Faster

6. What is Young Generation and Old Generation in Java?

Java breaks the memory (Heap) into two main parts to manage objects more efficiently:

👶 **1. Young Generation (New Objects)**

- All **new objects** are created here.
- It's like a **play school** – kids (objects) come and go quickly.

- It has 3 parts:
  - **Eden**: Where objects are first created.
  - **Survivor 1 & 2**: Temporary holding areas for objects that survive once.

👴 **2. Old Generation (Long-Lived Objects)**

- Objects that stay alive for a **longer time** (survive multiple cleanups in Young Gen) are moved here.
- Think of it as a **retirement home** for old people (long-living objects).
- It takes **more time to clean**, and causes app pauses when cleaned (called Major GC).

🔄 **Object Lifecycle in Simple Steps:**

1. You create an object → it goes to **Eden**.
2. If it's still needed after Minor GC → moves to **Survivor**.
3. If it still lives after a few more Minor GCs → promoted to **Old Generation**.

🧪 Code Example 1: Short-Lived Object (Young Gen)

```java
class ShortLived {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {

            String temp = new String(original:"Hello"); // New object created in Eden
        }
        // temp is not used again → eligible for Minor GC
    }
}
```

These temp strings are short-lived → GC will clean them quickly from Eden.

🧪 Code Example 2: Long-Lived Object (Old Gen)

```java
public class LongLived {
    static List<String> list = new ArrayList<>();

    public static void main(String[] args) {
        for (int i = 0; i < 100000; i++) {
            list.add("Data-" + i);  // Keeps growing
        }
        // list stays in memory → moved to Old Gen eventually
    }
}
```

Since list stays in memory and grows, it's not collected early → moved to Old Generation.

✅ **Recommended:**

- Keep short-lived objects short-lived (don't hold unnecessary references).
- Avoid memory leaks so that Major GC happens less often.
- Monitor GC behavior in production (use VisualVM, JConsole, etc.).

7. What is Minor vs Major GC?

🧼 **Minor GC**: Cleans Young Gen → fast and frequent
♻️ **Major GC (Full GC)**: Cleans Old Gen → slower

🟠 Full GC pauses the app → avoid too many Full GCs.

8. When is an object eligible for GC?

📌 When **no reference points to it**:

```
MyClass obj = new MyClass();

obj = null; // Now eligible for GC
```

✅ *Tip*: Set large unused objects to null for early cleanup.

9. What Are Strong, Weak, Soft & Phantom References in Java?

Java has **4 types of references** — they control how the **Garbage Collector (GC)** treats an object.

Think of it like how tightly you're **holding an object in your hand**, and GC is trying to take it away 🧹

Strong Reference – "Holding it TIGHTLY" 💪

- Most common type.
- As long as you hold it → GC **can't collect** it.
- You must **set it to null** before GC can collect.

🧪 **Example:**

```
String name = new String("Java");  // Strong reference
```

📌 **GC can't touch it** unless you say: name = null;

Weak Reference – "Loose Hold" 🧤

- GC can collect it **anytime** if no strong refs exist.
- Used when you **don't want to prevent GC** from cleaning up.

🧪 **Example:**

```
WeakReference<String> weakRef = new WeakReference<>(new String("Weak"));
```

📌 GC will remove "Weak" if it's not strongly referenced elsewhere.

Soft Reference – "Gentle Hold" 🌬️

- GC **tries not to collect it**, unless memory is low.
- Useful for **caching**.

🧪 **Example:**

```
SoftReference<String> softRef = new SoftReference<>(new String("Soft"));
```

📌 GC keeps "Soft" until memory runs low.

Phantom Reference – "Ghost Mode" 👻

- GC already collected the object → this is just a **notification**.
- You can't get the object anymore.

- Used for **cleanup work after GC**.

Reference Type	Collected By GC When?	Use Case
Strong	Never (unless null)	Default object refs
Soft	When memory is low	Caching (LRU cache)
Weak	Anytime if no strong refs	Maps, memory-sensitive
Phantom	After GC collects it (notify)	Post-cleanup hooks

10. How do you tune the JVM memory?

-Xms512m -Xmx2048m

-XX:MaxGCPauseMillis=200

- Xms: Initial Heap
- Xmx: Max Heap

✅ *Tip*: Avoid Full GC by monitoring memory usage.

11. What tools help monitor memory?

🛠️ JVM Profilers:

- **VisualVM**
- **jconsole**
- **JProfiler**
- **Eclipse MAT**
- jmap, jstat, jvisualvm

✅ Always monitor memory during stress/load testing.

12. What is OutOfMemoryError?

Occurs when JVM **can't allocate memory** even after GC.

```
List<byte[]> list = new ArrayList<>();

while (true) list.add(new byte[1024 * 1024]); // Boom 💥
```

🛡️ Avoid memory leaks and use monitoring tools.

13. What is Memory Leak in Java?

🔹 **Meaning:** When objects are **not used anymore but still referenced**, so GC can't clean them.

📌 **Why it's bad?**

- Wastes memory
- Slows down application
- Can cause OutOfMemoryError

🧪 **Example:**

```
List<String> list = new ArrayList<>();

while (true) {

   list.add("Leak"); // Keeps growing → GC can't clean it

}
```

✅ **Tip:** Remove unused objects or set them to null.

14. What are JVM Memory Areas?

Memory Area	Purpose
Heap	Stores objects
Stack	Stores method calls & local variables
Metaspace	Stores class metadata (Java 8+)

Memory Area	Purpose
Code Cache	Stores JIT-compiled bytecode

🧠 **Heap** is the main area for GC to work.

15. GC Process Overview :

1. New Object → Young Gen (Eden)
2. Survive GCs → Moved to Survivor
3. Live long enough → Promoted to Old Gen
4. Old Gen full → Major (Full) GC
5. Unreachable objects → Collected