perplexity

What is Garbage Collection (GC)?

Garbage Collection in Java is the automatic process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. You don't have to explicitly allocate and de-allocate memory (like in C++), avoiding common errors like memory leaks and dangling pointers.[1] [2]

How does it work?

The JVM uses a **Mark and Sweep** algorithm (and its variations).[3] [4]

1. **Marking (Identify):**
    - The GC scans memory starting from "GC Roots" (active threads, static variables, local variables).
    - It follows references to find every object that is still reachable.
    - Any object *not* reached is marked as "garbage".
2. **Sweeping (Delete):**
    - The GC clears the memory occupied by the unmarked (unreachable) objects.
3. **Compacting (Optional but common):**
    - After deleting garbage, the remaining objects are moved together to make new memory allocation faster (removing holes in the heap).[2] [5]

Generational Strategy

Most modern GCs (like G1GC or Parallel GC) divide the Heap into generations to be more efficient:[3]

- **Young Generation (Eden + Survivor):** Where new objects are born. Most objects die here quickly (e.g., variables inside a loop). GC here is frequent and fast ("Minor GC").
- **Old Generation (Tenured):** Objects that survive many Minor GCs are moved here. GC here is rare but slow ("Major GC").

Have I ever tuned GC? (Real-world Scenario)

Yes. In a high-throughput microservice processing financial transactions, we faced "Stop-the-World" pauses where the application froze for 2 seconds, causing timeouts.

The Problem:
We were using the default GC (Parallel GC at the time), which pauses application threads to

clean the Old Generation. Our application created millions of short-lived objects (JSON parsing), filling the Young Gen too fast and spilling into Old Gen.

The Tuning Strategy:

1. **Switch to G1GC:** We switched to the **G1 Garbage Collector** (`-XX:+UseG1GC`), which is designed to minimize pause times by cleaning memory in small chunks (regions) rather than the whole heap at once.[2]
2. **Adjust Goal:** We set a strict pause time target (`-XX:MaxGCPauseMillis=200`). This told the JVM: "I don't care if you have to run GC more often, but never pause for more than 200ms."
3. **Increase Young Gen:** We increased the size of the Young Generation (`-XX:NewRatio`) so that short-lived objects had enough space to die there *before* being promoted to the Old Generation.

Result: The long 2-second freezes disappeared, and the 99th percentile latency dropped significantly.

⁂

1. https://www.geeksforgeeks.org/java/garbage-collection-in-java/
2. https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/
3. https://newrelic.com/blog/apm/java-garbage-collection
4. https://stackoverflow.com/questions/3798424/what-is-the-garbage-collector-in-java
5. https://dev.to/brilworks/garbage-collection-in-java-explained-8pg
6. https://www.youtube.com/watch?v=Mlbyft_MFYM
7. https://www.ibm.com/think/topics/garbage-collection-java
8. https://stackify.com/what-is-java-garbage-collection/
9. https://www.eginnovations.com/blog/what-is-garbage-collection-java/
10. https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html