♻️🧠 JAVA GARBAGE COLLECTORS — THE OPTIONS, IN PLAIN ENGLISH

@vvauban 20251021

⏱️ TL;DR

Java has multiple GCs with different trade-offs:

- **G1** is the *default* general-purpose GC. inside.java
- **ZGC** & **Shenandoah** focus on super-low pauses (concurrent, region-based). Generational modes are now mainstream in recent JDKs. openjdk.org
- **Parallel** maximizes throughput with longer pauses.

- **Serial** is simple/small, OK for tiny apps.

- **Epsilon** does no collection (use for testing/special cases). openjdk.org
  Pick based on **latency vs throughput vs heap size** (and your JDK vendor).

🤔 HOW JAVA MEMORY & GC WORKS (QUICK VIEW)

- Objects start in **Young Gen** (Eden ➜ Survivor S0/S1). Survivors eventually **promote** to **Old Gen**.

- **Minor GC** collects young space; **Major/Old GC** handles old space; some GCs also **compact** to avoid fragmentation.

- **Metaspace** holds class metadata (replaced PermGen since Java 8).

🗑️ THE COLLECTORS (WHAT, WHEN, WHY)

🤏 Serial GC

- **Single-threaded**, stops the world.
- Good for **small heaps**, **single-CPU** or very small container footprints.

🔀 Parallel GC (Throughput)

- **Multi-threaded** stop-the-world collector for **max throughput**.
- Accept **longer pauses** to keep overall work high—common for batch jobs.

🧹 CMS (Concurrent Mark-Sweep)

- Legacy low-pause GC (removed in modern JDKs). Prefer **G1** or **low-latency GCs** today.

1️⃣ G1 (Garbage-First)

Default GC since Java 9; balances **latency & throughput** on multi-core, large heaps. inside.java

▪ Region-based, does **concurrent marking** and **copy/compaction** to limit fragmentation.

▪ Tuning target: **max pause time goal**.

0ZGC

- **Ultra-low pause** (≲10ms) with **concurrent** mark, relocate, and compact; scales to **very large heaps**.
- **Generational ZGC** is the **default ZGC mode** in recent JDKs (non-generational is deprecated/removed). Use `-XX:+UseZGC`. [openjdk.org]
- Minimal tuning: enable it, size the heap, go.

🌲 Shenandoah

- **Low-pause** concurrent GC with **concurrent compaction**; great for predictable latencies.
- **Generational Shenandoah** became a **product feature** in JDK 25 (not just experimental). Default mode remains single-gen unless configured. openjdk.org
- Note: **Oracle JDK does not ship Shenandoah**; use vendors like **Red Hat/Temurin/Corretto/Azul** if you need it. wiki.openjdk.org

❌ Epsilon (No-Op)

- **Alloc-only** GC — performs **no reclamation**; JVM exits when heap is exhausted.
- Useful for **performance baselining**, **short-lived apps**, or **GC-free experiments**. It **stays experimental** on purp

openjdk.org

🤷‍♂️ HOW TO CHOOSE (RULE OF THUMB)

- **“I want sane defaults”** → Start with **G1**. inside.java
- **“I need ultra-low pause times”** (APIs, trading, real-timey flows) → Try **ZGC**; consider **Shenandoah** if your vendor ships it. openjdk.org+1
- **“I maximize throughput on batch jobs”** → **Parallel GC**.
- **“Tiny container or single core”** → **Serial GC** can be fine.
- **“I’m benchmarking GC impact”** → **Epsilon** (carefully).

☝️ PRACTICAL TIPS

- **Measure first**: enable GC logs (`-Xlog:gc*`) and track **pause P95/P99**, **allocation rate**, **survivor promotions**, **old-gen growth**.
- **Start with defaults**, then tune pause targets (G1) or just heap sizing (ZGC/Shenandoah).
- **Match vendor & version** to the GC features you need (e.g., Shenandoah availability). wiki.openjdk.org

What do you use in prod—G1, ZGC, Shenandoah, or Parallel? Why? Drop your context (heap size, SLA, traffic) in the comments 👇

#Java #JVM #Performance #GarbageCollection #ZGC #Shenandoah #G 5 #LowLatency #DevOps