1. Garbage Collector

The **Java Garbage Collector (GC)** is a **JVM subsystem** that automatically manages memory.

- Java applications create objects dynamically in the **heap** using new.
- When an object is no longer accessible (no reference path from **GC roots**), it is considered *garbage*.
- GC **frees that memory** so it can be reused.

 Key Point: Developers don't call free() (like in C/C++). Memory is reclaimed automatically.

---

2. Real-Life Analogy

Imagine an **office workspace**:

- Workers (Java threads) create documents (objects).
- When documents are no longer needed, they're left on desks.
- The janitor (GC) periodically comes in:
    1. Checks which documents are still in use (mark).
    2. Discards the useless ones (sweep).
    3. Rearranges desks to avoid gaps (compact).

Different janitors (GC algorithms) have different **cleaning strategies**:

- Some pause everyone while cleaning.
- Some clean while workers continue working.
- Some focus on the dirtiest desks first.

Internal working of Garbage Collection (GC),

emphasizing the common Mark–Sweep–Compact process:

Internal Working of Garbage Collection (Mark–Sweep–Compact Algorithm)

1. **Mark Phase**

- **Starting Point:** GC roots are identified:
- Local variables on the stack (associated with active method calls and threads)
- Static variables (associated with loaded classes)
- JNI (native) references (objects linked with native code)
- Active threads themselves
- **Traversal:** The GC traces the object graph, beginning from these roots, recursively following references to find all objects that are reachable (directly or indirectly).
- **Action:** Each reachable object is “marked” (its internal flag or bit is set), indicating it is "alive".

2. **Sweep Phase**

- **Identification:** Any objects in the heap not marked in the previous phase are considered unreachable (i.e., garbage).
- **Memory Reclamation:** The GC traverses all objects in the heap, frees memory used by unmarked objects, and resets the mark bits on marked objects for next cycles.

3. **Compact Phase (Optional)**

- **Objective:** Surviving (live) objects are moved together into contiguous memory locations.
- **Benefits:** This eliminates heap fragmentation, making allocation of new objects faster and more efficient since future memory requests can often be satisfied more quickly when free memory is one contiguous block.

How the JVM Knows Where GC Roots Are

- **Stack Frames:** The JVM tracks which local variables (in stack frames) are reference types.
- **Object Headers:** Every object has metadata detailing type information, so the GC can discover which fields are references.
- **Native and Static References:** The JVM maintains records of static variables, JNI handles, and active threads during program execution, ensuring all potential roots are found.

Why the Mark–Sweep–Compact Approach?

- **Handles Cyclic References:** Objects referring to each other in a cycle (circular references) are correctly detected if they are unreachable from any root.
- **Controls Fragmentation:** Compaction phase prevents fragmentation without needing double the heap size as in some copying algorithms.
- **Efficiency:** By marking only live objects, sweeping everything else, and optionally compacting, this algorithm strikes a balance between memory efficiency and runtime performance.

4.JVM Heap Structure

When a Java program runs, all **objects** live inside the **Heap Memory**. This heap is managed by the **Garbage Collector (GC)** and divided into **different generations** to handle objects of different lifetimes efficiently.

---

1. Young Generation

□ Think of this as the "nursery" for new objects.

- **Eden Space**:
  - Every time you do new Object(), it goes to Eden.
  - Eden fills up quickly.
- **Survivor Spaces (S0 and S1)**:
  - When Eden is full, a **Minor GC** happens.
  - Dead objects are cleared; surviving objects are moved to **S0 or S1**.
  - The JVM alternates between S0 and S1: one is used, the other stays empty.
- **Promotion to Old Gen**:
  - If an object survives multiple Minor GCs, it gets a higher "age".
  - Once it crosses a threshold (usually 15), it is **promoted** to Old Gen.

□ **Minor GC is fast** because most young objects die quickly.

---

2. Old Generation (Tenured)

 This is like "long-term storage".

- Holds objects that lived long enough in the Young Gen.
- Larger than Young Gen, but GC here is **slower**.
- Uses algorithms like **Mark-Sweep-Compact** or region-based collection (G1).

 Major GC (or Old Gen GC):

- Cleans the Old Gen.
- Less frequent but **more expensive** (longer pauses).
- If Old Gen fills up → risk of **Full GC** or OutOfMemoryError.

---

3. Metaspace (Java 8+)

 This is **outside the heap** (unlike old PermGen).

- Stores **class metadata**:
  - Class definitions.
  - Method structures.
  - Constant pools.
- Grows dynamically, limited by **native memory**.
- If full → OutOfMemoryError: Metaspace.

---

4. Garbage Collection Types

- **Minor GC**:
  - Cleans only Young Gen (Eden + Survivors).
  - Fast, frequent.
- **Major GC**:
  - Cleans Old Gen.
  - Slower, less frequent.
- **Full GC**:
  - Cleans Young + Old + Metaspace.
  - Most expensive, usually **stop-the-world**.

---

5. Modern GC Algorithms

- **Serial GC** → Single-threaded, small heaps.
- **Parallel GC** → Multi-threaded, throughput-oriented.
- **CMS** → Low pause, deprecated.
- **G1 GC** → Splits heap into regions, balances pause time & throughput.
- **ZGC / Shenandoah** → Ultra-low pause collectors, scale to large heaps.

---

6. Why Split Heap into Generations?

Because **most Java objects die young**:

- Example: local variables, temporary data structures.
- Instead of scanning the whole heap, JVM collects **Young Gen separately**.
- This makes GC **much faster and more efficient**.

Types of GC Events in JVM

The Garbage Collector (GC) in the JVM reclaims memory by removing objects that are no longer reachable. GC events differ depending on which part of the heap they clean and how disruptive they are to application execution.

---

1. Minor GC

What it does:

- Operates only on the **Young Generation** (Eden + Survivor spaces).
- Triggered when **Eden Space is full** and no more memory can be allocated there.

How it works:

1. When Eden fills up, a Minor GC starts.
2. All application threads are paused briefly (**Stop-the-World event**).

3. GC scans Eden and finds all **live objects** (those still referenced).
4. These surviving objects are copied to one of the **Survivor spaces** (S0 or S1).
5. The other Survivor space stays empty (used in the next cycle).
6. Objects that have survived multiple Minor GCs (based on an "age" threshold, typically 15) are **promoted** to the Old Generation.

Characteristics:

- **Fast and frequent**, because most objects in Eden are short-lived and die quickly.
- Causes **short application pauses**, usually a few milliseconds.
- Helps keep the Young Generation efficient and uncluttered.

---

2. Major GC

What it does:

- Operates on the **Old Generation**.
- Triggered when the Old Generation is running out of space for promotions from Young Gen or allocations directly made in Old Gen.

How it works:

1. A Major GC scans the Old Generation for unreachable objects.
2. Uses algorithms like **Mark-Sweep-Compact** or region-based collection (e.g., G1).
   - **Mark**: Identify live objects by tracing references from GC roots.
   - **Sweep**: Remove dead objects.
   - **Compact**: Rearrange live objects to eliminate memory fragmentation.
3. After compaction, continuous memory regions are freed, making it easier to allocate large objects.

Characteristics:

- **Less frequent than Minor GC**, but **much slower** because Old Gen contains long-lived objects.
- Pauses are **longer** and can significantly impact application performance.

- If Old Gen fills up completely and cannot reclaim enough space, it can lead to **OutOfMemoryError**.

---

3. Full GC

What it does:

- Cleans the **entire heap** (Young Gen + Old Gen) and sometimes **Metaspace**.
- Triggered in severe cases, such as:
  - When Old Gen is full and compaction is needed.
  - When explicit System.gc() is called (not recommended).
  - When Metaspace runs out of space for class metadata.

How it works:

- Stops all application threads (**Stop-the-World event**).
- Performs collection and compaction across all generations.
- May also involve unloading classes if Metaspace is cleaned.

Characteristics:

- **Most expensive GC event** in terms of time and resources.
- Can cause application pauses from hundreds of milliseconds to several seconds, depending on heap size.
- If frequent Full GCs are occurring, it usually indicates **memory leaks** or poor heap tuning.

Summary Comparison

GC Event	Cleans	Frequency	Pause Duration	Triggered By
Minor GC	Young Generation	Very often	Short (ms)	Eden full

GC Event	Cleans	Frequency	Pause Duration	Triggered By
Major GC	Old Generation	Less often	Longer (ms–s)	Old Gen pressure
Full GC	Young + Old + Metaspace	Rare (critical)	Longest (ms–s+)	Severe memory pressure, System.gc()

6.Types of Garbage Collectors in JVM

Different applications have different needs: some want **maximum throughput** (finish work as fast as possible), while others want **low latency** (short pauses, smooth user experience). JVM provides multiple collectors to balance these needs.

A. Serial GC

- **How it works**:
  - Uses a single thread for both Minor and Major collections.
  - Entire application pauses during GC (Stop-the-World).
  - Uses **Mark-Sweep-Compact** for Old Gen, and **copying collection** for Young Gen.
- **Best suited for**:
  - Small heaps (<100 MB).
  - Single-threaded applications.
  - Embedded systems or simple desktop apps.
- **Trade-offs**:
  - Very simple and predictable.
  - Not scalable on multi-core systems.
- **Flag**: -XX:+UseSerialGC.

B. Parallel GC (Throughput Collector)

- **How it works**:

    - Uses multiple threads for **Young and Old Gen collection**.
    - Still **STW** (all application threads pause), but multiple GC threads speed up work.
    - Focus is on **maximizing throughput** (ratio of app time vs. GC time).
- **Best suited for**:
    - Applications where **pause time is less critical** (batch processing, data crunching).
    - Multi-core servers with medium to large heaps.
- **Trade-offs**:
    - Large pause times possible, especially with big heaps.
    - Not ideal for latency-sensitive applications.
- **Flag**: -XX:+UseParallelGC.

C. CMS (Concurrent Mark-Sweep)

(Deprecated in Java 9, Removed in Java 14)

- **How it works**:
    - Introduced as the first **low-pause collector**.
    - Splits GC into concurrent phases:
        1. Initial Mark (**STW**).
        2. Concurrent Mark (runs with app).
        3. Remark (**STW**).
        4. Concurrent Sweep (frees dead objects).
- **Best suited for**:
    - Interactive apps (web servers, UI apps) where shorter pauses matter.
- **Trade-offs**:
    - **Fragmentation** (no compaction).
    - High CPU usage (app + GC threads compete).
    - Complexity → replaced by G1 GC.
- **Flag**: -XX:+UseConcMarkSweepGC.

D. G1 GC (Garbage First)

Default collector since Java 9.

- **How it works**:
  - Heap is split into **regions** (not fixed Young/Old).
  - Collects **regions with most garbage first**.
  - Has **predictable pause times**, tunable with -XX:MaxGCPauseMillis.
  - Performs **compaction during collection**, avoiding fragmentation.
- **Best suited for**:
  - Large heaps (multi-GB).
  - Apps needing a balance between **throughput and low latency**.
  - General-purpose, now the default in modern Java.
- **Trade-offs**:
  - More overhead than Parallel GC in tiny heaps.
  - Tuning is more complex.
- **Flag**: -XX:+UseG1GC.

E. ZGC (Java 11+)

- **How it works**:
  - Designed for **ultra-low latency** (<10ms pauses).
  - Works with heaps from MBs up to **multiple terabytes**.
  - Uses **colored pointers** to track object states (removes the need for long STW phases).
  - Mostly concurrent: marking, relocation, and remapping all happen while app is running.
  - JDK 21 introduced **Generational ZGC**, improving performance by separating young/old objects.
- **Best suited for**:
  - Large heap, latency-sensitive systems (fintech, trading, real-time apps).
- **Trade-offs**:
  - Slightly lower throughput than Parallel or G1.
  - Relatively new; evolving.
- **Flag**: -XX:+UseZGC.

F. Shenandoah (Java 12+)

- **How it works**:

- Developed by Red Hat.
  - Focus: **pause time independent of heap size**.
  - Uses **Brooks pointers** for object indirection (extra pointer level).
  - Supports **concurrent compaction** → moves objects while app is running.
- **Best suited for**:
  - Very large heaps where pause times are critical.
  - Latency-sensitive workloads (similar to ZGC).
- **Trade-offs**:
  - Extra pointer dereference → slight CPU overhead.
  - Not as widely adopted as G1/ZGC.
- **Flag**: -XX:+UseShenandoahGC.

G. Epsilon GC (Java 11+)

- **How it works**:
  - A "no-op" GC.
  - Allocates memory but **never reclaims it**.
  - When heap is exhausted, app crashes.
- **Best suited for**:
  - **Performance testing** (to measure max throughput without GC).
  - Short-lived jobs (apps that finish before memory runs out).
- **Trade-offs**:
  - No memory reclamation → guaranteed crash in long-running apps.
- **Flag**: -XX:+UseEpsilonGC.

Summary Table

Collector	Focus	Pause Style	Heap Size Support	Use Case
Serial GC	Simplicity	Long, single-thread	Small heaps	Single-core, small apps
Parallel GC	Throughput	Long, multi-	Medium/Large	Batch jobs, data

Collector	Focus	Pause Style	Heap Size Support	Use Case
		thread		apps
CMS	Low-latency	Short, concurrent	Medium/Large	Web servers (obsolete)
G1 GC	Balanced, predictable	Tunable pauses	Large heaps	Default choice since Java 9
ZGC	Ultra-low latency	<10ms pauses	Huge heaps (TB)	Real-time, fintech
Shenandoah	Low latency, compact	Pause ≈ constant	Huge heaps	Enterprise, latency-sensitive
Epsilon GC	None (testing only)	No collection	Any	Benchmarking

7.GC Subtypes & Key Concepts

---

1. Generational Hypothesis

Core idea:

- **Most objects die young** (short-lived).
- Only a small fraction live long enough to need long-term storage.

How JVM uses it:

- JVM divides the heap into **Young Generation** and **Old Generation**.

- **Minor GCs** clean Young Gen frequently and cheaply (because most objects are garbage right away).
- **Promotion** happens only for survivors (to Old Gen).

Benefit:

- Reduces the need to scan the entire heap for garbage all the time.
- Makes garbage collection **much faster** and scalable.

---

2. Region-based GC

Instead of fixed divisions (Young vs Old), the heap is split into **smaller regions**.

- Used by **G1, ZGC, and Shenandoah**.
- Regions can be dynamically assigned to act as **Young or Old**, depending on need.
- GC collects **regions with most garbage first** (hence G1 = *Garbage First*).

Advantages:

- Flexible, heap can scale to very large sizes (hundreds of GB to TB).
- Allows concurrent collection (instead of full heap scans).
- Better memory locality → reduces fragmentation.

---

3. Reference Types

Java provides **different levels of references** to control GC behavior.

a) Strong Reference

- Example: Object o = new Object();
- Default type of reference in Java.
- GC will **never reclaim** a strongly referenced object until it goes completely out of scope.

b) Soft Reference

- Example: SoftReference<Object> ref = new SoftReference<>(obj);
- Used for **caching**.
- GC reclaims softly referenced objects **only if memory is low**.
- Prevents OutOfMemoryError while still keeping cached objects as long as possible.

c) Weak Reference

- Example: WeakReference<Object> ref = new WeakReference<>(obj);
- GC reclaims weakly referenced objects **as soon as no strong references exist**.
- Commonly used in **maps for caches** (e.g., WeakHashMap).

d) Phantom Reference

- Example: PhantomReference<Object> ref = new PhantomReference<>(obj, queue);
- Object is collected, but reference is enqueued in a **ReferenceQueue** after finalization.
- Used for **post-cleanup actions** (like freeing native resources).
- Unlike other references, get() always returns null.

Why These Concepts Matter

- **Generational hypothesis** explains *why* GCs split heap into Young vs Old.
- **Region-based GC** is a modern evolution that allows scaling to massive heaps with predictable pause times.
- **Reference types** give developers **fine-grained control** over object lifecycle, useful for caching, memory-sensitive apps, and resource cleanup.

8.GC Internals

---

1. Write Barriers

- **What**: Extra machine instructions that the JVM inserts whenever an object reference is updated.
- **Why**: Concurrent GCs (like G1, ZGC, Shenandoah) need to track changes while the application is still running. Without barriers, the GC could miss updates made during marking/collection.

Types:

- **Pre-write barrier** → triggered before a reference is overwritten.
- **Post-write barrier** → triggered after a new reference is stored.

Example (conceptually):

```
obj.field = newObj;
// Barrier runs here to inform GC about this change
```

- **Impact**:
    - Slight overhead per write, but crucial for correctness in concurrent GC.

---

2. Remembered Sets (RSet) in G1

- **Problem**: In region-based collectors (like G1), objects in one region can point to objects in another region. GC must know these cross-region references.
- **Solution**: Each region maintains a **Remembered Set** that records references *from outside into that region*.

How it works:

- Whenever a write updates a reference to point into another region → a **card table entry** is updated.
- During GC, the RSet allows scanning only relevant regions instead of the entire heap.
- **Benefit**:
    - Localizes GC work to regions with garbage.
    - Improves efficiency on large heaps.

---

3. SATB (Snapshot-at-the-Beginning)

- Used by **G1** (and similar concurrent collectors) during the **marking phase**.
- **Problem**: While GC is marking live objects, the application may keep creating/updating references, which could cause the GC to miss objects.

Solution:

- SATB takes a **logical snapshot of all live references at the beginning** of marking.
- Even if references change, the old value is remembered (via write barriers).
- Ensures that every object reachable at the start is considered live.
- **Benefit**:
    - Makes concurrent marking safe and correct.

---

4. TLAB (Thread-Local Allocation Buffers)

- **Problem**: If all threads allocate objects directly in Eden, they would contend for the same memory region (synchronization bottleneck).

Solution:

- JVM gives each thread its own **small chunk of Eden space** called a TLAB.
- Allocation inside a TLAB is just a simple pointer bump → **extremely fast**.
- When a TLAB fills up, the thread requests a new one from Eden.
- **Benefit**:
    - Reduces contention between threads.
    - Makes object allocation almost as fast as stack allocation.

---

5. Humongous Objects

- **What**: Objects larger than **50% of a region size** in G1 are called *humongous objects*.
- **How handled**:

    - Allocated directly into **Old Gen** (spanning multiple contiguous regions).
    - Avoids wasting Eden/Survivor space.
- **Problem**:
    - Humongous objects can fragment Old Gen.
    - G1 tries to reclaim them aggressively during mixed collections.
- **Examples**:
    - Large arrays, big byte buffers, image data, etc.

Why These Internals Matter

- **Write Barriers** → Enable concurrent GC without missing updates.
- **Remembered Sets** → Allow region-based GCs to scale efficiently.
- **SATB** → Ensures correctness during concurrent marking.
- **TLAB** → Makes allocation super fast and lock-free per thread.
- **Humongous Objects** → Special handling prevents them from overwhelming the Young Gen.

9. Tuning & Monitoring GC

Heap Size Tuning

- -Xms<size> → Initial heap size.
    - Setting it equal to -Xmx avoids dynamic heap resizing overhead.
- -Xmx<size> → Maximum heap size.
    - Too small → frequent GC, possible OOM.
    - Too large → long pause times if not using concurrent collectors.
- -Xmn<size> → Young Generation size.
    - Increasing Young Gen reduces Minor GC frequency but increases promotion pressure into Old Gen.

Tip: For large heaps (multi-GB), let the JVM auto-tune unless you have profiling data.

Collector Choice

- **Serial GC** (-XX:+UseSerialGC) → Single-threaded, predictable, best for small heaps.
- **Parallel GC** (-XX:+UseParallelGC) → High throughput, batch jobs.
- **G1 GC** (-XX:+UseG1GC) → Balanced, predictable pauses, default since Java 9.
- **ZGC** (-XX:+UseZGC) → Ultra-low pause, huge heaps (up to TBs).
- **Shenandoah GC** (-XX:+UseShenandoahGC) → Low latency, pause times independent of heap size.

Tip: Always test GC choices under **production-like loads**, don't rely only on defaults.

GC Logs (Unified Logging, JDK 9+)

- Example flag:
- -Xlog:gc*,gc+heap=debug:file=gc.log:time,uptime,level,tags
- Captures:
  - GC events (Minor, Major, Full).
  - Heap usage before/after.
  - Pause times.

Tip: Analyze logs with tools like GCViewer or GCeasy.io for human-friendly visualization.

Monitoring Tools

- **jconsole / VisualVM** → GUI-based, real-time monitoring of heap and GC.
- **jmap** → Creates heap dumps (jmap -dump:format=b,file=heap.hprof <pid>).
- **jstat** → Lightweight CLI, periodic GC stats (jstat -gc <pid> 1s).
- **Eclipse MAT (Memory Analyzer Tool)** → Post-mortem heap dump analysis, leak detection.
- **Java Flight Recorder (JFR) / JMC** → Advanced profiling (built into JVM).

---

10. Real-World Best Practices

Memory Management

- Use **try-with-resources** to automatically close I/O, DB connections, streams → prevents resource leaks.
- Release references when no longer needed (e.g., set large arrays/maps to null if not reused).
- Avoid **unbounded caches** → use WeakReference or SoftReference wrappers, or bounded caches (like Caffeine, Guava).
- For collections, choose proper data structures to minimize memory overhead (e.g., ArrayList vs LinkedList).

Profiling & Debugging

- Take **heap dumps** when memory pressure or OOM happens, analyze with Eclipse MAT or VisualVM.
- Enable **GC logging** in production (with rotation to avoid disk issues).
- Monitor **GC pause times**, **promotion rates**, **heap occupancy after GC**.
- Watch for **frequent Full GCs** → may indicate leaks or poor sizing.

GC Choice Based on Workload

- **Batch jobs / background tasks** → Parallel GC (max throughput).
- **Web servers / microservices** → G1 GC (predictable pauses, scalable).
- **Low-latency trading / fintech / real-time apps** → ZGC or Shenandoah (pause times <10ms).
- **Tiny apps / single-thread tools** → Serial GC.

General Tips

1. **Don't over-tune prematurely** — modern JVM defaults are strong (G1 is excellent for most apps).
2. **Always measure with real traffic** — synthetic benchmarks often mislead.

3. **Balance heap size vs. GC strategy** — larger heap = fewer GCs but longer pauses unless using ZGC/Shenandoah.
4. **Monitor promotion rate** — if Young Gen is too small, too many promotions → Old Gen fills quickly.
5. **Educate developers** — GC tuning can’t fix memory leaks caused by bad code.

□ **In summary**:

- Tune heap & GC flags based on profiling, not guesses.
- Use logging + monitoring tools for visibility.
- Apply coding best practices to minimize GC load.
- Choose the right collector for your workload (throughput vs. latency trade-off).
