# Garbage Collection

**Garbage Collection (GC)** in Java is the process by which the Java Virtual Machine (JVM) automatically removes **objects from memory** that are no longer used or accessible by the application. It helps prevent **memory leaks** and **OutOfMemoryErrors** by reclaiming heap space. It runs in the background as a daemon thread.

**Advantages of Garbage Collection**

* It makes java memory-efficient because the garbage collector removes the unreferenced objects from heap memory.
* It is automatically done by the garbage collector (a part of JVM), so we don't need extra effort.

**Disadvantages of Garbage Collection**

* Unpredictable pauses 🡪 GC can cause **"stop-the-world"** pauses, freezing all application threads.
* Lack of Control 🡪 Developers cannot explicitly control when and how memory is freed.
* Overhead 🡪 GC itself **consumes CPU resources** to identify and collect unreachable objects. This can **slow down application performance**, especially during full GCs.
* False Sence of Safety 🡪 GC reduces memory leaks but **doesn’t eliminate them completely**. Long-living references (like in static maps/lists) can still cause **leaks**.
* Memory Fragmentation (in some collectors) 🡪 Older collectors (e.g., CMS) may not compact memory, leading to fragmentation and inefficient heap usage.
* Higher Memory Usage 🡪 Java apps often require more memory than manual memory-managed languages to give GC room to operate. This increases infrastructure costs.

# Working of Garbage Collection

 **Mark**: Finds all objects reachable from GC Roots. All reachable (live) objects are marked. Unreachable ones are **left unmarked**.

 **Sweep**: Deletes unmarked (unreachable) objects.

 **Compact (optional)**: Rearranges memory to eliminate fragmentation. After sweeping, the heap may have **fragmented free spaces**. GC may **compact the heap** by moving live objects together and updating references. This improves **allocation efficiency**.

### Types of References 🡪

Java provides different types of references:

* **Strong Reference** – default reference type, GC never collects this unless explicitly set to null.
* **Soft Reference** – GC may collect these if memory is low.
* **Weak Reference** – GC collects them eagerly, even if memory is sufficient.
* **Phantom Reference** – Used to schedule post-mortem cleanup after GC.

## Types of Activities in Java Garbage Collection

Java heap is divided into generations:

* **Young Generation:** In this new objects are allocated.
* **Old Generation:**In this long-lived objects are stored.

Two types of garbage collection activities usually happen in Java. These are:

* **Minor or incremental Garbage Collection (GC)**: This occurs when unreachable objects in the Young Generation heap memory are removed.
* **Major or Full Garbage Collection (GC)**: This happens when objects that survived minor garbage collection are removed from the Old Generation heap memory. It occurs less frequently than minor garbage collection.

### 🔄 Minor vs Major GC

| **Type** | **Description** | **Affects** |
| --- | --- | --- |
| **Minor GC** | Collects only the **Young Generation** | Short pause, fast |
| **Major GC** | Collects **Old Generation** | Longer pause |
| **Full GC** | Collects entire heap + Metaspace | Most expensive |

# Making Objects Eligible for GC

An object is said to be eligible for garbage collection if it is unreachable.

* Nullifying the reference variable (obj = null).
* Re-assigning the reference variable (obj = new Object()).
* An object created inside the method (eligible after method execution).
* Island of Isolation (Objects that are isolated and not referenced by any reachable objects).

# Requesting Garbage Collection

* Once an object is eligible for garbage collection, it may not be destroyed immediately.The garbage collector runs at the JVM's discretion, and you cannot predict when it will occur.
* We can also request JVM to run Garbage Collector. There are two ways to do it :   
  + **Using System.gc()**: This static method requests the JVM to perform garbage collection.
  + **Using Runtime.getRuntime().gc():** This method also requests garbage collection through the Runtime class.

**Note:** There is no guarantee that the garbage collector will run immediately after these calls.

# The finalize() Method (Deprecated in Java 9+)

Before destroying an object, the garbage collector calls the [finalize()](https://www.geeksforgeeks.org/finalize-method-in-java-and-how-to-override-it/)method to perform cleanup activities. The method is defined in the [Object class](https://www.geeksforgeeks.org/object-class-in-java/)as follows:

*@Override*

*protected void finalize() throws Throwable {*

*System.out.println("GC cleaning up...");*

*}*

**Note:**

* finalize() method is deprecated since Java 9 because it is unpredictable and can cause performance issues.
* Alternatives like try-with-resources or explicit cleanup methods are preferred.
* The garbage collector calls finalize() at most once per object.
* Exceptions thrown in finalize() are ignored.

# Types of Garbage Collectors

## 1. Serial Garbage Collector

* **🧵 Single-threaded collection (both minor and major GC).**
* **🔍 Best for: Small applications or devices with single-core CPUs or limited memory.**
* **⚠️ Causes** long stop-the-world pauses**. Means when garbage collection is triggered, it pauses all running application threads.**

## 2. Parallel Garbage / throughput Collector

* Parallel Garbage Collector is the default garbage collector in Java 8.
* 🧵 Multi-threaded collection(usage multiple thread to perform GC task) for both young and old generations.
* 🔍 Best for: Applications where **throughput** and **high performance is more important than pause time(batch processing systems)**.
* ❗ Still **stop-the-world**, but faster.

## 3. Concurrent Mark-Sweep Collector

* 🕐 Minimizes pause times using concurrent(multiple threads) marking and sweeping(by doing most of its work while the application is still running).
* 🔍 Good for low-latency applications such as web servers and interactive applications.
* ⚠️ No compaction → memory fragmentation can occur.
* CMS Garbage collector will perform freezing of running threads i.e. application pause in two cases only:
  + While performing the garbage collection, If there is a change in heap memory in parallel.
  + While marking the referenced objects in the old generation space.

## 4. G1 Garbage Collector(JDK 7 – default in 9)

* Designed for applications with large heap sizes (greater than 4GB).
* 🔄 Divides heap into regions of equal size; cleans regions with most garbage first. i.e. Marks regions with active objects and prioritizes collection in regions with the most unused objects, hence the name "Garbage-First" (G1).
* 📈 Balances throughput and pause times.
* 🧠 Concurrent marking, compaction (Compacts free heap space after garbage collection, improving memory efficiency), and region-based cleanup.

## 5. Z Garbage Collector(experimental in 11, stable in 15+)

* It is designed to handle large heaps with minimal pause times, typically in the range of milliseconds.
* 🧠 Scalable, **sub-millisecond pause times**, even for huge heaps (GBs to TBs).
* 🔄 Mostly concurrent and non-intrusive.
* ✅ No compaction pauses (heap defragmentation is also concurrent).

## 6. Shenandoah Garbage Collector(java 12)

* It aims to reduce pause times by performing garbage collection concurrently with application threads.
* 🧠 Low-pause, concurrent compaction.
* 🔍 Focuses on **pause time consistency**.
* 📌 Introduced by Red Hat, available in OpenJDK (Java 12+).

**🔄 Categories of Garbage Collectors**

| **Collector** | **Target Use Case** | **JVM Flag** |
| --- | --- | --- |
| **Serial GC** | Small applications, single-threaded environments | -XX:+UseSerialGC |
| **Parallel GC (Throughput Collector)** | High-throughput apps with long-lived data | -XX:+UseParallelGC |
| **CMS (Concurrent Mark-Sweep)** *(Deprecated)* | Low-latency apps (Java 8 and before) | -XX:+UseConcMarkSweepGC |
| **G1 GC (Garbage First)** | Balanced pause time and throughput | -XX:+UseG1GC |
| **ZGC (Z Garbage Collector)** | Sub-millisecond pause time for large heaps | -XX:+UseZGC |
| **Shenandoah GC** | Low pause, scalable apps | -XX:+UseShenandoahGC |

# Memory Leaks in Java

An object that is no longer used by the application is still reachable and not garbage collected.

**🧠 Real-World Analogy**

Imagine forgetting to throw away an empty pizza box in your room. The box is useless but still takes space. If you never throw it away, soon your room (heap) will be full of trash (leaked objects), even though you're not using them.

## Why Do Memory Leaks Happen in Java?

We know that Java cleans up memory automatically with the help of garbage collector but still memory leaks can happen, this happens because our program keep holding onto things that are no longer needed. Our Java program still remembers these objects and thinks they are important and that's why it does not remove them and this wastes memory cause the problems.

**Example:**

// Demonstrating memory leaks in Java   
  
import java.util.ArrayList;  
import java.util.List;  
  
public class Geeks {  
 public static void main(String[] args) {  
 List<Object> list = new ArrayList<>();  
  
 while (true) {  
 Object obj = new Object();  
 // The list keeps growing and holds references to all objects  
 list.add(obj);  
 }  
 }  
}

**Explanation:**In the above example, the program keep on creating new object and each object is added to the list because list stores the references of all the objects and because of this memory usage keeps on growing and causes OutOfMemoryError.

### 1. Unclosed Resources🡪

Forgetting to close files, sockets, DB connections.

Connection con = dataSource.getConnection();

// no con.close() → leaked connection

✅ Use try-with-resources:

try (Connection con = dataSource.getConnection()) {

// use connection

}

### 2. Static References

* Static variables live for the entire application life.
* Referencing heavy or temporary objects can cause leaks.

static List<User> users = new ArrayList<>(); // grows forever if not managed

### 3. Listeners & Callbacks

* Registering event listeners without removing them.

button.addActionListener(listener); // never removed

✅ Always **deregister** listeners when done.

### 4. Large Caches (e.g., Maps, Lists)

* Caching without an eviction policy can hold objects indefinitely.

Map<String, Object> cache = new HashMap<>();

✅ Use:

new WeakHashMap<>(); // values will be GC'ed if no strong ref

Or a library like **Caffeine** with TTL/LRU eviction.

### 5. Thread Locals

* ThreadLocal variables may live as long as the thread.
* Especially bad in thread pools (like web servers).

ThreadLocal<SimpleDateFormat> sdf = new ThreadLocal<>();

✅ Always call threadLocal.remove(); after use.

### 6. Inner Classes Holding Outer Class Reference

* Anonymous or inner classes hold an implicit reference to the outer class.

class Outer {

void start() {

Runnable r = new Runnable() {

public void run() { /\*...\*/ }

};

}

}

✅ Use static inner classes if possible.

## Tools to Find Memory Leaks

* **VisualVM (comes with JDK)**
* **Eclipse Memory Analyzer (MAT)**
* **Java Mission Control**
* **YourKit Java Profiler**

## How to Avoid Memory Leaks?

* Stop keeping things that we do not need in our program, do not initialize unnecessary variables and list items to null.
* Do not let lists or caches keep growing forever without removing old stuff.
* It is always recommended to close files and database connections when we are done with them.

### 🧰 Best Practices to Avoid Memory Leaks

| **Practice** | **Description** |
| --- | --- |
| ✅ Use try-with-resources | Ensures auto-closing of IO, DB, network resources |
| ✅ Deregister listeners | Prevent long-lived references |
| ✅ Use weak references | For caches and observers |
| ✅ Limit static collections | Avoid global data growth |
| ✅ Monitor memory usage | Use profilers regularly |
| ✅ Use AutoCloseable | For all custom closeable resources |