**Java Heap Memory Structure**

Heap memory is divided into three main sections:

1. **Young Generation** (for new objects)
   * **Eden Space** → New objects are created here.
   * **Survivor Spaces (S0, S1)** → Objects that survive **Minor GC** are moved between these two spaces.
2. **Old Generation (Tenured)** (for long-lived objects)
   * Objects that survive multiple GC cycles in the Young Generation are **promoted** here.
   * A **Major GC (Full GC)** happens when this space fills up.
3. **Metaspace** (stores class metadata)
   * Stores **class structures, method bytecode, and static variables**.
   * Introduced in Java 8 (replacing **PermGen**).

**How Objects Move in Heap**

1. **New Object** → Created in **Eden Space**.
2. **Minor GC** → Moves surviving objects to **Survivor Space (S0, S1)**.
3. **Objects Survive Multiple GCs** → Moved to the **Old Generation**.
4. **Old Generation Fills Up** → **Major GC (Full GC)** is triggered.

**Garbage Collection Process**

* **Minor GC** → Cleans Young Generation (fast & frequent).
* **Major GC (Full GC)** → Cleans Old Generation (slow but necessary).
* **Metaspace Cleanup** → Unloads unused class metadata.

**How Garbage Collection (GC) Works Internally in Java?**

**1️. Steps of Garbage Collection (GC)**

GC works in **3 major steps** internally:

1. **Mark** → Identify unused objects.
2. **Sweep** → Remove those objects from memory.
3. **Compact** → Rearrange remaining objects to optimize memory usage.

**2️. Types of Garbage Collectors in Java**

Java provides different GC algorithms for different needs. The main ones are:

| **GC Algorithm** | **How It Works?** | **Best Use Case** |
| --- | --- | --- |
| **Serial GC** | Uses a **single thread** for GC (stop-the-world). | Best for **small apps** (single-core CPUs). |
| **Parallel GC** | Uses **multiple threads** for GC. | Best for **multi-core systems**. |
| **G1 GC** (Garbage First) | Divides heap into regions, collects garbage **incrementally**. | Best for **large applications** with low-latency needs. |
| **ZGC & Shenandoah GC** | Ultra-low-latency, collects garbage **concurrently**. | Best for **real-time systems**. |

**3️. Detailed GC Process (Internal Working)**

**Step 1: Object Allocation in Heap**

* When we create an object, it is allocated in **Eden Space** (Young Generation).
* If Eden fills up, **Minor GC** runs to clear unused objects.
* Surviving objects are moved to **Survivor Spaces (S0 & S1)**.

**Step 2: Minor GC (Young Generation Cleanup)**

* When Eden is **full**, Minor GC runs:
  1. **Unused objects** in Eden are deleted.
  2. **Surviving objects** move to Survivor Space **S0**.
  3. On the next Minor GC, they move to **S1**.
  4. If an object **survives multiple cycles**, it is **promoted to the Old Generation**.
* **Minor GC is very fast** because it happens only in the Young Generation.

**Example Code:**

public class MinorGCExample {

public static void main(String[] args) {

for (int i = 0; i < 100000; i++) {

byte[] data = new byte[1024 \* 1024]; // Creates objects in Eden

}

}

}

**Step 3: Major GC (Old Generation Cleanup)**

* If the **Old Generation fills up**, a **Major GC** (Full GC) runs.
* **Major GC is slow** because it scans the **entire heap memory**.
* **If Major GC fails to free memory**, Java throws **OutOfMemoryError**.

**Step 4: Compacting and Defragmentation**

* **After GC removes objects**, memory gets **fragmented**.
* **Compacting** moves all remaining objects together to **optimize space**.
* This **reduces fragmentation** and ensures new objects get allocated efficiently.

**4️. GC Logs: How to Monitor GC?**

To track how GC is working, we can enable GC logs using JVM options:

**Enable GC Logs (JVM Parameters)**

java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:gc.log -jar MyApp.jar

This will print GC activity in the gc.log file.

| **GC Process** | **Description** |
| --- | --- |
| **Marking** | Finds unused objects |
| **Sweeping** | Deletes unused objects |
| **Compacting** | Rearranges objects to optimize memory |
| **Minor GC** | Cleans Young Generation (fast) |
| **Major GC** | Cleans Old Generation (slow) |
| **GC Logs** | Helps analyze memory performance |

**Internal Working of Stack Memory in Java**

**🔹 What is Stack Memory?**

Stack memory is a **temporary, fast, and organized memory** that is used for method execution, storing local variables, and tracking method calls. It follows the **LIFO (Last-In, First-Out) principle**, meaning the last stored data is the first to be removed.

**🔹 How Stack Memory Works Internally?**

1. **Method Call Stack Formation**
   * Each time a method is called, a **stack frame** is created in stack memory.
   * The stack frame holds the **method’s local variables, parameters, intermediate calculations, and return address**.
   * When the method completes execution, its stack frame is **removed (popped)** from the stack.
2. **Structure of a Stack Frame**  
   A stack frame consists of:
   * **Method Metadata** → Stores method execution details.
   * **Local Variable Array** → Stores method parameters and local variables.
   * **Operand Stack** → Holds intermediate calculation values.
   * **Return Address** → Stores the location where execution should return after method completion.
3. **Stack Growth & Shrinkage**
   * When a **new method is called**, a new frame is **pushed** onto the stack.
   * When a method **returns**, its frame is **popped** from the stack.
   * This dynamic allocation and deallocation make stack memory **highly efficient**.

**🔹 Stack Memory Characteristics**

* **Thread-Specific** → Each thread has its **own stack** memory.
* **Fast Access** → Stack operations are faster than heap memory because of **LIFO ordering**.
* **Automatic Cleanup** → No Garbage Collection (GC) is needed; memory is freed when methods exit.
* **Limited Size** → Stack memory is smaller than heap memory and can cause **StackOverflowError** if overused.

**🔹 Stack Memory Lifecycle**

1. **A method is called** → A new stack frame is created.
2. **Local variables and execution details are stored** in the stack frame.
3. **Method completes execution** → Stack frame is removed.
4. **If the stack limit is exceeded** → A **StackOverflowError** occurs.

**🔹 What Happens When Stack Memory Overflows?**

* If too many methods are called **without returning**, the stack grows continuously.
* When the allocated stack size exceeds the JVM limit, a **StackOverflowError** occurs.
* This usually happens in cases of **infinite recursion** or excessive deep method nesting.