Chapter 9: Memory Management and Garbage Collection

Introduction

Efficient memory management is crucial in Java programming, especially for large-scale applications. Unlike languages like C or C++, Java automatically handles memory allocation and deallocation through its built-in **Garbage Collector (GC)**. This feature minimizes memory leaks and frees developers from the complexities of manual memory handling. This chapter explores Java's memory model, how objects are stored and removed, the working of garbage collection, and ways to optimize memory usage in Java applications.

9.1 Java Memory Model Overview

Java divides memory into several distinct areas, each serving a specific purpose in the program's execution. These memory regions are managed by the Java Virtual Machine (JVM).

9.1.1 Key Memory Areas

Memory Area	Description
Heap	Stores objects and class instances. This is where garbage collection operates.
Stack	Stores method calls and local variables. Each thread has its own stack.
Method Area	Contains class-level data, such as method definitions and static variables.
PC Register	Keeps track of the JVM instruction currently being executed by each thread.
Native Method Stack	Used for native (non-Java) method calls, typically written in C/C++.

9.2 Object Allocation in Java

In Java, objects are always created on the **heap** using the `new` keyword. For example:

```
javaCopy codeEmployee emp = new Employee();  // Object stored in heap
```

Key Points:

- Heap memory is shared among all threads.
- Once there are no references to an object, it becomes **eligible for garbage collection**.
- Variables inside methods (primitives or object references) are stored in the **stack**.

9.3 Understanding Garbage Collection (GC)

9.3.1 What is Garbage Collection?

Garbage collection is the **process of automatically identifying and freeing memory** used by objects that are no longer reachable in the application.

9.3.2 Why Use Garbage Collection?

- Prevents **memory leaks**.
- Reduces chances of **OutOfMemoryError**.
- Makes memory management easier and safer.

9.4 How Garbage Collection Works

9.4.1 Mark and Sweep Algorithm

1. **Mark Phase**: The GC identifies which objects are still reachable (i.e., still referenced).
2. **Sweep Phase**: Unreachable objects are removed, and memory is reclaimed.

9.4.2 Reachability Analysis

Java uses reachability from **GC Roots** like:

- Local variables in stack
- Static variables

- Active thread references

If an object is not reachable from any of these, it is **considered garbage**.

9.5 Types of Garbage Collectors in Java

9.5.1 Serial Garbage Collector

- Uses a single thread.
- Best suited for small applications.

```
bashCopy code-XX:+UseSerialGC
```

9.5.2 Parallel Garbage Collector (Throughput GC)

- Uses multiple threads for minor GC.
- Good for multi-threaded applications.

```
bashCopy code-XX:+UseParallelGC
```

9.5.3 CMS (Concurrent Mark-Sweep) Collector

- Minimizes pauses by doing GC in parallel with the application.
- Deprecated in newer versions of Java.

```
bashCopy code-XX:+UseConcMarkSweepGC
```

9.5.4 G1 (Garbage First) Collector

- Designed for large heaps.
- Balances pause time and throughput.
- Breaks heap into regions.

```
bashCopy code-XX:+UseG1GC
```

9.5.5 Z Garbage Collector (ZGC) and Shenandoah

- Low-latency collectors in Java 11+ and OpenJDK.
- Suitable for ultra-low pause time requirements.

9.6 JVM Heap Structure

Java divides the heap into **generations**:

9.6.1 Young Generation

- New objects are allocated here.
- Frequent minor GCs occur.
- Includes **Eden** and **Survivor** spaces.

9.6.2 Old (Tenured) Generation

- Stores long-lived objects.
- Less frequent but more time-consuming **major GCs**.

9.6.3 Permanent Generation / Metaspace

- Stores class metadata.
- Replaced by **Metaspace** in Java 8 onwards.

9.7 Finalization and `finalize()` Method

Java provides the `finalize()` method to allow cleanup actions before an object is removed.

```
javaCopy code@Override
protected void finalize() throws Throwable {
    System.out.println("Cleaning up before GC");
}
```

> **Note:** `finalize()` is **deprecated** since Java 9. Use `try-with-resources` or **clean-up hooks** instead.

9.8 Memory Leaks in Java

Even though Java has GC, memory leaks can still occur if references are unintentionally held.

Common Causes:

- Static variables holding object references
- Listeners not removed
- Caches or maps (e.g., `HashMap`) growing without bounds

Solution: Use **WeakReference**, **SoftReference**, or proper removal strategies.

9.9 Monitoring and Tuning Garbage Collection

9.9.1 JVM Tools

- `jconsole` – GUI-based monitoring tool
- `jvisualvm` – Profiling, GC activity, memory usage
- `jstat`, `jmap`, `jstack` – Command-line monitoring

9.9.2 JVM Options for GC Tuning

```
bashCopy code-verbose:gc                        # Prints GC details
-Xms512m -Xmx2048m              # Set heap size
-XX:+PrintGCDetails             # Show GC stats
-XX:+UseG1GC                    # Use G1 collector
```

9.10 Best Practices for Efficient Memory Management

- **Reuse objects** when possible.
- **Avoid memory leaks** by clearing unused references.
- Use **StringBuilder** instead of `String` for concatenations.
- Use **Object Pools** for expensive objects (e.g., DB connections).
- Profile applications for **heap usage** regularly.

Summary

In this chapter, we explored how Java handles memory through its **automatic garbage collection system**. Java's **heap-based object allocation**, coupled with a variety of garbage collectors (Serial, Parallel, G1, ZGC), makes memory management both powerful and efficient. We also covered memory areas, the object life cycle, GC algorithms, tools for monitoring, and best practices to avoid memory leaks. Understanding memory management helps Java developers write **robust**, **efficient**, and **scalable** applications.
