Java Memory Management - Detailed Guide

Java manages memory using a combination of **stack**, **heap**, and **metaspace**. The JVM automatically handles memory allocation and deallocation using a process called **Garbage Collection (GC)**. This document covers the various aspects of Java memory management in detail.

---

1. Stack Memory Allocation for Local Variables

In Java, each thread has its own **stack** which stores:

- Primitive data types (e.g., `int`, `char`, `float`)
- References to objects in the heap

Each time a method is invoked, a new **stack frame** is created for that method. The stack frame contains all the method's local variables and references. When the method call ends, the stack frame is removed.

Sample Code:

```
public class StackExample {
    public void methodA() {
        int localInt = 5; // Stored in stack
        String localString =
"Hello"; // Reference stored in stack, object in heap (string pool)
    }
}
```

---

2. Scope in Stack Memory

Scope defines the visibility and lifetime of variables. A variable exists in memory only for the duration of its scope.

- Local variables are valid only within the method/block they are defined in.
- Once the method exits, the stack frame is destroyed and the variables are no longer accessible.

```
public void showScope() {
    int x = 10; // x is accessible only inside showScope
    {
        int y = 20; // y is accessible only in this block
    }
    // y is out of scope here
}
```

---

3. Object Creation in Heap Memory

All **objects** and **instance variables** are created in the **heap**. The heap is shared across all threads.

Sample Code:

```
class Car {
    String model;
    Car(String model) {
        this.model = model;
    }
}

public class HeapExample {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla"); // Object created in heap, reference in
stack
    }
}
```

---

4. Memory Creation for String Literals vs String Objects

String Literal:

- Stored in the **String Pool**, which is part of heap memory.
- Reused if the same value already exists.

String Object:

- Created using `new` keyword, stored in **heap**, not reused unless explicitly interned.

Sample Code:

```
public class StringMemory {
    public static void main(String[] args) {
        String s1 = "Java";           // String pool
        String s2 = "Java";           // Reuses s1 from pool

        String s3 = new String("Java"); // New object in heap

        System.out.println(s1 == s2); // true
        System.out.println(s1 == s3); // false
    }
}
```

---

5. What is Garbage Collection?

Garbage Collection is the process of **automatically identifying and deallocating memory** used by unreachable objects in the heap.

- Helps avoid memory leaks
- Performed by JVM periodically
- Objects with no active references are considered garbage

```
public class GCDemo {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Object is garbage collected");
    }

    public static void main(String[] args) {
        GCDemo obj = new GCDemo();
        obj = null; // Eligible for GC
        System.gc(); // Suggest JVM to perform GC
    }
}
```

---

6. Young and Old Generation of Objects

Java Heap is divided into:

Young Generation:

- Newly created objects
- Includes **Eden Space** and **Survivor Spaces (S0, S1)**
- Frequent GC (minor GC)

Old (Tenured) Generation:

- Long-living objects
- GC happens less frequently (major GC)

Promotion:

- Objects that survive multiple minor GCs in Young Gen are promoted to Old Gen.

---

7. What is Metaspace?

Metaspace replaced **PermGen** in Java 8.

- Stores **class metadata**, such as class structure, methods, constant pool, etc.
- Resides in native memory (not part of heap)

- Grows automatically (bounded by system memory)

---

8. Mark and Sweep Algorithm

A classic GC algorithm used in Java:

Steps:

1. **Mark phase**: Traverse object graph and mark all reachable objects.
2. **Sweep phase**: Deallocate memory used by unmarked (unreachable) objects.

Disadvantage:

- Causes **stop-the-world** pauses.
- Inefficient in large heaps due to full scan.

---

9. Other Garbage Collector Algorithms

1. Serial GC

- Single-threaded
- Suitable for small applications
- JVM Option: `-XX:+UseSerialGC`

2. Parallel GC (Throughput GC)

- Multi-threaded collector
- Focuses on high throughput
- JVM Option: `-XX:+UseParallelGC`

3. Concurrent Mark Sweep (CMS)

- Concurrent GC with low pause time
- Deprecated in newer Java versions
- JVM Option: `-XX:+UseConcMarkSweepGC`

4. G1 (Garbage First) GC

- Default from Java 9 onward
- Divides heap into regions
- Performs concurrent marking and compaction
- JVM Option: `-XX:+UseG1GC`

5. ZGC and Shenandoah (Java 11+)

- Extremely low pause GCs
- Designed for large heaps

Conclusion

Java’s memory management abstracts the complexity of manual memory handling. With its stack-heap separation, generational GC, and dynamic class metadata management, Java provides a robust and scalable model for applications of any size.

Use appropriate JVM tuning parameters to optimize GC behavior based on your application's needs.