Java Memory Management - Complete Guide

Table of Contents

Memory Architecture Overview

Java memory is divided into **two main categories**:

```
JVM Memory
├── Heap Memory (Objects)
│   ├── Young Generation
│   │   ├── Eden Space
│   │   ├── Survivor 0 (S0)
│   │   └── Survivor 1 (S1)
│   └── Old Generation (Tenured)
│
└── Non-Heap Memory
    ├── Stack Memory (per thread)
    ├── Method Area/Metaspace
    ├── PC Registers
    └── Native Method Stacks
```

Key Principle

- **Heap**: Stores all objects and instance variables
- **Stack**: Stores method calls, local variables, and object references
- **Method Area**: Stores class-level information

Heap Memory (Object Storage)

Structure & Flow

```
New Object Creation Flow:
[New Object] → Eden Space → Survivor 0/1 → Old Generation
```

1. Young Generation (New Objects)

Eden Space

- **Purpose**: All new objects are created here
- **Size**: Largest portion of Young Generation
- **Lifecycle**: Objects stay until first GC cycle

Survivor Spaces (S0 & S1)

- **Purpose**: Hold objects that survived at least one GC
- **Mechanism**: Only one survivor space is active at a time
- **Rule**: Objects move between S0 ↔ S1 during minor GC

2. Old Generation (Tenured)

- **Purpose**: Long-lived objects that survived multiple GC cycles
- **Promotion**: Objects move here after surviving 8-15 GC cycles (configurable)
- **Size**: Larger than Young Generation

Memory Allocation Example

```
// These objects start in Eden Space
String name = new String("John");           // Object in Eden
List<String> list = new ArrayList<>();     // Object in Eden
Person person = new Person("Alice", 25);   // Object in Eden

// After multiple GC cycles, long-lived objects move to Old
Generation
```

Stack Memory (Method Execution)

Characteristics

- **Thread-specific**: Each thread has its own stack
- **LIFO Structure**: Last In, First Out
- **Automatic Management**: No manual memory management needed

Stack Frame Components

Each method call creates a stack frame containing:

1. **Local Variables**: Method parameters and local variables
2. **Operand Stack**: For intermediate calculations
3. **Reference to Method Area**: Points to method metadata

Stack Example

```
public class StackExample {
    public static void main(String[] args) { // Stack Frame 1
        int x = 10;   // Local variable in stack
        methodA();    // Creates new stack frame
    }

    public static void methodA() {   // Stack Frame 2
        int y = 20;                  // Local variable in stack
        Object obj = new Object();  // Reference in stack, object
in heap
        methodB();                  // Creates new stack frame
    } // Stack Frame 2 destroyed here

    public static void methodB() {   // Stack Frame 3
        int z = 30;              // Local variable in stack
    } // Stack Frame 3 destroyed here
} // Stack Frame 1 destroyed here
```

Stack vs Heap Reference:

```
public void example() {
    int primitive = 42;              // Stored directly in stack
    String reference = "Hello";      // Reference in stack, object
in heap
    List<String> list = new ArrayList<>();  // Reference in stack,
```

```
object in heap
}
```

🏛 Method Area & Metaspace

Java 8+ (Metaspace)

Replaced PermGen with **Metaspace** for better memory management:

What's Stored:

- **Class Metadata**: Class structure, method signatures
- **Runtime Constant Pool**: String literals, numeric constants
- **Static Variables**: Class-level variables
- **Method Bytecode**: Compiled method instructions

Key Improvements over PermGen:

- **Dynamic Sizing**: Grows automatically
- **Native Memory**: Uses system memory, not heap
- **Better GC**: Reduced OutOfMemoryError issues

Static vs Instance Memory

```
public class MemoryExample {
    // Stored in Method Area
    private static int staticVar = 100;
    private static final String CONSTANT = "Hello";

    // Stored in Heap (with object instance)
    private int instanceVar = 50;

    public static void staticMethod() {
        // Method definition stored in Method Area
        int localVar = 10;  // Stored in Stack when method executes
    }
}
```

🗑️ Garbage Collection

Types of Garbage Collection

Minor GC (Young Generation)

- **Trigger**: When Eden space fills up
- **Process**:
    1. Copy surviving objects to Survivor space
    2. Clear Eden space
    3. Swap active Survivor space
- **Performance**: Fast (few milliseconds)

Major GC (Old Generation)

- **Trigger**: When Old Generation fills up
- **Process**: Clean up Old Generation
- **Performance**: Slow (can cause application pauses)

Full GC

- **Scope**: Cleans entire heap + Method Area
- **Performance**: Slowest (significant application pause)

GC Process Visualization

```
Before Minor GC:
Eden: [Obj1][Obj2][Obj3][Obj4] (FULL)
S0:   [OldObj1]
S1:   [] (empty)

After Minor GC:
Eden: [] (cleared)
S0:   [] (cleared)
S1:   [OldObj1][Obj2][Obj4] (Obj1,Obj3 were garbage collected)
```

GC Performance Impact

```
// BAD - Creates many temporary objects
public String concatenateStrings(List<String> strings) {
    String result = "";
    for (String s : strings) {
        result += s;  // Creates new String object each time
    }
    return result;
}

// GOOD - Uses StringBuilder to reduce object creation
public String concatenateStrings(List<String> strings) {
    StringBuilder sb = new StringBuilder();
    for (String s : strings) {
        sb.append(s);  // Reuses internal buffer
    }
    return sb.toString();
}
```

Memory Pools

String Pool (String Interning)

- **Location**: Heap memory (Java 7+)
- **Purpose**: Reuse identical string literals
- **Benefit**: Memory optimization

```
// String Pool Example
String s1 = "Hello";           // Created in String Pool
String s2 = "Hello";           // Reuses same object from pool
String s3 = new String("Hello"); // Creates new object in heap

System.out.println(s1 == s2);  // true (same reference)
System.out.println(s1 == s3);  // false (different references)
System.out.println(s1.equals(s3)); // true (same content)
```

Integer Pool (-128 to 127)

```
Integer i1 = 100;  // From Integer pool
Integer i2 = 100;  // Same reference from pool
Integer i3 = 200;  // New object (outside pool range)
Integer i4 = 200;  // New object

System.out.println(i1 == i2);  // true
System.out.println(i3 == i4);  // false
```

Practical Examples

Memory Allocation Example

```
public class MemoryAllocationExample {
    private static int staticCounter = 0;  // Method Area
    private String instanceName;           // Heap (with object)

    public MemoryAllocationExample(String name) {
        this.instanceName = name;          // Reference in heap
        staticCounter++;                   // Updated in Method
Area
    }

    public void processData() {            // Method definition in
Method Area
        int localVar = 42;                 // Stack
        String temp = "Processing...";     // Reference in Stack,
object in String Pool
        List<String> data = new ArrayList<>(); // Reference in
Stack, object in Heap

        for (int i = 0; i < 1000; i++) {   // Loop variable in
Stack
            data.add("Item " + i);         // New objects created
in Heap
        }
    } // Local variables cleared from Stack when method ends
}
```

Memory Leak Prevention

```
// MEMORY LEAK - Static collection grows indefinitely
public class BadExample {
    private static List<String> cache = new ArrayList<>();

    public void addToCache(String data) {
        cache.add(data);  // Objects never removed, potential
memory leak
    }
}

// GOOD PRACTICE - Bounded cache with cleanup
public class GoodExample {
    private static final int MAX_SIZE = 1000;
    private static List<String> cache = new ArrayList<>();

    public void addToCache(String data) {
        if (cache.size() >= MAX_SIZE) {
            cache.remove(0);  // Remove oldest entry
        }
        cache.add(data);
    }
}
```

Interview Questions & Tips

Essential Topics to Master

1. Memory Areas

- Explain the difference between Heap and Stack
- Describe Young Generation vs Old Generation
- What is Metaspace and how it differs from PermGen

2. Garbage Collection

- When does Minor GC vs Major GC occur?
- How does GC identify objects for cleanup?
- What causes OutOfMemoryError?

3. Code Analysis

```
// Interview Question: Where are these stored?
public class InterviewExample {
    private static final String CONSTANT = "Hello";  // Method Area
(Runtime Constant Pool)
    private static int count = 0;                     // Method
Area
    private String name;                              // Heap (with
object instance)

    public void method() {
        int local = 10;                               // Stack
        String temp = new String("World");            // Reference:
Stack, Object: Heap
        this.name = "Test";                           // Updates
heap object
    }
}
```

Memory Tuning JVM Parameters

```
# Heap Size
-Xms512m          # Initial heap size
-Xmx2g            # Maximum heap size

# Young Generation
-XX:NewRatio=3    # Old:Young ratio (3:1)
-XX:NewSize=256m  # Initial young generation size

# Garbage Collection
-XX:+UseG1GC                    # Use G1 garbage collector
-XX:MaxGCPauseMillis=200        # Target max GC pause time
-XX:+HeapDumpOnOutOfMemoryError # Generate heap dump on OOM

# Monitoring
-XX:+PrintGC                    # Print GC information
-XX:+PrintGCDetails            # Detailed GC information
```

Common Interview Answers

Q: What happens when heap memory is full? A: JVM triggers garbage collection. If GC cannot free enough memory, it throws `OutOfMemoryError`.

Q: Can you force garbage collection? A: You can suggest GC using `System.gc()`, but JVM decides when to actually run it.

Q: What's the difference between stack overflow and heap overflow? A:

- **StackOverflowError**: Too many method calls (infinite recursion)
- **OutOfMemoryError**: Heap space exhausted (too many objects)

Best Practices

1. **Avoid Memory Leaks**
    - Close resources (files, connections)
    - Remove listeners and callbacks
    - Clear collections when done
2. **Optimize Object Creation**
    - Reuse objects when possible
    - Use object pools for expensive objects
    - Prefer StringBuilder over String concatenation
3. **Monitor Memory Usage**
    - Use profiling tools (JProfiler, VisualVM)
    - Monitor GC logs
    - Set appropriate heap sizes
4. **String Optimization**
    - Use String literals for constants
    - Use StringBuilder for dynamic strings
    - Be aware of String pool behavior

Summary

Java memory management is crucial for writing efficient applications. Understanding the distinction between heap and stack, garbage collection cycles, and memory optimization techniques will help you write better code and ace technical interviews.

Key Takeaways:

- Objects live in Heap, method execution happens in Stack
- GC automatically manages memory but understanding it helps optimization
- Metaspace replaced PermGen for better memory management
- Proper coding practices prevent memory leaks and improve performance