Java Memory Management

Java Memory Management

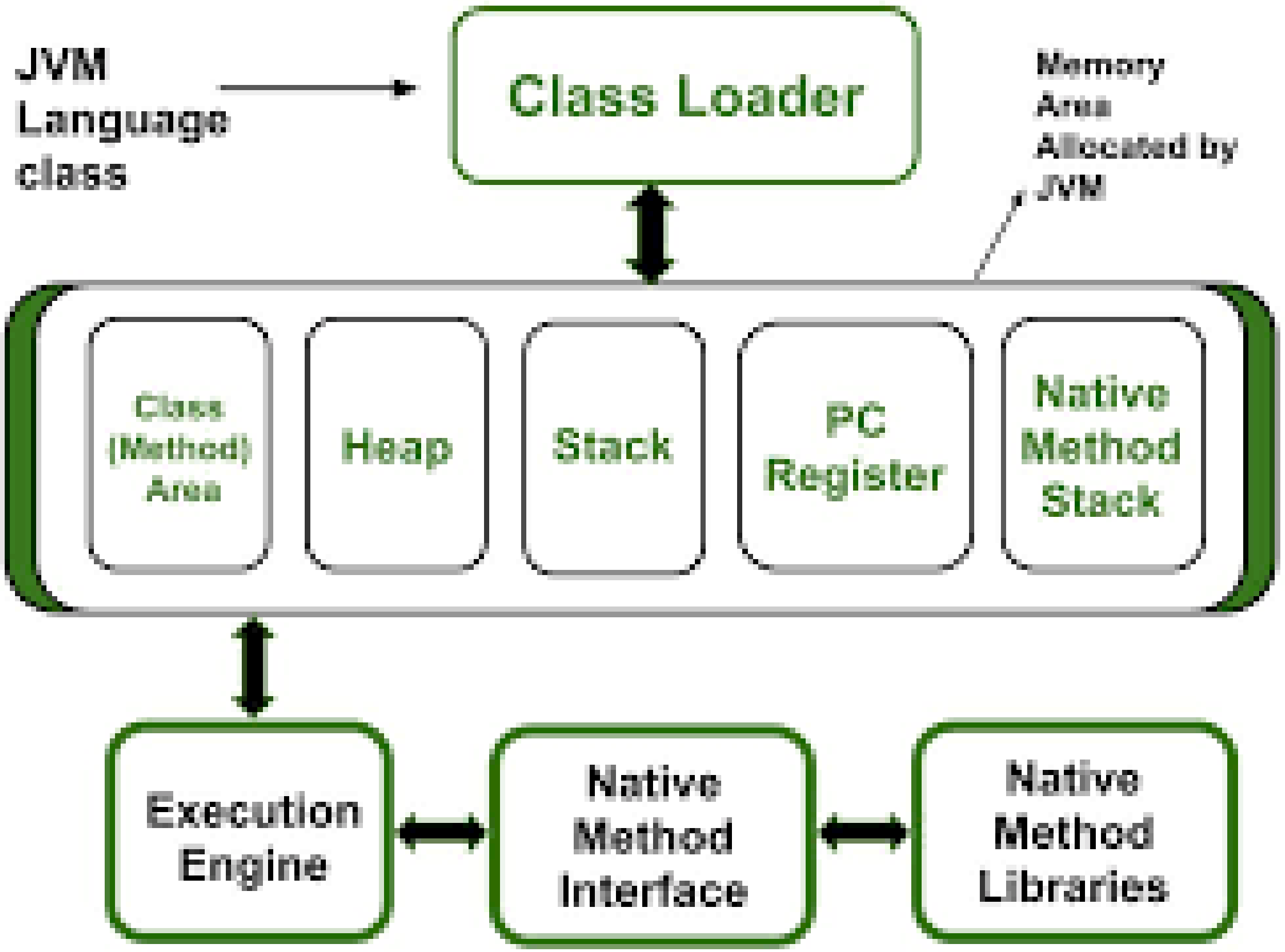
- Java uses an automatic memory management system that helps developers avoid manual allocation and deallocation of memory.
- The JVM (Java Virtual Machine) handles all of this behind the scenes.

Java Memory Areas (JVM Memory Structure)

- JVM is an abstract machine that runs Java bytecode.
- It acts as an interface between Java program and underlying OS.
- Responsibilities of JVM:
  - Loads class files
  - Manages memory
  - Executes bytecode
  - Handles garbage collection

JVM Architecture

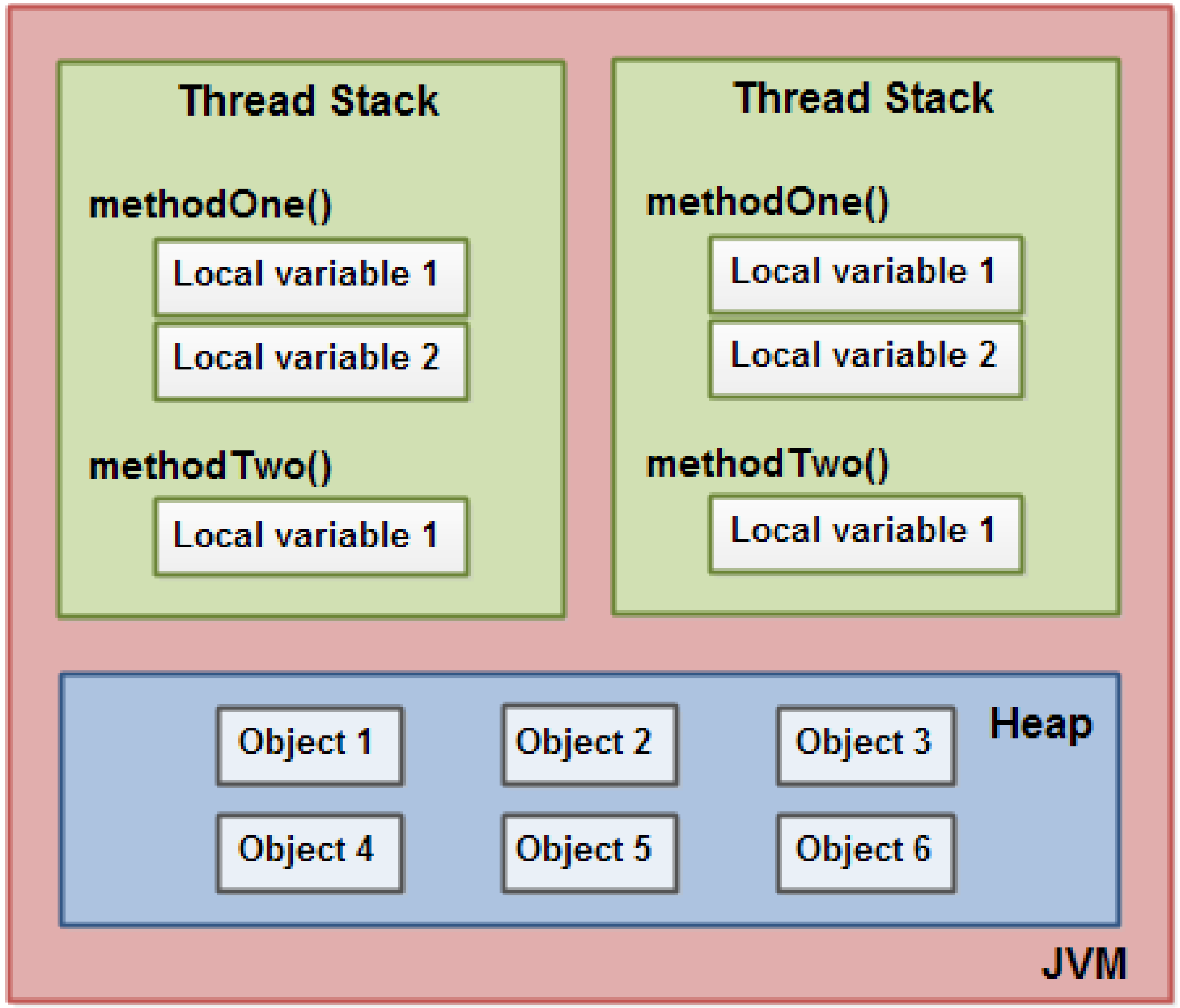
- Main components of JVM: -
- Class Loader Subsystem
- Runtime Data Areas
- Execution Engine
- Native Interface (JNI)



Runtime Data Areas (Memory Areas in JVM)

- Heap Memory
- Stack Memory
- Method Area (Metaspace)
- PC Register
- Native Method Stack

Runtime Data Areas (Memory Areas in JVM)



Heap Memory

Used for:

- Objects
- Instance variables
- Arrays

Characteristics:

- Managed by Garbage Collector (GC)
- Largest memory area
- Shared across all threads

Heap Memory - Example

```java
class Student {
    int marks;              // Instance variable → Heap
    int[] scores;           // Array reference → Heap
}

public class HeapStorageExample {

    public static void main(String[] args) {

        Student s1 = new Student();    // Object → Heap
        s1.marks = 80;

        s1.scores = new int[3];        // Array → Heap
        s1.scores[0] = 70;
        s1.scores[1] = 80;
        s1.scores[2] = 90;

        System.out.println("Marks: " + s1.marks);
        System.out.println("Scores: " + s1.scores[0] + ", "
                                   + s1.scores[1] + ", "
                                   + s1.scores[2]);

    }
}
```

Heap Memory

Heap is further divided into:

Young Generation

- Eden Space
- Survivor Spaces (S0, S1)

Old Generation

Young Generation vs Old Generation (Java Heap Memory)

Java’s Heap is divided into two main areas to optimize object allocation and garbage collection:

Young Generation (Young Gen)

Old Generation (Old Gen)

Why Java Uses Generations

If Java checked **all objects every time**, GC would be **very slow**.

Instead:

- New objects → checked frequently
- Old objects → checked less often

This makes garbage collection **faster and efficient**.

Young Generation (Young Gen)

This is where new objects are created.

Sub-areas

- Eden Space → All new objects are allocated here.
- Survivor Space S0 & S1 → Objects that survive one or more GC cycles are moved here.

What happens when Eden is full?

- Minor GC happens

Minor GC (Only Young Generation)

Fast

Happens frequently

Cleans up short-lived objects

Behavior

- Most objects die young (e.g., temporary variables, short-lived objects).
- Those that survive several Minor GCs are promoted to Old Generation.

Key Characteristics

- Small memory area
- Very fast cleanup
- Optimized for short-lived objects

Survivor Spaces (S0 & S1)

Two survivor spaces are used alternately

Objects that survive multiple Minor GCs grow older

Old Generation

Purpose

- Stores long-lived objects.

Examples:

- Cached objects
- Large collections
- Objects that survive multiple Minor GCs

Major GC (Old Generation)

Major GC (or Full GC)

Slower than Minor GC

Less frequent

Cleans the Old Gen and sometimes the entire heap

Key Characteristics

- Larger memory area
- Contains stable objects
- Major GC is costlier and may cause noticeable pauses

Lifecycle Summary

Object created → Eden

Survives Minor GC → Survivor (S0/S1)

Survives multiple Minor GCs → Old Generation

GC Types Summary

GC Type	Area Cleaned	Speed	Frequency
Minor GC	Young Gen	Fast	Very Often
Major GC	Old Gen	Slow	Rare
Full GC	Entire Heap	Very Slow	Very Rare

Stack

- Stores method calls, local variables
- Each thread has its own stack
- Faster than heap

Stack - Example

```java
public class StackExample {

    public static void main(String[] args) {
        int a = 10;          // stored in Stack
        int b = 20;          // stored in Stack

        add(a, b);           // method call → new stack frame
    }

    static void add(int x, int y) {
        int sum = x + y;     // stored in Stack
        System.out.println("Sum = " + sum);
    }
}
```

Method Area (Metaspace in Java 8+)

- Stores:
- Class metadata
- Static variables
- Method bytecode

Method Area - Example

```java
class Student {
    static String schoolName = "ABC Public School"; // Method Area
    int marks;   // Heap (instance variable)
    static void displaySchool() {   // Method Area
        System.out.println("School: " + schoolName);
    }
    void displayMarks() {           // Method Area (method code)
        System.out.println("Marks: " + marks);
    }
}

public class MethodAreaExample {
    public static void main(String[] args) {
        Student.displaySchool();    // No object needed
        Student s1 = new Student(); // Object → Heap
        s1.marks = 85;
        s1.displayMarks();
    }
}
```

PC Register

Stores current instruction of a thread.

It keeps track of the address of the current (or next) bytecode instruction that a thread should execute.

Why does JVM need a PC Register?

Because Java runs on a multithreaded environment, and each thread executes its own stream of bytecode instructions.

Therefore:

Every thread has its own PC Register

(so that threads can resume execution exactly where they left off)

This is called thread-local memory.

Program: Memory Allocation Demo

```java
class MemoryDemo {

    static int staticVar = 100; // Method Area
    int instanceVar = 200;      // Heap
    void show() {
        int localVar = 300;     // Stack
        System.out.println(localVar);
    }
    public static void main(String[] args) {
        MemoryDemo obj = new MemoryDemo();
        obj.show();
    }
}
```

Memory Management Explained - Video