JavaXcelerate: Java Performance Optimization Workshop

BY M.H. SHOIAB
9840135749

Day 1

Day 1

Overview of Performance Optimization
o Importance of performance optimization
o Key performance metrics:Latency,throughput,responsetime,CPU,memory usage

Understanding the JavaVirtual Machine(JVM)
o JVM architecture and execution model
o How the JVM manages memory: Heap,Stack,GarbageCollection(GC)
o JVM options and tuning flags for performance
Performance Bottlenecks
o Common performance bottlenecks in Java applications
o Identifying bottlenecks: CPU-bound vs. IO-bound applications

AGILE

Overview of Performance Optimization

Performance optimization is crucial in Java applications to ensure smooth and efficient functionality.
Optimizing an application can improve the user experience, reduce resource consumption, and increase the scalability and reliability of the software.
Java, being widely used for high scale applications, benefits greatly from proper tuning and optimization.

Performance tuning in Java involves optimizing code and system configurations to improve the efficiency, speed, and resource usage of applications.
It is critical for ensuring high performance, scalability, and reliability in both Core Java applications (standalone programs) and Enterprise Java applications (web-based, distributed systems).
The approach varies depending on the application's complexity, architecture, and use case.

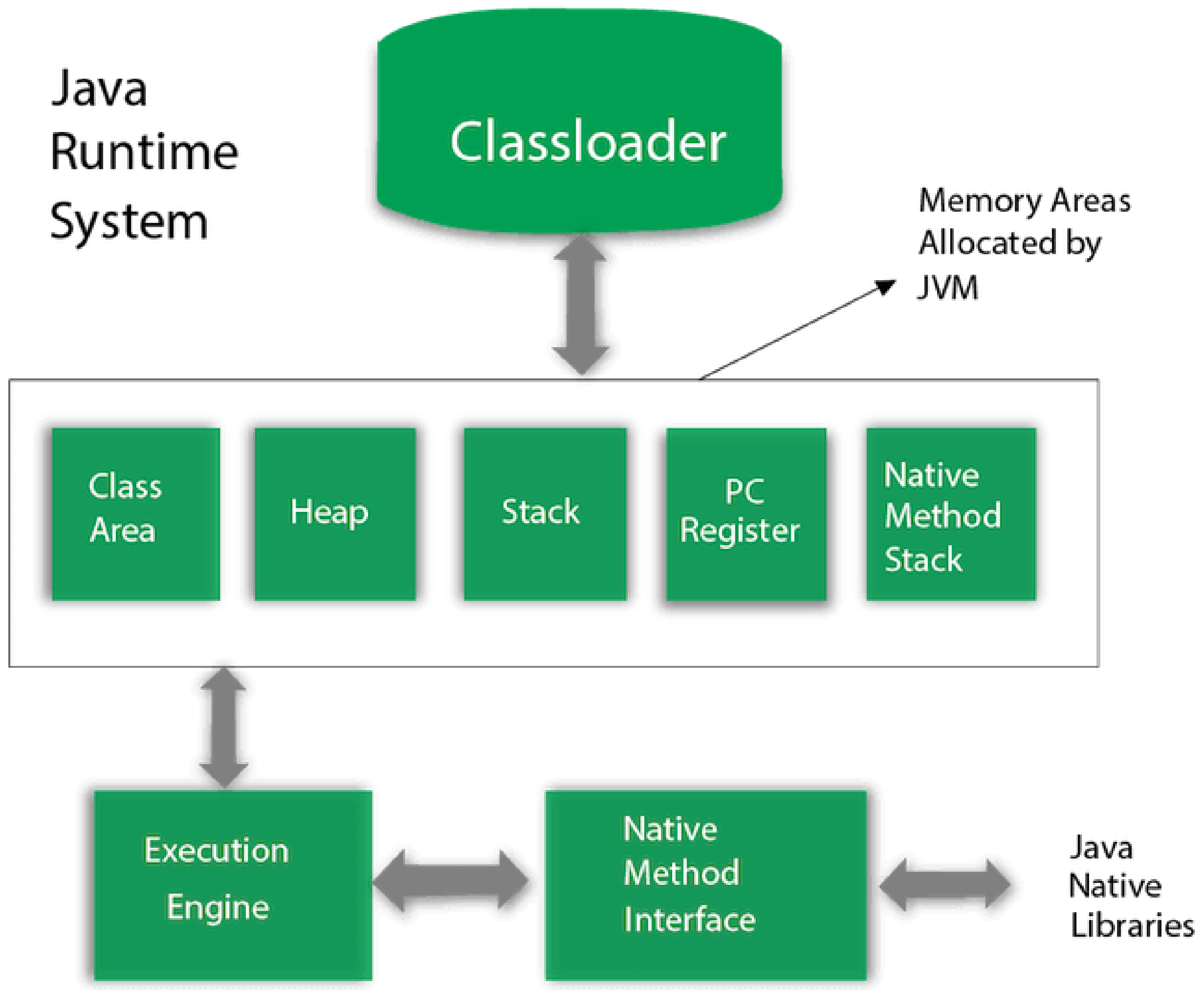
Overview of Performance Optimization

Key Performance Metrics
Understanding the key performance metrics is essential for evaluating how well your application is performing:

1. Latency: The time taken to process a single request. Lower latency is generally better, as it means requests are handled faster.
2. Throughput: The number of tasks or requests processed per unit of time. Higher throughput is typically a sign of a well optimized system.
3. Response Time: The total time taken from when a request is made until the response is received by the user. Response time is closely related to latency but includes additional factors.
4. CPU Usage: The percentage of CPU resources used by the application. High CPU usage can indicate a CPUbound application, while low CPU usage may indicate an IObound application.
5. Memory Usage: The amount of memory used by the application. Excessive memory usage can lead to memory leaks and increased garbage collection activity.
Initialization time
Managing Cache
Managing Garbage Collection
Boot up sequence.

Understanding the Java Virtual Machine (JVM)

The **Java Virtual Machine (JVM)** is the runtime engine that executes Java bytecode, providing platform independence ("Write Once, Run Anywhere"). Below is a breakdown of its architecture, memory management, and performance tuning.



Class Loader

Loads .class files into memory.

Three types:

Bootstrap ClassLoader (loads core Java classes, e.g., java.lang.*).

Extension ClassLoader (loads jre/lib/ext classes).

Application ClassLoader (loads user-defined classes).

Runtime Data Areas (Memory Management)

Heap (Object storage, shared across threads).

Method Area (Stores class metadata, static variables, constants).

JVM Stacks (Per-thread stack for method calls, local variables).

PC Register (Tracks thread execution point).

Native Method Stack (For native code, e.g., JNI calls).

Execution Engine

Interpreter: Executes bytecode line-by-line (slow).

JIT Compiler (Just-In-Time): Converts frequently used bytecode to native machine code (optimizes performance).

Garbage Collector (GC): Automatically reclaims unused memory.

JVM Manages Memory

A. Heap Memory (Dynamic Allocation for Objects)

Shared across all threads.

- Divided into generations for efficient GC:
    - Young Generation (Eden + Survivor Spaces)
        - New objects are allocated here.
        - Minor GC cleans short-lived objects.
    - Old Generation (Tenured Space)
        - Long-lived objects are promoted here.
        - Major GC (Full GC) runs here (slower).
    - Metaspace (Replaced PermGen in Java 8+)
        - Stores class metadata, static variables.

Core Java applications are typically standalone programs or tools. The performance bottlenecks in such applications are often related to memory management, algorithms, and threading.

Key Areas for Tuning in Core Java

1. Garbage Collection (GC):
   - Optimize JVM GC settings using flags like -XX:+UseG1GC or -Xms/-Xmx.
   - Analyze GC logs to minimize pauses.

2. Data Structures and Algorithms:
   - Use the right data structures (ArrayList, HashMap, etc.) for the task.
   - Optimize loops and recursion.

3. String Management:
   - Use StringBuilder for concatenations instead of String.
   - Leverage Java 9+ Compact Strings.

4. Multithreading:
   - Use thread pools (ExecutorService) instead of manually creating threads.
   - Reduce thread contention using synchronized blocks or java.util.concurrent locks.

5. File I/O:
   - Use buffered streams for efficient file operations.
   - Leverage NIO or NIO.2 for non-blocking I/O.

Performance Tuning in Enterprise Java - Enterprise Java applications are complex, distributed systems often built with frameworks like Spring, Hibernate, or JEE technologies. Tuning involves optimizing both the application code and the infrastructure it runs on.

Key Areas for Tuning in Enterprise Java

1. Database Optimization (JPA/Hibernate):
   - Use lazy loading to fetch only necessary data.
   - Optimize queries with indexes and projections.
   - Use batching for inserts/updates.

2. Caching:
   - Implement caching strategies using tools like Ehcache, Redis, or Spring Cache.
   - Cache frequently accessed data to reduce database calls.

3. Web Application Performance:
   - Use connection pooling for efficient database access.
   - Compress HTTP responses using GZIP.
   - Optimize static resource delivery with CDNs.

4. Garbage Collection and JVM Tuning:
   - Monitor and tune JVM parameters (e.g., -XX:+UseParallelGC, -Xmx for heap size).
   - Enable detailed logging with -Xlog:gc.

5. Asynchronous Processing:
   - Use message brokers like RabbitMQ or Kafka for asynchronous tasks.
   - Leverage non-blocking APIs (e.g., CompletableFuture, WebFlux).

6. Microservices and Scalability:
   - Ensure lightweight service communication using gRPC or REST.
   - Optimize containerized environments using tools like Kubernetes and Docker.

Improving the performance of a Spring Boot application by:
- Caching expensive database queries.
- Reducing the payload size in REST APIs with DTOs.
- Monitoring with APM tools like New Relic or Dynatrace.

General Tools for Performance Tuning
1. Profilers:
   - JProfiler, MAT, VisualVM.
2. Garbage Collection Analyzers:
   - GCViewer, GCEasy.
3. Heap Dump Analyzers:
   - Eclipse MAT (Memory Analyzer Tool).
4. Load Testing Tools:
   - Apache JMeter.

Key Performance Metrics
1. Throughput: Number of requests/operations per second.
2. Latency: Time taken to process a single request.
3. Memory Usage: Heap/stack utilization.
4. CPU Usage: CPU efficiency during processing.

Core Java Tuning: Focuses on efficient use of algorithms, threading, and memory.
Enterprise Java Tuning: Involves database optimization, caching, GC tuning, and scaling distributed systems.
Tools and Monitoring: Use profilers, heap analyzers, and APM tools to diagnose bottlenecks.
Performance tuning is an iterative process that requires profiling, diagnosing bottlenecks, and applying targeted optimizations.

Exercise: Measure Performance Metrics in a Sample Application

```
package performancepack;
public class Ex4 {
public static void main(String[] args) {
        // Measure precise time in nanoseconds
        long startTimeNano = System.nanoTime();
        System.out.println("Start time in nanoseconds (from nanoTime): " +
        startTimeNano);
        try {Thread.sleep(1);}catch(Exception e) {}
        long endTimeNano=System.nanoTime();
        System.out.println("End time in nanoseconds (from nanoTime): " +
        endTimeNano);
        System.out.println("Time Difference is..:"+(endTimeNano-startTimeNano));
        // Memory usage
        long freeMemory = Runtime.getRuntime().freeMemory();
        long totalMemory = Runtime.getRuntime().totalMemory();
        long usedMemory = totalMemory - freeMemory;


        System.out.println("Start Free Memory (bytes): " + freeMemory);
        System.out.println("Total Memory (bytes): " + totalMemory);
        System.out.println("Memory Usage Before (bytes): " + usedMemory);
}
}
```

In Java's JVM options, XX stands for non-standard or advanced options that are specific to the HotSpot JVM implementation. These options are typically used for fine-tuning JVM performance, debugging, or experimenting with internal JVM behaviors. They are not guaranteed to be available across all JVM implementations or versions.

Types of JVM Options

1. Standard Options:
   - Begin with a single - (e.g., -classpath, -Dproperty=value).
   - These are guaranteed to work across all JVM implementations.

2. Non-Standard Options:
   - Begin with -X (e.g., -Xms, -Xmx for heap sizes, -Xss for stack size).
   - Commonly supported but not part of the Java SE specification.

3. Advanced Options:
   - Begin with -XX: (e.g., -XX:+UseG1GC).
   - Offer advanced controls over JVM internals and are JVM-specific.