JavaXcelerate: Java Performance Optimization Workshop
Dates:

JavaXcelerate: Java Performance Optimization Workshop

Welcome to JavaXcelerate, an intensive, hands-on workshop designed to help you unlock the full potential of your Java applications by mastering the art of performance optimization. In today's competitive tech landscape, high performance is critical to delivering responsive, scalable, and efficient software solutions. JavaXcelerate is tailored for developers and engineers who want to fine-tune their Java applications, ensuring they run at peak efficiency.

Why JavaXcelerate?

Java is a powerful, versatile language, but performance bottlenecks can easily emerge in applications that handle large-scale data processing, complex algorithms, or high concurrency. JavaXcelerate goes beyond basic Java programming to explore advanced performance tuning techniques, providing practical insights into how to identify, troubleshoot, and optimize performance issues. From memory management to threading and garbage collection, this workshop equips you with the skills to build faster, more efficient applications.

What You Will Learn:

1. Java Memory Management & Garbage Collection
Gain a deep understanding of how Java manages memory and how to optimize it:
- Java heap structure and memory allocation
- How garbage collectors work: G1, Parallel, and ZGC
- Tuning garbage collection to reduce latency and improve throughput
- Analyzing memory leaks and heap dumps

2. Profiling and Benchmarking Java Applications
Learn how to identify performance bottlenecks using modern profiling tools:
- Using tools like JProfiler, VisualVM, and YourKit to profile CPU, memory, and thread activity
- Best practices for running performance benchmarks using JMH (Java Microbenchmark Harness)
- Techniques to analyze profiling data and improve application performance

3. Optimizing Multithreading and Concurrency
Understand how to efficiently manage threads in high-concurrency environments:
- Best practices for managing threads and thread pools
- Reducing contention and deadlocks through effective synchronization
- Leveraging Java's `concurrent` package to write efficient, thread-safe code
- Fine-tuning `ForkJoinPool` for parallelism

4. JVM Tuning for High-Performance Applications
Dive into the intricacies of tuning the JVM to squeeze the best performance out of your applications:

- JVM flags and settings that impact performance
- Heap size, metaspace tuning, and thread stack management
- Monitoring JVM performance using tools like `jstat`, `jstack`, and `jmap`

5. Database Optimization and I/O Performance

Enhance your application's interaction with databases and I/O systems:
- Best practices for optimizing database connections and query execution
- Caching strategies (e.g., Redis, EHCache) to reduce database load
- Reducing I/O bottlenecks with efficient file handling, serialization, and deserialization

6. Code Optimization Techniques

Learn practical techniques to improve the efficiency of your Java code:
- Writing efficient algorithms and data structures
- Reducing object creation and memory overhead
- Leveraging Java's `Stream` API and lambdas for optimal performance
- Analyzing and improving code execution paths

Why Choose JavaXcelerate?

- Industry-Focused Curriculum: Our workshop covers the latest Java performance optimization techniques used by top tech companies.
- Expert Instructor: Learn from a seasoned Java professional who have hands-on experience in building and tuning high-performance systems.
- Hands-On Projects: Put theory into practice with real-world performance tuning exercises and case studies.
- Comprehensive Tools: Gain mastery over the essential tools and utilities that every Java developer should know to monitor and improve application performance.
- Post-Workshop Support: After completing the workshop, access additional resources, Q&A sessions, and follow-up materials to continue your optimization journey.

Who Should Join?

- Java Developers looking to enhance the performance of their applications.
- Software Engineers working on high-concurrency or data-intensive systems.
- Performance Engineers who want to deepen their knowledge of JVM tuning and code optimization.
- Tech Leads responsible for ensuring that their teams' applications meet performance goals.
- System Architects designing scalable, high-performance Java systems.

Elevate Your Java Skills

In a world where software performance directly impacts user experience and business success, knowing how to optimize your Java applications is crucial. JavaXcelerate equips you with the knowledge and practical experience to boost the performance of your applications and ensure they are robust, efficient, and scalable.

Join the JavaXcelerate: Java Performance Optimization Workshop and take your

Java applications to the next level. Maximize efficiency, reduce bottlenecks, and become an expert in Java performance tuning!

Day 1

Introduction To

- **Overview of Performance Optimization**
  - Importance of performance optimization
  - Key performance metrics:Latency,throughput,responsetime,CPU,memory usage
- **Understanding the JavaVirtual Machine(JVM)**
  - JVM architectureand execution model
  - How the JVM manages memory: Heap,Stack,GarbageCollection(GC)
  - JVM options and tuning flags for performance
- **Performance Bottlenecks**
  - Common performance bottlenecks in Java applications
  - Identifying bottlenecks: CPU-bound vs. IO-bound applications

Day 2

- **Key Areas for Tuning in Java**
  - GC
  - Data Structures and Algorithms
  - String Management
  - Multithreading
  - File I/O
- **Key Areas For Tuning in Enterprise Java**
  - Database Optimization
  - Caching
  - Web App
  - GC and JVM Tuning
  - Asynchronous Processing
  - MicroServices and Scalability
- **General Tools for Performance Tuning**
  - JProfilers
  - GCViewer
  - Apache JMeter

Exercise 1: Measure Performance Metrics in a Sample Application
Exercise 2: Setting JVM Options for Performance Improvement
Exercise 3: HelloWorld Exercise with Memory – Heap and Stack
Exercise 4: A program that simulates a memory leak by creating objects and retaining references to them.
Exercise 5: To trigger a heap dump programmatically from within a Java application.
Exercise 6: Other Dumps which are needed for us to evaluate our code

for performance tuning.

Exercise 7: JVM Options with Java 8 and later version of Java 8 to Java 23

Day 3:

- **Analysing HPROF files though MAT or VisualVM Tools.**
- **Analysing CollectioLnevelMemoryLeak**
- **Possible solutions using Patterns**
- **Java API's Solutions for Performance**

  String
  Optional
  Elimination of If-Else-If Ladder for Performance
  DateTime Class
  Atomic Classes
  Atomic Reference
  Multithreading
  Concurrency Framework

Day 4:

Collections Package
Introduction to Garbage Collection in Java
Garbage Collection Architecture
Garbage Collection Algorithms
Garbage Collection Analysis and Monitoring
Performance Tuning for GC
Coding Exercises

Exercise 1: Monitoring GC with Verbose Logs
GARBAGE COLLECTION MONITORING
Optimize GC for Performance
Performance Tuning for GC
Introduction to Telemetry
Implementing Telemetry in Java
Instrumenting Your Code for Telemetry
synchronization constructs used to coordinate the execution of multiple threads.
JDBC Performance Improvement
Reentrant Read Write Lock
NIO
HTTP Tuning

Books for Pre Reading

"Java Performance: The Definitive Guide" by Scott Oaks
https://archive.org/details/javaperformance_thedefinitiveguide/page/n71/mode/2up

"Java Performance Tuning" by Jack Shirazi
https://archive.org/details/javaperformancet0000shir_p0d9/page/n451/mode/2up

Pre Test - Performance Optimization and JVM

1. Which of the following is NOT a key performance metric for Java applications?
   A. Latency
   B. Throughput
   C. Response Time
   D. Code Style

2. What is the primary purpose of the Garbage Collection (GC) process in the JVM?
   A. To manage CPU usage
   B. To reclaim memory used by unreachable objects
   C. To optimize code execution speed
   D. To log performance metrics

3. Which part of the JVM memory stores method call data and local variables?
   A. Heap
   B. Stack
   C. Permanent Generation
   D. Code Cache

4. Which of the following flags is commonly used to tune the garbage collector in Java?
   A. Xms
   B. Xmx
   C. XX:+UseG1GC
   D. Dfile.encoding=UTF8

5. What is telemetry in the context of software applications?
   A. A method of measuring software bugs
   B. Automated data collection and transmission from software
   C. An approach to database optimization
   D. None of the above

6. Which of the following is NOT a type of telemetry data?
   A. Logs

B. Metrics
C. Traces
D. Source Code

7. Which tool is commonly used for telemetry in Java applications?
A. Open Telemetry
B. Hibernate
C. JUnit
D. Jenkins

8. In distributed tracing, what is a 'span'?
A. A complete trace of all system interactions
B. A single unit of work or operation
C. A thread lock mechanism
D. A memory management technique

9. Which of the following is NOT a distributed tracing tool?
A. Zipkin
B. Jaeger
C. Visual VM
D. OpenTelemetry

10. Which concept in distributed tracing helps in tracking requests as they propagate through multiple services?
A. Call Stack
B. Context Propagation
C. Thread Pool
D. Garbage Collection

11. Which type of profiler is used to analyze memory usage in Java applications?
A. CPU Profiler
B. Thread Profiler
C. Memory Profiler
D. Lock Profiler

12. What is the main purpose of a flame graph in performance analysis?
A. To visualize CPU usage hotspots
B. To display memory leaks
C. To track user authentication logs
D. To analyze database queries

13. Which profiling tool is specifically built into the JVM for advanced performance diagnostics?
A. JProfiler
B. YourKit
C. Java Mission Control (JMC)
D. Visual Studio Code

14. What is the significance of tuning the Xmx JVM option?
A. It sets the initial size of the heap memory

B. It sets the maximum heap memory size
C. It enables garbage collection
D. It controls the thread count

15. Which Java Virtual Machine (JVM) garbage collector is optimized for short pause times?
A. Serial GC
B. Parallel GC
C. CMS (Concurrent MarkSweep) GC
D. G1 (Garbage First) GC

1. Answer: D. Code Style

2. Answer: B. To reclaim memory used by unreachable objects

3. Answer: B. Stack

4. Answer: C. XX:+UseG1GC

5. Answer: B. Automated data collection and transmission from software

6. Answer: D. Source Code

7. Answer: A. Open Telemetry

8. Answer: B. A single unit of work or operation

9. Answer: C. Visual VM

10. Answer: B. Context Propagation

11. Answer: C. Memory Profiler

12. Answer: A. To visualize CPU usage hotspots

13. Answer: C. Java Mission Control (JMC)

14. Answer: B. It sets the maximum heap memory size

15. Answer: D. G1 (Garbage First) GC

POST Test - Performance Optimization & JVM

1. Which JVM option can be used to set the initial size of the heap memory?
A. Xms
B. Xmx
C. XX:+UseG1GC
D. XX:+PrintGCDetails

2. Which type of bottleneck is caused when a process spends most of its time waiting for data input/output operations?
   A. CPUbound
   B. Memorybound
   C. I/Obound
   D. Networkbound

3. In the JVM, which memory area is used for storing object instances?
   A. Stack
   B. Heap
   C. Native Method Stack
   D. Program Counter

4. Which of the following tools is commonly used to capture and visualize telemetry data in Java applications?
   A. Maven
   B. Micrometer
   C. Jenkins
   D. Hibernate

5. Which of the following telemetry data types provides a highlevel overview of application behavior over time?
   A. Logs
   B. Traces
   C. Metrics
   D. Debug Statements

6. Which component in OpenTelemetry is responsible for collecting and exporting telemetry data?
   A. Collector
   B. Tracer
   C. Instrumentation
   D. Metrics Analyzer

7. What is the role of 'context propagation' in distributed tracing?
   A. It logs the transaction status
   B. It identifies code bottlenecks
   C. It maintains trace information across multiple services
   D. It profiles the CPU usage

8. Which tool is NOT typically used for distributed tracing?
   A. Jaeger
   B. Zipkin
   C. YourKit
   D. OpenTelemetry

9. What does a 'span' represent in distributed tracing?
   A. An individual operation within a trace
   B. A complete code execution path
   C. A memory allocation instance

D. A CPU profiling snapshot

10. Which type of profiler would you use to identify code that causes high memory usage?
A. CPU Profiler
B. Memory Profiler
C. Thread Profiler
D. Network Profiler

11. Which Java profiling tool can help visualize thread contention and analyze lock issues?
A. JConsole
B. JProfiler
C. Java Mission Control (JMC)
D. Visual VM

12. What does a flame graph help to identify in Java performance analysis?
A. Memory leaks
B. Slow database queries
C. Code paths consuming the most CPU
D. Unhandled exceptions

13. Which JVM tuning parameter should you adjust to manage the maximum heap size?
A. Xms
B. Xmx
C. XX:+UseConcMarkSweepGC
D. XX:MaxPermSize

14. Which of the following indicates that a Java application is experiencing CPUbound performance issues?
A. High memory consumption
B. Long response times
C. High CPU utilization
D. Frequent garbage collection

15. During memory profiling, what does the term 'memory leak' refer to?
A. Inefficient CPU usage
B. Unreleased memory causing a gradual increase in usage
C. Excessive database calls
D. Slow I/O operations

1. Answer: A. Xms

2. Answer: C. I/Obound

3. Answer: B. Heap

4. Answer: B. Micrometer

5. Answer: C. Metrics

6. Answer: A. Collector

7. Answer: C. It maintains trace information across multiple services

8. Answer: C. YourKit

9. Answer: A. An individual operation within a trace

10. Answer: B. Memory Profiler

11. Answer: C. Java Mission Control (JMC)

12. Answer: C. Code paths consuming the most CPU

13. Answer: B. Xmx

14. Answer: C. High CPU utilization

15. Answer: B. Unreleased memory causing a gradual increase in usage

Brief Notes

1. Performance Optimization Overview

- Why it matters: Faster apps, lower resource costs, better scalability.
- Key Metrics:
  - Latency: Time taken for a single operation.
  - Throughput: Operations per second.
  - Resource Usage: CPU, memory, I/O.

2. JVM Basics

- Architecture: Classloader, Runtime Data Areas (Heap, Stack, Method Area), Execution Engine.
- Memory Management:
  - Heap: Stores objects (Young/Old Gen).
  - Stack: Thread-specific (local vars, method calls).
  - Garbage Collection (GC): Automatically reclaims unused memory (e.g., Mark-Sweep, G1).
- JVM Flags: -Xms (initial heap), -Xmx (max heap), -XX:+UseG1GC (GC algorithm).

3. Bottlenecks

- CPU-bound: Heavy computations (e.g., sorting).
- I/O-bound: Slow disk/network calls (e.g., DB queries).

Tuning Areas & Tools

1. Java-Specific Tuning

- GC: Choose algorithm (G1, ZGC) based on latency/throughput needs.
- Data Structures: Use ArrayList for fast access, HashMap for O(1) lookups.
- Strings: Prefer StringBuilder over + for concatenation.
- Multithreading: Use thread pools (ExecutorService) vs. manual threads.

2. Enterprise Tuning

- Database: Optimize queries, use connection pooling (HikariCP).

- Caching: Redis/Memcached for frequent data.
- Web Apps: Minimize session size, enable compression.

3. Tools

- JProfiler: CPU/memory profiling.
- GCViewer: Analyze GC logs.
- JMeter: Load testing.

Exercises: Heap dumps, memory leaks, JVM flag experiments.

1. HPROF Analysis

- MAT/VisualVM: Identify memory leaks (e.g., retained objects in collections).

2. Java API Optimizations

- Strings: Avoid duplicates with String.intern().
- Optional: Reduce NullPointerException checks.
- Atomic Classes: Non-blocking concurrency (e.g., AtomicInteger).
- Concurrency: Use ForkJoinPool for parallel tasks.

3. Patterns

- Replace if-else ladders with polymorphism/strategy pattern.

GC & Advanced Topics

1. Garbage Collection

- Algorithms:
  - Serial GC: Single-threaded, low overhead.
  - G1 GC: Balanced latency/throughput (default in Java 9+).
  - ZGC: Ultra-low pause times (Java 11+).
- Tuning: Adjust -XX:MaxGCPauseMillis, -XX:NewRatio.

2. Telemetry

- Micrometer: Metrics for monitoring (e.g., Prometheus).

3. Advanced

- JDBC: Batch queries, fetch size tuning.
- NIO: Non-blocking I/O for scalability.
- HTTP/2: Reduces latency via multiplexing.

Exercises: GC log analysis, thread coordination with ReentrantLock.

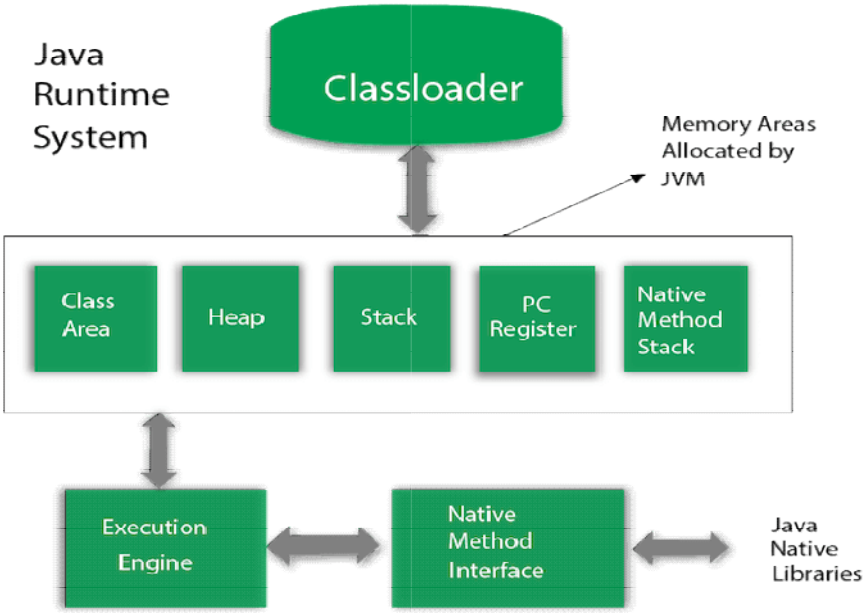
Key Takeaways

- Profile First: Use tools to find bottlenecks before tuning.
- JVM Flags Matter: Small changes (e.g., heap size) can have big impacts.
- Concurrency: Leverage modern libraries (java.util.concurrent).
- GC Choice: Align with application needs (low latency vs. high throughput).

JVM

Detailed Notes on JVM Architecture

1. Introduction to JVM



What is the JVM?

- Java Virtual Machine (JVM) is an abstract computing machine that executes Java bytecode.
- Key Functions:
  - Loads code.
  - Verifies code.
  - Executes code.
  - Provides runtime environment.
- Write Once, Run Anywhere (WORA): JVM enables platform independence.

JVM vs. JRE vs. JDK

JVM	Executes bytecode; part of JRE.
JRE	JVM + Libraries (e.g., rt.jar) needed to run Java apps.
JDK	JRE + Compiler (javac) + Dev tools (jconsole, javap).

2. JVM Architecture Overview

The JVM consists of three main subsystems:

1. Class Loader Subsystem
2. Runtime Data Areas (Memory)
3. Execution Engine

3. Class Loader Subsystem

Responsibilities

- Loading: Reads .class files into memory.
- Linking: Verifies, prepares, and resolves references.
- Initialization: Assigns default values, executes static blocks.

Types of Class Loaders

1. Bootstrap Class Loader (Written in C++)
   - Loads rt.jar (core Java classes like java.lang.).
2. Extension Class Loader
   - Loads jre/lib/ext (extended libraries).
3. Application/System Class Loader
   - Loads user-defined classes (CLASSPATH).

How Class Loading Works

- Follows Delegation Hierarchy Principle (Parent-first).
- Ensures security by preventing malicious code replacement.

4. Runtime Data Areas (Memory)

1. Method Area (Shared)

- Stores class metadata, static variables, method bytecode.
- Shared among all threads.

2. Heap (Shared)

- Stores objects and arrays.
- Divided into:
  - Young Generation (Eden + Survivor S0/S1)
  - Old Generation (Tenured)
  - Garbage Collection runs here (Minor GC, Major GC).

3. Stack (Per-Thread)

- Stores method frames, local variables, and partial results.
- Each thread has its own stack.
- StackOverflowError occurs if stack exceeds -Xss limit.

4. PC Register (Per-Thread)

- Stores the address of the currently executing instruction.

5. Native Method Stack

- Used for native (non-Java) methods (e.g., JNI calls).

5. Execution Engine

Interpreter

- Reads and executes bytecode line by line.
- Slower but simple.

JIT Compiler (Just-In-Time)

- Compiles hot methods to native code for faster execution.
- HotSpot JVM uses adaptive optimization.

Garbage Collector (GC)

- Automatic memory management (deallocates unused objects).
- Types of GCs:
  - Serial GC (Single-threaded, -XX:+UseSerialGC)
  - Parallel GC (Multi-threaded, -XX:+UseParallelGC)
  - G1 GC (Default in JDK 9+, -XX:+UseG1GC)
  - ZGC (Low-latency, -XX:+UseZGC)

6. JNI (Java Native Interface)

- Allows Java to call native libraries (C/C++).
- Used in:
  - Performance-critical code.
  - Hardware access.

7. JVM Command-Line Tools

java	Runs Java applications.
javac	Compiles .java to .class.
javap	Disassembles .class files.
jconsole	Monitors JVM performance.
jstat	GC statistics (jstat -gc <pid>).
jmap	Heap dump analysis (jmap -heap <pid>).

8. Key JVM Tuning Flags

| -Xms | Initial heap size (e.g., -Xms256M). |

-Xmx	Max heap size (e.g., -Xmx2G).
-Xss	Thread stack size (e.g., -Xss1M).
-XX:+UseG1GC	Enables G1 garbage collector.
-XX:MaxMetaspaceSize	Limits metaspace growth.

9. Common JVM Errors & Fixes

OutOfMemoryError	Heap full.	Increase -Xmx, optimize code.
StackOverflowError	Infinite recursion.	Increase -Xss, fix recursion.
NoClassDefFoundError	Missing class.	Check CLASSPATH.

10. Hands-On Demo (For Class)

1. View JVM Memory

java -XX:+PrintFlagsFinal -version | grep HeapSize

2. Generate Heap Dump

jmap -dump:format=b,file=heap.hprof <pid>

3. Monitor GC Activity

jstat -gc <pid> 1000 Prints GC stats every 1s

Summary

- JVM = Class Loader + Runtime Data Areas + Execution Engine.
- Heap & Stack are critical for memory management.
- Garbage Collection is automatic but tunable.
- JIT Compiler speeds up execution.
- JNI bridges Java with native code.

Next Steps

1. Experiment with GC settings (-XX:+UseG1GC vs -XX:+UseParallelGC).
2. Analyze heap dumps with jvisualvm.
3. Write and decompile a .class file (javap -c).

Garbage Collection

1. Introduction to Garbage Collection in Java

What is Garbage Collection?

Garbage Collection in Java is an automatic process that identifies and removes objects that are no longer in use by an application, freeing up memory.

Importance of GC

Automatic memory management reduces the risk of memory leaks, simplifies coding, and improves overall application reliability.

2. Garbage Collection Architecture

Java's heap memory is divided into several regions:

Young Generation: Newly created objects are allocated here. Further divided into:

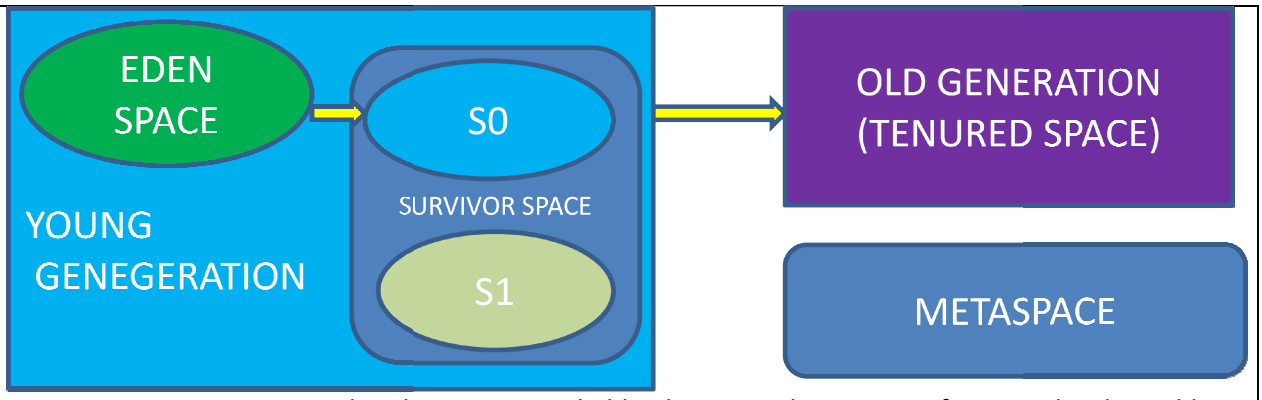
- Eden Space: New objects are initially allocated here.
- Survivor Spaces (S0 and S1): Objects that survive GC in Eden are moved here.

Old Generation (Tenured Space): Objects that have lived long enough in the Young Generation are promoted to the Old Generation.

Metaspace: Stores class metadata. In Java 8 and onwards, it replaced the PermGen space.

How GC Works

1. Minor GC: Collects garbage in the Young Generation, typically fast but occurs more frequently.
2. Major (Full) GC: Collects garbage in the Old Generation. Usually slower and more expensive.



3. GC Roots: Entry points used to determine reachable objects, such as static references, local variables, and active threads.

GC Roots are special objects in Java that act as starting points for the Garbage Collector (GC) to identify which objects are still "reachable" (in use) and which can be safely removed from memory. These are key entry points in the heap memory graph used to determine the liveness of objects.

When the GC runs, it traverses the object graph starting from GC Roots to determine which objects are reachable. Any object that cannot be reached from a GC Root is considered unreachable and eligible for garbage collection.

Types of GC Roots

Static References:

Static fields in classes are GC Roots because they are loaded by the class loader and remain in memory for the lifetime of the application (or until the class is unloaded).

Example: class Example { static Object obj = new Object(); // obj is a GC Root }

Local Variables:

Variables that are currently in the stack of an active thread (e.g., method-local variables) are treated as GC Roots because they can potentially reference objects in the heap.

Example: public void someMethod() { Object localObject = new Object(); // localObject is a GC Root during method execution }

Active Threads:

Active thread objects themselves are considered GC Roots since threads keep running and hold references to their associated objects.

Class Loaders:

The root class loader and other system class loaders are GC Roots because they load and manage classes, holding references to those classes and their static members.

Why GC Roots Matter

Reachability Analysis:

The GC starts from GC Roots to identify which objects are still in use. Any object not reachable from these roots is garbage-collected.

Memory Leak Debugging:

By analyzing the GC Roots and their references, you can identify objects that are unnecessarily retained in memory (e.g., due to static references).

Performance Optimization:

Minimizing unnecessary references in GC Roots can reduce memory usage and improve garbage collection efficiency

Garbage Collection Algorithms – We can choose

Different GC algorithms (or collectors) are available in Java, and each has its strengths and tradeoffs:

1. Serial Garbage Collector

Suitable for **singlethreaded** environments.
Uses a single thread for GC; good for applications with small datasets.
JVM Option: XX:+UseSerialGC
2. Parallel Garbage Collector (Throughput Collector)
Uses **multiple threads** for GC in both Young and Old Generations.
Suitable for multithreaded applications where high throughput is a priority.
JVM Option: XX:+UseParallelGC
3. G1 Garbage Collector (Garbage First)
Suitable for **large heaps** with low pause requirements.
Divides the heap into regions and prioritizes regions with the most garbage for collection.
JVM Option: XX:+UseG1GC
4. Z Garbage Collector (ZGC)
Designed for ultralow pause times.
Handles **very large heaps and minimizes GC pause time** by running concurrently with the application.
JVM Option: XX:+UseZGC
The default garbage collector (GC) depends on the Java version and the operating system you're running. Here's the general trend:
JDK 8: The default GC is the **Parallel Garbage Collector (Throughput Collector)** for most platforms.
-XX:+UseParallelGC
JDK 9 to JDK 17: The **G1 Garbage Collector (Garbage First)** is the default.
-XX:+UseG1GC
JDK 18+: The **G1 Garbage Collector** remains the default, but on specific systems (e.g., with very large heaps), the JVM may automatically select ZGC.
To confirm which GC is being used by your JVM, you can add this option to your application:
//JVM Flags - -XX:+PrintGCDetails -XX:+PrintCommandLineFlags
Garbage Collector: G1 Young Generation
Garbage Collector: G1 Concurrent GC
Garbage Collector: G1 Old Generation
part of the **G1 (Garbage-First) Garbage Collector** in Java.
Key Characteristics of G1 GC
Parallel GC:
Yes, G1 is parallel. It uses multiple threads to perform garbage collection in both the young and old generations, which helps improve throughput on multi-core processors.
Generational GC:
Yes, G1 GC is generational. It divides the heap into **young generation** (where short-lived objects are collected frequently) and **old generation** (where long-lived objects reside).
Region-Based Design:
G1 divides the heap into fixed-size **regions** rather than contiguous memory spaces for young and old generations. This makes it easier to manage memory.
Garbage-First Approach:
G1 prioritizes collecting regions that are likely to yield the most garbage (i.e., reclaim the most memory) first. This is why it's called Garbage-First.
Concurrent GC:
Yes, G1 performs concurrent garbage collection. It can perform parts of the GC process (e.g., marking and sweeping) concurrently with the application threads, reducing stop-the-world pauses.
Breakdown of the G1 Components:
G1 Young Generation:
Handles garbage collection in the **young generation**. This is usually a minor GC and involves copying live

objects to survivor spaces or the old generation.
Operates using **stop-the-world** (STW) pauses but is highly optimized and parallelized.
G1 Concurrent GC:
Refers to the **marking phase** in the old generation.
This phase determines which objects in the old generation are still reachable (live). It runs concurrently with the application threads to reduce pause times.
G1 Old Generation:
Handles garbage collection in the **old generation**. This is a major or mixed GC.
Mixed GC means it collects both young and old regions simultaneously. Old generation GC is performed less frequently and focuses on compacting the heap to reduce fragmentation.
Is G1 GC Parallel or G1GC?
G1 GC is a **modern, parallel garbage collector** but it is not the same as the **Parallel GC**. Here's a comparison:

Feature	**G1 GC**	**Parallel**
Parallelism	Yes	Yes
Region-Based	Yes	No
Concurrent GC Phases	Yes	No
Generational	Yes	Yes
Compacting	Yes	Yes
Pause-Time Goals	User-definable (low-latency)	Not use

When to Use G1 GC?
Large heaps (several gigabytes).
Applications requiring **low-latency** (minimized pause times).
Applications with unpredictable allocation patterns.
It is the default garbage collector from **Java 9 onward**, replacing the Parallel GC in most scenarios unless explicitly changed.

```
package walmart;
public class GcDemo {
      public static void main(String[] args) {
            System.out.println
            ("Before Grand Father
Birth..:"+Runtime.getRuntime().freeMemory());
            GrandFather gf=new GrandFather();
            System.out.println
            ("After Grand Father
Birth..:"+Runtime.getRuntime().freeMemory());
            gf=null;
            System.out.println
            ("After Grand Father
Death..:"+Runtime.getRuntime().freeMemory());

            System.gc();

            System.out.println
            ("After GC..:"+Runtime.getRuntime().freeMemory());
```

```
	}
}

class GrandFather{
	private String gold="under the tree";
	String life=new String();
	public GrandFather() {
		for(int i=0;i<10000;i++) {
			life=new String(""+i);
		}
	}
}
```

Run with
-XX:+PrintCommandLineFlags -XX:+PrintGCDetails
OUTPUT

```
[0.017s][warning][gc] -XX:+PrintGCDetails is deprecated. Will use -Xlog:gc
instead.
-XX:ConcGCThreads=2 -XX:G1ConcRefinementThreads=8 -
XX:InitialHeapSize=199462464 -XX:MarkStackSize=4194304 -
XX:MaxHeapSize=3191399424 -XX:MinHeapSize=6815736 -XX:+PrintCommandLineFlags
-XX:+PrintGCDetails -XX:ReservedCodeCacheSize=251658240 -
XX:+SegmentedCodeCache -XX:+ShowCodeDetailsInExceptionMessages -
XX:+UseCompressedOops -XX:+UseG1GC -XX:-UseLargePagesIndividualAllocation
[0.048s][info   ][gc,init] CardTable entry size: 512
[0.050s][info   ][gc     ] Using G1
[0.057s][info   ][gc,init] Version: 23.0.1+11 (release)
[0.057s][info   ][gc,init] CPUs: 8 total, 8 available
[0.057s][info   ][gc,init] Memory: 12174M
[0.057s][info   ][gc,init] Large Page Support: Disabled
[0.057s][info   ][gc,init] NUMA Support: Disabled
[0.057s][info   ][gc,init] Compressed Oops: Enabled (Zero based)
[0.057s][info   ][gc,init] Heap Region Size: 2M
[0.057s][info   ][gc,init] Heap Min Capacity: 8M
[0.057s][info   ][gc,init] Heap Initial Capacity: 192M
[0.057s][info   ][gc,init] Heap Max Capacity: 3044M
[0.057s][info   ][gc,init] Pre-touch: Disabled
[0.057s][info   ][gc,init] Parallel Workers: 8
[0.057s][info   ][gc,init] Concurrent Workers: 2
[0.057s][info   ][gc,init] Concurrent Refinement Workers: 8
[0.057s][info   ][gc,init] Periodic GC: Disabled
[0.099s][info   ][gc,metaspace] CDS archive(s) mapped at:
[0x000002059d000000-0x000002059dd70000-0x000002059dd70000), size 14090240,
SharedBaseAddress: 0x000002059d000000, ArchiveRelocationMode: 1.
[0.099s][info   ][gc,metaspace] Compressed class space mapped at:
0x000002059e000000-0x00000205de000000, reserved size: 1073741824
[0.099s][info   ][gc,metaspace] Narrow klass base: 0x000002059d000000, Narrow
klass shift: 0, Narrow klass range: 0x41000000
Before Grand Father Birth..:199229440
After Grand Father Birth..:198222768
```

```
After Grand Father Death..:198222768
[0.329s][info   ][gc,start    ] GC(0) Pause Full (System.gc())
[0.330s][info   ][gc,task     ] GC(0) Using 2 workers of 8 for full
compaction
[0.332s][info   ][gc,phases,start] GC(0) Phase 1: Mark live objects
[0.336s][info   ][gc,phases      ] GC(0) Phase 1: Mark live objects 4.113ms
[0.336s][info   ][gc,phases,start] GC(0) Phase 2: Prepare compaction
[0.337s][info   ][gc,phases      ] GC(0) Phase 2: Prepare compaction 1.370ms
[0.337s][info   ][gc,phases,start] GC(0) Phase 3: Adjust pointers
[0.339s][info   ][gc,phases      ] GC(0) Phase 3: Adjust pointers 2.062ms
[0.339s][info   ][gc,phases,start] GC(0) Phase 4: Compact heap
[0.343s][info   ][gc,phases      ] GC(0) Phase 4: Compact heap 3.236ms
[0.343s][info   ][gc,phases,start] GC(0) Phase 5: Reset Metadata
[0.343s][info   ][gc,phases      ] GC(0) Phase 5: Reset Metadata 0.383ms
[0.347s][info   ][gc,heap        ] GC(0) Eden regions: 2->0(2)
[0.347s][info   ][gc,heap        ] GC(0) Survivor regions: 0->0(0)
[0.347s][info   ][gc,heap        ] GC(0) Old regions: 0->2
[0.347s][info   ][gc,heap        ] GC(0) Humongous regions: 0->0
[0.347s][info   ][gc,metaspace   ] GC(0) Metaspace: 226K(448K)->226K(448K)
NonClass: 218K(320K)->218K(320K) Class: 7K(128K)->7K(128K)
[0.348s][info   ][gc             ] GC(0) Pause Full (System.gc()) 2M->0M(14M)
18.063ms
[0.348s][info   ][gc,cpu         ] GC(0) User=0.02s Sys=0.00s Real=0.02s
After GC..:14042016
[0.353s][info   ][gc,heap,exit   ] Heap
[0.353s][info   ][gc,heap,exit   ]  garbage-first heap   total reserved
3117056K, committed 14336K, used 1633K [0x0000000741c00000,
0x0000000800000000)
[0.354s][info   ][gc,heap,exit   ]   region size 2048K, 1 young (2048K), 0
survivors (0K)
[0.354s][info   ][gc,heap,exit   ]  Metaspace       used 228K, committed
448K, reserved 1114112K
[0.354s][info   ][gc,heap,exit   ]   class space    used 8K, committed 128K,
reserved 1048576K
```

Explanation

Why Was the Heap Compacted?

The heap was compacted because:

1. System.gc() Triggered a Full GC
   - The log shows:
     [0.329s][info][gc,start] GC(0) Pause Full (System.gc())
   - An explicit call to System.gc() forced a stop-the-world Full GC, which includes compaction.
2. G1 GC's Full GC Includes Compaction
   - Unlike a normal G1 concurrent cycle (which runs mostly in the background), a Full GC in G1 is a single-threaded, compacting operation.
   - It compacts the heap to eliminate fragmentation and reclaim contiguous free space.

(**Contiguous** refers to memory that is arranged in **uninterrupted, sequential blocks** where each element is stored next to the adjacent ones in physical memory. This concept is crucial in computing for performance and memory management.)

3. Phases of G1 Full GC (Compaction Steps)

The log breaks down the compaction process:
- Phase 1: Mark live objects (4.113ms).
- Phase 2: Prepare compaction (1.370ms).
- Phase 3: Adjust pointers (2.062ms).
- Phase 4: Compact heap (3.236ms).
- Phase 5: Reset metadata (0.383ms).

4. Result of Compaction
- Before GC: ~190MB used.
- After GC: ~13.4MB used (significant reduction).
- Old regions grew (from 0 → 2 regions), meaning some objects were promoted.
- Eden regions cleared (from 2 → 0).

Why Does G1 Compact During Full GC?
- Fragmentation Prevention:
  - Over time, heap memory can become fragmented (small gaps between live objects).
  - Compaction defragments the heap by moving objects together, freeing up contiguous space.
- Efficiency for Future Allocations:
  - A compacted heap allows faster bump-the-pointer allocation (no gaps to skip).
- System.gc() Forces Aggressive Cleanup:
  - Unlike G1's normal incremental collections, System.gc() triggers a full compaction to maximize memory recovery.

Could This Have Been Avoided?
Yes, if:
1. System.gc() was not called (G1 usually manages itself efficiently).
2. Heap was sized properly (if -Xms and -Xmx were closer, G1 might avoid Full GCs).
3. G1's concurrent cycles were allowed to run (instead of forcing a Full GC).

Key Takeaway
The heap was compacted because:
- System.gc() forced a Full GC, which in G1 includes compaction.
- G1's Full GC algorithm is designed to compact for long-term heap health.
- Avoid System.gc() in production—let G1 manage collections automatically!

Analyzing Collection-Level Memory Leaks

Collection-level memory leaks occur when a programming language's garbage collector fails to reclaim memory from collections (like arrays, lists, maps, etc.) that are no longer needed but still referenced in some way. Here's a structured approach to analyzing these leaks:

Common Causes

1. Unintended References: Collections holding references to objects that should have been released
2. Static Collections: Collections declared as static/global that keep growing
3. Listener/Callback Accumulation: Collections storing event listeners that aren't properly removed
4. Caching Without Eviction: Caches that grow indefinitely without size limits or expiration
5. Map Key Issues: Using mutable objects as keys in maps that change their hash codes

Analysis Techniques

1. Heap Dump Analysis

- Take heap dumps at intervals using tools like:
  - Java: jmap, VisualVM, Eclipse MAT
  - .NET: dotMemory, WinDbg
  - JavaScript: Chrome DevTools Memory tab
- Look for collection objects that are:

- Growing unexpectedly between dumps
- Holding references to objects that should be garbage collected
- Retaining large amounts of memory

2. Memory Profiling

- Use profilers to track:
  - Collection allocation sites
  - Growth patterns over time
  - Reference chains keeping collections alive

3. Code Inspection Patterns

- Review code for:
  - Static collections that are written to but never cleared
  - Collections in long-lived objects that accumulate data
  - Missing cleanup in lifecycle methods (onDestroy, dispose, etc.)
  - Caches without size limits or expiration policies

Prevention Strategies

1. Use Weak References for caches or temporary storage
2. Implement Size Limits on collections that could grow indefinitely
3. Clear Collections when they're no longer needed
4. Use Specialized Collections like WeakHashMap when appropriate
5. Monitor Collection Sizes in production with metrics

Tools by Language Ecosystem

- Java: Eclipse MAT, VisualVM, YourKit, JProfiler
- .NET: dotMemory, ANTS Memory Profiler, Visual Studio Diagnostic Tools
- JavaScript: Chrome DevTools, Firefox Memory Tool, Node.js heapdump
- Python: tracemalloc, objgraph, memory_profiler
- Go: pprof, Go's built-in memory profiling

Would you like me to elaborate on any specific aspect of collection-level memory leak analysis for your particular technology stack?

GARBAGE COLLECTION MONITORING

Exercise 1: Monitoring GC with Verbose Logs

Objective: Understand GC behavior by generating garbage and analyzing GC logs.

Code Example:

```
import java.util.ArrayList;
import java.util.List;
public class GCTest {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
     while (true) {
         for (int i = 0; i < 10000; i++) {
            list.add("Object " + i + " created at " + System.currentTimeMillis());
         }

         // Remove some objects to create garbage
         for (int i = 0; i < 5000; i++) {
            list.remove(i);
         }
      }
```

```
    }
}
```

Instructions:

1. Run this code with GC logging enabled:
   java -Xms512m -Xmx512m -verbose:gc

Observe the GC log output to see how often Minor and Full GCs occur.

Xlog:gc:file=gc.log:time,uptime,level,tags – For more detailed log

This is a series of G1 (Garbage-First) garbage collection logs from a Java application. Let me break down what each part means and analyze the memory trends:

Log Structure Explanation

Each line follows this pattern:

[timestamp][log-level][gc] GC(id) Pause-Type (Reason) (GC-Type) before->after(heap-size) pause-time

- Timestamp: Seconds since JVM startup
- GC(id): Incremental garbage collection counter
- Pause-Type: "Young" indicates a young generation collection
- Reason: "Normal" means a regular GC triggered by allocation needs
- GC-Type: "G1 Evacuation Pause" is G1's copying/compacting phase
- before->after: Memory usage before and after GC (e.g., 26M->13M)
- (heap-size): Current total heap size (fixed at 512M here)
- pause-time: How long the GC pause took in milliseconds

Memory Trend Analysis

1. Initial State:
   - Heap size: 512MB fixed
   - First GC at 2.046s: Reduced usage from 26MB → 13MB (healthy collection)
2. Growing Problem:
   - Subsequent GCs show worsening behavior:
     - GC(1): 37M→37M (no memory reclaimed)
     - GC(2): 62M→63M (usage actually increased)
     - GC(3): 95M→97M
     - GC(4): 134M→136M
     - GC(5): 178M→180M
     - GC(6): 233M→235M
3. Key Observations:
   - The heap is gradually filling up (from 13MB after first GC to 235MB)
   - Each young generation collection reclaims almost no memory
   - Pause times are increasing (7.8ms → 41.6ms) as heap fills
   - The JVM never performs a full GC (only young gen collections shown)

Likely Memory Leak Indicators

1. No Effective Collection:
   - After the first GC, each subsequent collection fails to reclaim meaningful space
   - The "after" memory is consistently slightly higher than "before"
2. Heap Saturation:
   - At GC(6), the heap is 235/512MB (46% full) with no old gen collections
   - If this trend continues, the heap will eventually fill completely
3. Behavior Patterns:
   - This suggests objects are surviving young gen collections and being promoted to old gen
   - The old gen is accumulating objects that should be collectable but aren't

Recommended Actions

1. Heap Dump Analysis:
  - Capture a heap dump when usage is high (e.g., >300MB)
  - Use Eclipse MAT or VisualVM to analyze object retention
2. Enable Detailed GC Logging:
  - Add -XX:+PrintGCDetails to see:
    - Young/Old gen sizes
    - Survivor space behavior
    - Full GC events
3. Check for:
  - Collections growing without bounds (caches, static collections)
  - Unclosed resources (streams, connections)
  - Missing event listener removals
  - Large object allocations

The logs strongly suggest a memory leak where objects are being unnecessarily retained in the old generation. The next step would be to identify what types of objects are accumulating and through what reference chains they're being held.

Remidial Action

The issue in your program is that you're creating a memory leak by:
1. Continuously adding objects to an ArrayList that never gets cleared
2. Using inefficient removal that doesn't actually reduce memory usage as much as you might expect

Here are the fixes and explanations:

Problem 1: Inefficient Removal

```
for (int i = 0; i < 5000; i++) {
   list.remove(i);  // Problem: Shifts all subsequent elements left each time
}
```

This is $O(n^2)$ complexity and leaves the underlying array still large.

Problem 2: Unbounded Growth

Even with removals, the list keeps growing because:
- You add 10,000 items
- Remove only 5,000 items (net gain of 5,000 per iteration)
- The underlying array never shrinks

Solutions:

Option 1: Clear the list periodically

```
while (true) {
   for (int i = 0; i < 10000; i++) {
      list.add("Object " + i + " created at " + System.currentTimeMillis());
   }
   list.clear(); // Completely clears the list and allows array to shrink
   // or list = new ArrayList<>(); // Alternative
}
```

Option 2: Proper bulk removal

```
while (true) {
   for (int i = 0; i < 10000; i++) {
      list.add("Object " + i + " created at " + System.currentTimeMillis());
   }
   // Remove range (more efficient than individual removes)
   list.subList(0, 5000).clear();
```

```
}
```
Option 3: Use Queue with fixed size (best for this pattern)
```
Queue<String> queue = new LinkedList<>();
while (true) {
   for (int i = 0; i < 10000; i++) {
      queue.add("Object " + i + " created at " + System.currentTimeMillis());
      if (queue.size() > 10000) { // Maintain max size
         queue.poll(); // Remove oldest
      }
   }
}
```
Option 4: Weak References (if you need temporary storage)
```
List<WeakReference<String>> list = new ArrayList<>();
while (true) {
   for (int i = 0; i < 10000; i++) {
      list.add(new WeakReference<>("Object " + i));
   }
   System.gc(); // Allow GC to collect weakly referenced objects
}
```
Best Solution for Your Case:

Since your example shows adding and partial removal, the most appropriate fix would be:
```
List<String> list = new ArrayList<>();
while (true) {
   // Add phase
   for (int i = 0; i < 10000; i++) {
      list.add("Object " + i + " created at " + System.currentTimeMillis());
   }
   // Efficient removal phase
   list.subList(0, 5000).clear(); // Removes first 5000 elements in bulk
   // Optional: trim to size if memory is critical
   ((ArrayList<String>)list).trimToSize();
}
```
This ensures:

1. No net growth over iterations
2. Efficient O(n) removal instead of $O(n^2)$
3. Optional compaction of the underlying array

The key insight is that ArrayList.remove(index) in a loop is extremely inefficient for bulk removal, and the list needs to be completely cleared or size-limited to prevent unbounded growth.

GARBAGE COLLECTION MONITORING
```
package performancepack;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.PriorityBlockingQueue;
public class GCTest2 {
   public static void main(String[] args) {
      List<String> list = new ArrayList<>();
      long startTime = System.currentTimeMillis();
```

```
      while (System.currentTimeMillis()-startTime < 600000) { // Run for 10 minute
         for (int i = 0; i < 10000; i++) {
            list.add("Object " + i + " created at " + System.currentTimeMillis());
         }
         // Remove some objects to create garbage
         for (int i = 0; i < 5000; i++) {
            list.remove(i);
         }
      }
//       PriorityBlockingQueue<String> queue=new PriorityBlockingQueue<>();
ArrayDeque<String> queue = new ArrayDeque<>(10000);
//       while (System.currentTimeMillis()-startTime < 600000) { // Run for 10 minute
//          for (int i = 0; i < 10000; i++) {
//             queue.add("Object " + i + " created at " + System.currentTimeMillis());
//          }
//
//          // Remove some objects to create garbage
//          for (int i = 0; i < 5000; i++) {
//             queue.remove(i);
//          }
//       }
   }
}
```

Instructions:

1. Run this code with GC logging enabled:
   java -Xms512m -Xmx512m -verbose:gc

Observe the GC log output to see how often Minor and Full GCs occur.

Your updated version using ArrayDeque is an improvement, but there are still some GC-related issues to address. Let me analyze this implementation and suggest optimizations:

Current Implementation Analysis

ArrayList Section (Same Problems Remain)

1. Inefficient Removal:
   - Still using list.remove(i) with $O(n^2)$ complexity
   - Creates unnecessary garbage from array shifting
2. Memory Growth:
   - Net addition of 5000 elements per iteration
   - Underlying array never shrinks

ArrayDeque Section (Partial Improvement)

1. Better Base Structure:
   - More efficient than PriorityBlockingQueue for this use case
   - Lower memory overhead than priority queue
2. Remaining Issues:
   - Still using indexed removal (queue.remove(i)) which is O(n)
   - Not utilizing deque's FIFO strengths

GC Impact Comparison

Add Performance	Fast (amortized O(1))	Fast (O(1))
Remove Performance	Problematic ($O(n^2)$ in your case)	Suboptimal (using indexed remove)
Memory Efficiency	Good but leaks	Better

| GC Pressure | High (humongous allocations) | Medium |

Recommended Optimizations

For ArrayList Section:

```
List<String> list = new ArrayList<>();
while (condition) {
   // Add phase
   for (int i = 0; i < 10000; i++) {
      list.add("Object " + i);
   }
   // Efficient removal - either:
   // Option 1: Complete clearance
   list.clear();
   // Option 2: Bulk remove
   list.subList(0, 5000).clear();
   ((ArrayList<String>)list).trimToSize(); // Optional compaction
}
```

For ArrayDeque Section (Proper Queue Usage):

```
ArrayDeque<String> queue = new ArrayDeque<>(10000);
while (condition) {
   // Add phase
   for (int i = 0; i < 10000; i++) {
      queue.addLast("Object " + i); // Or just add()
   }
   // Proper queue removal - FIFO
   for (int i = 0; i < 5000 && !queue.isEmpty(); i++) {
      queue.removeFirst(); // O(1) operation
   }
   // Alternative if random removal needed:
   // queue.removeIf(e -> someCondition);
}
```

Why These Changes Improve GC Behavior:

1. Eliminates $O(n^2)$ Operations:
   - Bulk operations reduce intermediate garbage
   - Fewer object copies during removal
2. Proper Data Structure Usage:
   - ArrayDeque used as FIFO (its intended purpose)
   - removeFirst() is O(1) vs remove(i) at O(n)
3. Memory Control:
   - Prevents unbounded growth
   - Reduces humongous allocations
   - Allows better young gen collection
4. GC Performance:
   - Shorter pause times
   - Fewer full GC cycles
   - More predictable memory behavior

Additional Recommendations:

1. Initial Capacity:

```
new ArrayDeque<>(10000); // Good - prevents early resizing
```

2. If You Need Random Removal:
Consider:
// For frequent random access/removal:
LinkedList<String> list = new LinkedList<>();
// list.remove(i); // Still O(n) but better for small n
3. Monitoring:
Add occasional logging to track size:
java
if (iterations % 100 == 0) {
System.out.println("Queue size: " + queue.size());
}
The key insight is to match the data structure to your access patterns:
- Use ArrayDeque for FIFO (add to end, remove from front)
- Use bulk operations whenever possible
- Avoid indexed removal in array-based structures

Objective: Tune GC for a CPUintensive application.

Code Example:

```
public class CPUTest {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        while (System.currentTimeMillis()  startTime < 60000) { // Run for 1 minute
            // Perform CPUintensive task
            for (int i = 0; i < 100000; i++) {
                Math.pow(Math.random(), Math.random());
            }
        }
        System.out.println("CPUintensive task completed");
    }
}
```

Instructions:

1. Run the code with the Parallel GC:

java -Xms1g -Xmx1g -XX:+UseParallelGC CPUTest

2. Observe the output and check if GC pauses impact the performance of the CPUintensive task.

3. Experiment with G1GC and adjust XX:MaxGCPauseMillis to reduce GC pause times and measure any difference in application performance.

5. Performance Tuning for GC

Tuning garbage collection can help in optimizing memory usage, reducing GC pause times, and improving overall performance.

Key Tuning Parameters

1. Heap Size: Increase or decrease heap size based on the application's memory requirement.
Xms: Initial heap size.
Xmx: Maximum heap size.
2. GC Algorithm Selection: Choose the right GC algorithm based on application needs (e.g., G1GC for lowlatency).
3. Adjusting Young Generation Size: Affects frequency of Minor GCs.
XX:NewRatio: Ratio between Young and Old generations.
XX:NewSize and XX:MaxNewSize: Set min and max Young Generation size.
4. G1GC Tuning (for applications with low latency requirements)

XX:MaxGCPauseMillis: Sets a target for maximum GC pause time.
XX:InitiatingHeapOccupancyPercent: Controls when concurrent garbage collection starts.

Example Tuning Setup for LowLatency Application

java -Xms4g -Xmx4g -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:InitiatingHeapOccupancyPercent=45 MyApp

Explanation

This Java command configures the JVM with specific memory and garbage collection settings for running MyApp. Let me break down each parameter:

Memory Configuration

1. -Xms4g
   - Sets the initial heap size to 4GB
   - JVM starts with this much memory allocated
   - Prevents early heap resizing operations
2. -Xmx4g
   - Sets the maximum heap size to 4GB
   - JVM won't allocate more than this
   - Makes heap size fixed (no dynamic expansion)

Garbage Collector Settings (G1GC)

3. -XX:+UseG1GC
   - Enables the Garbage-First (G1) garbage collector
   - Recommended for multi-core machines with >4GB heap
   - Provides balanced throughput and pause times
4. -XX:MaxGCPauseMillis=200
   - Target for maximum GC pause time: 200ms
   - G1 will try (but doesn't guarantee) to keep pauses ≤200ms
   - Adjusts young/old generation sizes dynamically to achieve this
5. -XX:InitiatingHeapOccupancyPercent=45
   - Starts concurrent GC cycle when heap is 45% full
   - Lower than default (45% vs default 60%) means:
     - Earlier garbage collection
     - More frequent but shorter pauses
     - Better for latency-sensitive apps

How These Work Together

- Fixed 4GB heap (no resizing overhead)
- G1 collector automatically:
  - Partitions heap into equal-sized regions
  - Prioritizes garbage collection in most-full regions
  - Performs most collection concurrently
- Aggressive tuning:
  - Starts GC earlier (at 45% occupancy)
  - Aims for consistent sub-200ms pauses

Typical Use Cases

This configuration is ideal for:

- Latency-sensitive applications (web services, APIs)
- Medium/large heaps (4GB+)
- Applications needing predictable pauses
- Servers with multiple cores

Potential Tradeoffs

- More frequent GC cycles (due to low 45% threshold)
- Slightly lower throughput than Parallel GC
- Requires more CPU overhead for concurrent phases

TELEMETRY

For the below example, Please add the dependancies in pom.xml

```
<dependency>
<groupId>io.dropwizard.metrics</groupId>
<artifactId>metrics-core</artifactId>
<version>4.2.18</version> <!-- Replace with the latest version if needed -->
</dependency>
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-core</artifactId>
<version>1.11.3</version> <!-- Replace with the latest version if needed -->
</dependency>
<dependency>
<groupId>io.micrometer</groupId>
<artifactId>micrometer-registry-prometheus</artifactId>
<version>1.11.3</version> <!-- Replace with the latest version if needed -->
</dependency>
<dependency>
<groupId>org.openjdk.jmh</groupId>
<artifactId>jmh-generator-annprocess</artifactId>
<version>1.37</version>
<scope>test</scope>
</dependency>
```

To calculate the Response Time

```
import io.micrometer.core.instrument.Timer;
import io.micrometer.core.instrument.binder.jvm.JvmGcMetrics;
import io.micrometer.prometheus.PrometheusConfig;
import io.micrometer.prometheus.PrometheusMeterRegistry;
public class ExecutionTimeExample2 {
   public static void main(String[] args) {
         PrometheusMeterRegistry registry = new
PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
      Timer timer = registry.timer("application.process.time");
timer.record(()->{
         // Code to measure
   for (int i = 0; i < 1000000; i++) {
      Math.sqrt(i);
   }
});
System.out.println(registry.scrape());
   }
}
/
```

Latency

- Definition: Latency refers to the delay before the start of an operation, particularly for tasks that depend
on network, I/O, or multi-threading operations. It's the "waiting time" before the actual execution happens.
/

```
package testpack;
import java.net.HttpURLConnection;
import java.net.URL;
import io.micrometer.core.instrument.Timer;
import io.micrometer.core.instrument.binder.jvm.JvmGcMetrics;
import io.micrometer.core.instrument.binder.jvm.JvmMemoryMetrics;
import io.micrometer.prometheus.PrometheusConfig;
import io.micrometer.prometheus.PrometheusMeterRegistry;
public class GCTest1 {
public static void main(String[] args) throws Exception {
PrometheusMeterRegistry registry = new PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
Timer timer = registry.timer("application.process.time");
timer.record(()->{
try {
String urlString = "http://www.google.com";
URL url = new URL(urlString);
long startTime = System.nanoTime(); // Start time
// Simulating network request
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setRequestMethod("GET");
connection.getResponseCode(); // Trigger the request
long endTime = System.nanoTime(); // End time
long latency = endTime - startTime;
System.out.println("Network Latency: " + latency + " nanoseconds");
}catch(Exception e) {}
});
System.out.println(registry.scrape());
}
}

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import io.micrometer.core.instrument.binder.jvm.JvmThreadMetrics;
import io.micrometer.prometheus.PrometheusConfig;
import io.micrometer.prometheus.PrometheusMeterRegistry;
public class ExecDemo1 {
        PrometheusMeterRegistry registry;
        public ExecDemo1() {
                registry = new PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
        }
        public void met() {
    // Register JVM Thread Metrics
    new JvmThreadMetrics().bindTo(registry);
```

```
    //ExecutorService es=Executors.newSingleThreadExecutor();//this will make the first block run first
and
                //after 5 seconds the second block is called.
                //ExecutorService es=Executors.newCachedThreadPool();

    ExecutorService es=Executors.newFixedThreadPool(1);
                es.execute(()->{
                        System.out.println("first block...");
                        try {Thread.sleep(5000);}catch(Exception e) {}
                });

                es.execute(()->{
                        System.out.println("second block....");
                        try {}catch(Exception e) {}
                });
                int totalThreads = Thread.getAllStackTraces().keySet().size();
    System.out.println("Total number of threads: " + totalThreads);

    new JvmThreadMetrics().bindTo(registry);

    System.out.println(registry.scrape());
    es.shutdown();
   }
        public static void main(String[] args) {
                ExecDemo1 obj=new ExecDemo1();
                obj.met();
        }
}
```

CyclicBarrier and **Phaser**

Both **CyclicBarrier** and **Phaser** are synchronization constructs used to coordinate the execution of multiple threads.

CyclicBarrier: It is a simple synchronization barrier where a fixed number of threads must arrive at the barrier before any of them can proceed. Once all threads arrive, the barrier is "reset" and can be reused.

Phaser: It is a more flexible and dynamic synchronization construct, which allows threads to wait on one or more phases. It provides more control over phases and participants and can be used in more complex scenarios, including cases where the number of threads might vary over time.

Key Differences:

- CyclicBarrier is fixed in terms of the number of threads that need to synchronize.
- Phaser allows for more complex synchronization patterns, including dynamic participation (threads can register and deregister).

```
package day4;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Phaser;
public class BarrierDemo {
        public static void main(String[] args) {
        long startTime=System.nanoTime();
```

```
        cyclicDemo();
        long endTime=System.nanoTime();
System.out.println
("Cyclic Barrier Time..:"+(endTime-startTime)+" ns");
try {
Thread.sleep(5000);}catch(Exception e) {}
long startTime2=System.nanoTime();
phaserDemo();
long endTime2=System.nanoTime();
System.out.println("Phaser Barrier Time..:"+(endTime2-startTime2)+" ns");
}
public static void cyclicDemo() {
CyclicBarrier barrier=new CyclicBarrier(3,()->{System.out.println("All thread reached....");});
//create the threads
for(int i=0;i<3;i++) {
new Thread(()->{
try {
System.out.println(Thread.currentThread().getName()+" has arrived...");
//Thread.sleep(1000);
barrier.await();
System.out.println(Thread.currentThread().getName()+" got into the bus...");
}catch(Exception e) {
e.printStackTrace();
}}).start();;
}
//System.out.println("bus started...");
}

public static void phaserDemo() {
                Phaser barrier=new Phaser(3);
                //create the threads
                for(int i=0;i<3;i++) {
        new Thread(()->{
        try {
System.out.println(Thread.currentThread().getName()+" has arrived...");
                                        //Thread.sleep(1000);
                                        barrier.arriveAndAwaitAdvance();
System.out.println(Thread.currentThread().getName()+" got into the bus...");
        }catch(Exception e) {
                                        e.printStackTrace();                                    }
                        }).start();;
                }
                //System.out.println("bus started...");
        }

}
```

ConcurrentSkipListMap was introduced in **Java 6** as part of the **java.util.concurrent** package.
It is a thread-safe, scalable, and navigable map implementation based on a **skip list** data structure. The

map is sorted according to the natural ordering of its keys or by a specified comparator. Its non-blocking nature and guaranteed log(n) time complexity for most operations make it suitable for highly concurrent environments.

Feature	TreeMap	ConcurrentSkipListMap
Thread Safety	Not thread-safe; requires external synchronization	Thread-safe; supports concurrent acces
Synchronization Scope	Must synchronize externally	Uses fine-grained locks for efficient cor
Performance	Slower in multi-threaded environments due to explicit locking	Faster in multi-threaded environments
Ordering	Maintains natural order of keys	Maintains natural order of keys
Scalability	Limited scalability in multi-threaded	scenarios Highly scalable for conc
Use Case	Single-threaded scenarios or explicit synchronization	Multi-threaded scenarios with sorted d

```
package day4;
import java.util.TreeMap;
import java.util.concurrent.ConcurrentSkipListMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class MapAndConcurMapDemo {
public static void main(String[] args) {
long startTime=System.nanoTime();
treeMapDemo();
long endTime=System.nanoTime();
System.out.println("TreeMap time..:"+(endTime-startTime)+" ns");
long startTime2=System.nanoTime();
concurrentMapDemo();
long endTime2=System.nanoTime();
System.out.println("concurrent map time..:"+(endTime2-startTime2)+" ns");
}
public static void treeMapDemo() {
TreeMap<Integer, String> treemap=new TreeMap<>();
Thread t1=new Thread(()->{                    synchronized(treemap) {
for(int i=0;i<5;i++) {
treemap.put(i, "1 thread Value..:"+i);                    }
}
```

```
});

Thread t2=new Thread(()->{                    synchronized(treemap) {                    for(int
i=0;i<5;i++) {                    treemap.put(i, "2 thread Value..:"+i);                }
        }
        });
t1.start();t2.start();
try {t1.join();t2.join();}catch(Exception e) {}
synchronized (treemap) {        treemap.forEach((key,value)->{System.out.println(key+":"+value);});
                }
        }

public static void concurrentMapDemo() {
final ReentrantReadWriteLock mylock=new ReentrantReadWriteLock();
ConcurrentSkipListMap<Integer, String> treemap=new ConcurrentSkipListMap<>();
        Thread t1=new Thread(()->{                    mylock.writeLock().lock();
        for(int i=0;i<5;i++) {
        treemap.put(i, "1st thread Value..:"+i);
        }
                        mylock.writeLock().unlock();
        });
        Thread t2=new Thread(()->{                    mylock.writeLock().lock();
        for(int i=0;i<5;i++) {
        treemap.put(i, "2nd thread Value..:"+i);
        }
        mylock.writeLock().unlock();
});
                t1.start();t2.start();
                try {t1.join();t2.join();}catch(Exception e) {}
                mylock.readLock().lock();
                        treemap.forEach((key,value)->{System.out.println(key+":"+value);});
                mylock.readLock().unlock();
        }
}
```

JVM HotSpot Performance Tuning Tips

Configuring the JVM HotSpot properly can significantly improve Java application performance. Below are key optimization techniques:

1. Heap Memory Configuration

- Set Initial (-Xms) and Maximum (-Xmx) Heap Size
  - Avoid dynamic heap resizing by setting them to the same value.
  - Example:
    java -Xms4G -Xmx4G -jar app.jar
  - Rule of Thumb: Allocate 50-70% of available RAM (leave room for OS and other processes).
- Young Generation (-Xmn) Tuning
  - Adjust the size of the Eden + Survivor spaces.
  - Example (allocate 1GB for Young Gen):
    java -Xmn1G -jar app.jar

- Too small? Frequent minor GCs.
- Too large? Longer minor GC pauses.

2. Garbage Collection (GC) Tuning

Choose the Right GC Algorithm

G1 GC (Default in Java 9+)	Balanced throughput & latency	-XX:+UseG1GC
ZGC (Java 11+)	Ultra-low pause times (<10ms)	-XX:+UseZGC
Shenandoah (Java 12+)	Low latency (like ZGC)	-XX:+UseShenandoahGC
Parallel GC	High throughput (batch apps)	-XX:+UseParallelGC

Key GC Flags

- Max GC Pause Goal (G1 GC):
  -XX:MaxGCPauseMillis=200 Target 200ms max pause
- Adaptive Sizing (Let JVM optimize):
  -XX:+UseAdaptiveSizePolicy
- Disable Explicit GC Calls (Prevent System.gc() pauses):
  -XX:+DisableExplicitGC

3. JIT Compiler Optimizations

- Tiered Compilation (Enabled by Default)
  - Uses both C1 (Client) and C2 (Server) compilers.
  - Force C2 for long-running apps:
    -XX:-TieredCompilation -server
- Increase Inlining Threshold
  - More aggressive method inlining:
    -XX:InlineSmallCode=2000
- Optimize Code Cache
  - Increase size if JIT runs out of space:
    -XX:ReservedCodeCacheSize=256M

4. Thread & Stack Tuning

- Reduce Thread Stack Size (If using many threads):
  -Xss256k Default is 1MB (may waste memory)

- Use Biased Locking (For Legacy Apps)
  - Helps in high-contention scenarios (disabled in Java 15+):
    -XX:+UseBiasedLocking

5. Miscellaneous Optimizations

- Disable Logging for Production
  - Reduce overhead from debug logs:
    -Dlog4j2.disable.jmx=true -Dlogback.statusListenerClass=OFF
- Use Compressed OOPs (64-bit JVM)
  - Saves memory by compressing references (enabled by default):
    -XX:+UseCompressedOops
- Prefer -XX:+UseStringDeduplication (G1 GC Only)
  - Reduces memory by deduplicating strings.

6. Monitoring & Debugging Flags

- Enable GC Logging
  -Xlog:gc:file=gc.log:time:filecount=5,filesize=10M
- Heap Dump on OOM
  -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/dump.hprof
- Flight Recorder (Continuous Profiling)

-XX:StartFlightRecording=duration=60s,filename=recording.jfr

Example Optimized JVM Command

```
java -Xms8G -Xmx8G -Xmn2G \
    -XX:+UseG1GC -XX:MaxGCPauseMillis=200 \
    -XX:+UseStringDeduplication \
    -XX:+DisableExplicitGC \
    -XX:+HeapDumpOnOutOfMemoryError \
    -jar app.jar
```

Final Tips

Profile Before Tuning (Use VisualVM, JProfiler, JFR).

Test Changes (Small flag tweaks can have big impacts).

Update JVM (Newer versions like Java 17+ have better GC and optimizations).

```
package testpack;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

/* -XX:+UseParallelGC -Xmx1G -Xlog:gc*
 * Time Taken..:2632014300
 *
 * # G1 GC (Balanced)
 * java -XX:+UseG1GC -Xmx1G -Xlog:gc*
 * Time Taken..:3381564600
 *
 * # ZGC (Low latency)
 * java -XX:+UseZGC -Xmx1G -Xlog:gc*
 * Time Taken..:2939105900
 *
 * Expected Result:
    Parallel GC finishes fastest but has long pauses.
    ZGC has shortest pauses but slightly lower throughput.
.....
Tune Heap Sizes - Find optimal heap size to minimize GC frequency.
-Xms1G -Xmx1G -XX:+UseG1GC -Xlog:gc
    Time Taken..:4521367400
-Xms64M -Xmx64M -XX:+UseG1GC -Xlog:gc
      Time Taken..:3090628400
.....
Goal: Tune G1 for predictable pauses.- Tune for shorter pauses:
-XX:+UseG1GC -Xmx1G -XX:MaxGCPauseMillis=50 -Xlog:gc
Time Taken..:2845240000
https://gceasy.io - Read the log file using gceasy site
 */
/*
 * -Xms4g -Xmx4g -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -
XX:InitiatingHeapOccupancyPercent=45
 */
public class GCTest1 {
    public static void main(String[] args) {
```

```
      long st=System.nanoTime();
        long start = System.currentTimeMillis();
       // while (System.currentTimeMillis() - start < 60_000) { // 1 min
        for(int k=0;k<10;k++) {
            List<String> list = new ArrayList<>();
            for (int i = 0; i < 100_000; i++) {
                list.add(UUID.randomUUID().toString());
            }
        }
        long et=System.nanoTime();
        System.out.println("Time Taken..:"+(et-st));
    }
}
```

Ex2:

```
package walmart;

import java.util.Optional;

public class Ex6 {
      public static void main(String[] args) {
            MyObject obj=null;
//          if(obj==null) {
//                System.out.println("obj is nulll");
//          }
//          else {
//                System.out.println("its not null");
//          }
            String result=
                        Optional.ofNullable(obj)
                              .map(MyObject::getName).orElse("its null");
            System.out.println(result);

            String value=null;
            Optional<String> ov=Optional.ofNullable(value);
            ov.ifPresentOrElse((val)->{System.out.println(val);}, ()->{
                  System.out.println("null value...");
            });
      }
}
class MyObject{
      private String name;
      public MyObject(String name) {
            this.name=name;
      }
      public String getName() {
            return this.name;
      }
}
package walmart;
```

```java
import java.util.Arrays;
import java.util.List;

public class StringPerformanceDemo {
	public static void main(String[] args) {
		String s1="hello";
		//hello is stored in the string pool
		String s2="hello";
		//s2 refers the same "hello" in the pool
		System.out.println(s1==s2);

		String s3=new String("hello");
		System.out.println(s1==s3);

		System.out.println(s1.equals(s3));

		String s4=new String("hello").intern();
		//will force the string to be picked up from pool
		System.out.println(".......................");
		System.out.println(s1==s4);
		System.out.println(".......................");
		String temp1="hello world";
		String temp2=temp1;
		temp1=temp1+"new world..";
		System.out.println(temp2);

		//StringBuilder(Non Synchronized) and StringBuffer
(Synchronized)
		//StringBuffer - Performance drop
		//SYNCHRONIZED KEYWORD IS BAD FOR PERFORMANCE
		//LOCKS INTRODUCED AFTER JDK8 -USE THEM TO SOLVE

		String s="hello"+ "world";
		s=String.join("hello", "world");//more optimized than + sign

		s=String.format("the value is...%d", 10000);

		System.out.println(s);

		List<String> strs=Arrays.asList("a","b","c","d");
		strs.stream().forEach(System.out::println);

		strs.parallelStream().forEach(System.out::println);
```

```
	}
}

…….

package walmart;

import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Date;

public class DateTimeDemo {
	public static void main(String[] args) {
		long startTime=System.nanoTime();
		Date oldDate=new Date();
		SimpleDateFormat sdf=new SimpleDateFormat("YYYY-MM-dd");
		String formatdate=sdf.format(oldDate);
		System.out.println(formatdate);
		long endTime=System.nanoTime();
		System.out.println("Elapsed Time.."+(endTime-startTime));

		//New Date Time
		long startTime2=System.nanoTime();
		LocalDate newdate=LocalDate.now();
		DateTimeFormatter dtf=DateTimeFormatter.ofPattern("YYYY-MM-
dd");
		String formatdate2=newdate.format(dtf);
		System.out.println(formatdate2);
		long endTime2=System.nanoTime();
		System.out.println("Elapsed Time2.."+(endTime2-startTime2));

	}
}

…….

package walmart;

public class GcDemo {
	public static void main(String[] args) {
		System.out.println
		("Before Grand Father
Birth..:"+Runtime.getRuntime().freeMemory());
		GrandFather gf=new GrandFather();
		System.out.println
```

```
            ("After Grand Father
Birth..:"+Runtime.getRuntime().freeMemory());

            gf=null;
            System.out.println
            ("After Grand Father
Death..:"+Runtime.getRuntime().freeMemory());

            System.gc();

            System.out.println
            ("After GC..:"+Runtime.getRuntime().freeMemory());


      }
}

class GrandFather{
      private String gold="under the tree";
      String life=new String();
      public GrandFather() {
            for(int i=0;i<10000;i++) {
                  life=new String(""+i);
            }
      }
}

……
package walmart;

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicDemo {
      private static int na=0;
      private static AtomicInteger ai=new AtomicInteger(0);
      public static void main(String[] args) {
            AtomicDemo obj=new AtomicDemo();
            long startTime=System.nanoTime();
                  obj.incrementNonAtomic();
            long endTime=System.nanoTime();
            System.out.println("Non Atomic..Elapsed Time.."+(endTime-
startTime));

          startTime=System.nanoTime();
                  obj.incrementAtomic();
            endTime=System.nanoTime();
```

```
          System.out.println("Atomic..Elapsed Time.."+(endTime-
startTime));

      }
      public void incrementNonAtomic() {
            Runnable r=()->{
                  for(int i=0;i<1000;i++) {
                        synchronized(this) {
                              na++;
                        }
                  }
            };
            Thread t1=new Thread(r);
            Thread t2=new Thread(r);
            t1.start();
            t2.start();
            try {
                  t1.join();
                  t2.join();
            }catch(Exception e) {}
            System.out.println("Incremented na Value...:"+na);
      }
      public void incrementAtomic() {
            Runnable r=()->{
                  for(int i=0;i<1000;i++) {
                        ai.incrementAndGet();//automatically increments
the value
                  }
            };
            Thread t1=new Thread(r);
            Thread t2=new Thread(r);
            t1.start();
            t2.start();
            try {
                  t1.join();
                  t2.join();
            }catch(Exception e) {}
            System.out.println("Incremented ai Value...:"+ai.get());
      }
}

.....
package walmart;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReference;
```

```
class User{
	private String name;
	public User(String name) {
		this.name=name;
	}
	public String getName() {
		return this.name;
	}
}
public class AtomicDemo2 {
	private static User user=new User("non synch");

	private static AtomicReference<User> atomuser
		=new AtomicReference<User>(new User("auto sync user"));
	public static void main(String[] args) {
		AtomicDemo2 obj=new AtomicDemo2();
		long startTime=System.nanoTime();
			obj.incrementNonAtomic();
		long endTime=System.nanoTime();
		System.out.println("Non Atomic..Elapsed Time.."+(endTime-
startTime));

	    startTime=System.nanoTime();
			obj.incrementAtomic();
		 endTime=System.nanoTime();
		System.out.println("Atomic..Elapsed Time.."+(endTime-
startTime));

	}
	public void incrementNonAtomic() {
		Runnable r=()->{
			for(int i=0;i<1000;i++) {
				//synchronized(this) {
					User newuser=
						new
User(Thread.currentThread().getName()+"user...."+i);
					user=newuser;

	if(Thread.currentThread().getName().equals("t1")) {
						try{Thread.sleep(1);}catch(Exception
e) {}
					}
				//}
			}
			System.out.println(user.getName());
		};
```

```
            Thread t1=new Thread(r,"t1");
            Thread t2=new Thread(r,"t2");
            t1.start();
            t2.start();
            try {
                  t1.join();
                  t2.join();
            }catch(Exception e) {}
            System.out.println("Final Non Atomic Name
value..:"+user.getName());
      }
      public void incrementAtomic() {
            Runnable r=()->{
                  for(int i=0;i<1000;i++) {
                        User newuser=new
User(Thread.currentThread().getName()+"user.."+i);
                        atomuser.set(newuser);
                  }
            };
            Thread t1=new Thread(r);
            Thread t2=new Thread(r);
            t1.start();
            t2.start();
            try {
                  t1.join();
                  t2.join();
            }catch(Exception e) {}
            System.out.println("Incremented ai
Value...:"+atomuser.get().getName());
      }
}
```

……

```
package walmart;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import io.micrometer.core.instrument.Timer;
import io.micrometer.core.instrument.binder.jvm.JvmThreadMetrics;
import io.micrometer.prometheus.PrometheusConfig;
import io.micrometer.prometheus.PrometheusMeterRegistry;

public class MetricsDemo1 {
      PrometheusMeterRegistry registry;
```

```
	public MetricsDemo1() {
		registry = new
PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
	}
	public void met() {
	   // Register JVM Thread Metrics
	   new JvmThreadMetrics().bindTo(registry);
	   //ExecutorService
es=Executors.newSingleThreadExecutor();//this will make the first
block run first and
	    //after 5 seconds the second block is called.
	    //ExecutorService es=Executors.newCachedThreadPool();

	   ExecutorService es=Executors.newFixedThreadPool(2);
	    es.execute(()->{
		System.out.println("first block...");
		try {Thread.sleep(5000);}catch(Exception e) {}
	    });

	    es.execute(()->{
		System.out.println("second block....");
		try {}catch(Exception e) {}
	    });
	    int totalThreads =
Thread.getAllStackTraces().keySet().size();
	   System.out.println("Total number of threads: " +
totalThreads);

	  // new JvmThreadMetrics().bindTo(registry);

	   System.out.println(registry.scrape());
	   es.shutdown();
	 }
	public static void main(String[] args) {
		MetricsDemo1 obj=new MetricsDemo1();
		obj.met();
	}

}


……..
package walmart;

import java.util.TreeMap;
import java.util.concurrent.ConcurrentSkipListMap;
import java.util.concurrent.locks.ReentrantReadWriteLock;
```

```
import io.micrometer.core.instrument.Timer;
import io.micrometer.prometheus.PrometheusConfig;
import io.micrometer.prometheus.PrometheusMeterRegistry;

public class MapAndConcurrentDemo {
	static PrometheusMeterRegistry registry;
	static{
		//registry = new
PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
	}
	public static void main(String[] args) {
		registry = new
PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
		Timer timer=registry.timer("application.process.time");
		long st=System.nanoTime();
		timer.record(()->{
			oldTreeMapDemo();
		});
		System.out.println(registry.scrape());

		long et=System.nanoTime();
		System.out.println("Time Dif1..:"+(et-st));
		registry = new
PrometheusMeterRegistry(PrometheusConfig.DEFAULT);
		st=System.nanoTime();
		Timer timer2=registry.timer("application.process.time");
		timer2.record(()->{
			newTreeMapDemo();
		});
		System.out.println(registry.scrape());
		et=System.nanoTime();
		System.out.println("Time Dif2..:"+(et-st));
	}
	public static void oldTreeMapDemo() {
		TreeMap<Integer,String> treemap=new TreeMap<>();
		Thread t1=new Thread(()->{
			synchronized(treemap) {
				for(int i=0;i<5;i++) {
					treemap.put(i, "1 thread value..:"+i);
				}
			}
		});
		Thread t2=new Thread(()->{
			synchronized(treemap) {
				for(int i=0;i<5;i++) {
```

```
                              treemap.put(i, "1 thread value..:"+i);
                        }
                  }
            });
            t1.start();
            t2.start();
            try {t1.join();t2.join();}catch(Exception e) {}
            synchronized(treemap) {
                  treemap.forEach((key,value)->{
                        System.out.println(key+":"+value);
                  });
            }
      }
      public static void newTreeMapDemo() {
            final ReentrantReadWriteLock mylock=new
ReentrantReadWriteLock();

            ConcurrentSkipListMap<Integer, String> treemap=new
ConcurrentSkipListMap<>();
            Thread t1=new Thread(()->{
                  mylock.writeLock().lock();
                  for(int i=0;i<5;i++) {
                        treemap.put(i, "1st thread value..:"+i);
                  }
                  mylock.writeLock().unlock();
            });
            Thread t2=new Thread(()->{
                  mylock.writeLock().lock();
                  for(int i=0;i<5;i++) {
                        treemap.put(i, "2nd thread value..:"+i);
                  }
                  mylock.writeLock().unlock();
            });
            t1.start();t2.start();
            try {t1.join();t2.join();}catch(Exception e) {}
            mylock.readLock().lock();
            treemap.forEach((k,v)->{
                  System.out.println(k+":"+v);
            });
            mylock.readLock().unlock();

      }

}
```

.....

```
package walmart;
```

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Phaser;

public class BarrierDemo {
	public static void main(String[] args) {
		long st=System.nanoTime();
			cyclicDemo();
		long et=System.nanoTime();
		System.out.println("Cyclic Time Dif..:"+(et-st));

		st=System.nanoTime();
			phaserDemo();
		et=System.nanoTime();
		System.out.println("Phaser Time Dif..:"+(et-st));
	}

	public static void cyclicDemo() {
		CyclicBarrier barrier=new CyclicBarrier(3,()->{
			System.out.println("All employees reached....");
		});
		for(int i=0;i<3;i++) {
			new Thread(()->{
				try {

	System.out.println(Thread.currentThread().getName()+" has
arrived..");
					barrier.await();

	System.out.println(Thread.currentThread().getName()+" got into
the bus..");
				}catch(Exception e) {
					e.printStackTrace();
				}
			}).start();
		}
	}
	public static void phaserDemo() {
		Phaser barrier=new Phaser(3);
		for(int i=0;i<3;i++) {
			new Thread(()->{
				try {

	System.out.println(Thread.currentThread().getName()+" has
arrived..");
					barrier.arriveAndAwaitAdvance();
```

```
	System.out.println(Thread.currentThread().getName()+" got into
the bus..");
                        }catch(Exception e) {
                              e.printStackTrace();
                        }
                  }).start();
            }
      }
}
……..
package testpack;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
/*
 * <dependency>
<groupId>com.zaxxer</groupId>
<artifactId>HikariCP</artifactId>
<version>5.0.1</version> <!-- Use the latest version -->
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>8.0.33</version>
</dependency>

 */
public class JDBCDemo {
      private static String url="jdbc:mysql://localhost:3306/testdb";
      private static String user="root";
      private static String password="root";
public static void main(String[] args) {
            long startTime=System.nanoTime();
            testConnection();
            long endTime=System.nanoTime();
            System.out.println("Ordinary above jdk 6
connection..:"+(endTime-startTime)+" ns");
            long startTime2=System.nanoTime();
            testHikariConnection();
            long endTime2=System.nanoTime();
            System.out.println("Hikari connection..:"+(endTime2-
startTime2)+" ns");
```

```
}
public static void testConnection() {
//no need to load the driver from jdk6
//Class.forName("com.mysql.cj.jdbc.Driver"); - not needed...
//try with resources - this will call the con.close automatically -
through AutoClose feature
//both the points - not loading the driver and using autoclose will
enhance performance for sure
      try(Connection
con=DriverManager.getConnection(url,user,password)){
                  Statement stmt=con.createStatement();
                  ResultSet rs=stmt.executeQuery("select * from users");
                  while(rs.next()) {
                        System.out.println("User
name..:"+rs.getString("username"));
                  }
            }catch(Exception e) {
            e.printStackTrace();
            }
}
public static void testHikariConnection() {
            HikariConfig config=new HikariConfig();

      config.setJdbcUrl(url);config.setUsername(user);config.setPasswor
d(password);
            config.setMaximumPoolSize(10);//maximum connections in the
pool
            HikariDataSource dataSource=new HikariDataSource(config);
            try(Connection con=dataSource.getConnection()){
                  Statement stmt=con.createStatement();
                  ResultSet rs=stmt.executeQuery("select * from users");
                  while(rs.next()) {
                  System.out.println("User
name..:"+rs.getString("username"));
                  }
            }catch(Exception e) {
                  e.printStackTrace();
                  }
                  }
}
……….
package testpack;
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
```

```
public class FileNIODemo {
public static void main(String[] args) {
            String fileName="pom.xml";
            //Warm up
            for(int i=0;i<1000;i++) {
                  new File(fileName).exists();
                  Files.exists(Paths.get(fileName));
            }
                  long startTime=System.nanoTime();
                  File file=new File(fileName);
                  if(file.exists()) {
                        System.out.println("file exists.....");
                  }
                  else {
                        System.out.println("file does not exists....");
                  }
            long endTime=System.nanoTime();
            System.out.println("Normal IO..:"+(endTime-startTime)+"
ns");
//NIO
            long startTime2=System.nanoTime();
            Path path=Paths.get(fileName);
            if(Files.exists(path)) {
                  System.out.println("file exists....");
            }
            else {
                  System.out.println("file does not exists.....");
            }
            long endTime2=System.nanoTime();
                  System.out.println("NIO...:"+(endTime2-startTime2)+"
ns");
                  System.out.println(path.toAbsolutePath());
            }
}

…..
package testpack;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URL;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
```

```
import java.net.http.HttpResponse;

public class HttpTest {
public static void main(String[] args)throws Exception {
            long startTime=System.nanoTime();
                  useOldHttpCall();
            long endTime=System.nanoTime();
            System.out.println("old http call...:"+(endTime-startTime)+"
ns");
            long startTime2=System.nanoTime();
                  useLatestHttpCall();
            long endTime2=System.nanoTime();
            System.out.println("new http call...:"+(endTime2-
startTime2)+" ns");
}
public static void useLatestHttpCall()throws Exception {
            for(int i=0;i<100;i++) {
            new Thread(()->{
            try {
            HttpClient client=HttpClient.newHttpClient();
            HttpRequest request=HttpRequest.

      newBuilder().uri(URI.create("https://jsonplaceholder.typicode.com
/posts/1")).build();
            HttpResponse<String> response=client.send(request,
HttpResponse.BodyHandlers.ofString());
            //System.out.println(response.body());
            }catch(Exception e) {e.printStackTrace();}
            }).start();
      }
}

public static void useOldHttpCall() throws Exception{
      for(int i=0;i<100;i++) {
      new Thread(()->{
      try {
            URL url=new
URL("https://jsonplaceholder.typicode.com/posts/1");
            HttpURLConnection
connection=(HttpURLConnection)url.openConnection();
            connection.setRequestMethod("GET");
            connection.setRequestProperty("Accept", "application/json");
            try(BufferedReader reader=new BufferedReader(new
InputStreamReader(connection.getInputStream()))){
            StringBuilder response=new StringBuilder();
            String line;
```

```
            while((line=reader.readLine())!=null) {
            response.append(line);
      }
      //System.out.println("Response body...:"+response.toString());
      }
      }catch(Exception e) {e.printStackTrace();}
      }).start();
      }
      }
}
```

.....

JMETER

If you're not getting the data in the **Body** tab of the **"View Results Tree"** listener in JMeter after executing an HTTP request, here are key things to check and troubleshoot:

1. Enable Response Body Recording

- In **View Results Tree**, make sure the **"Text"**, **"HTML"**, or **"Response Data"** tab is selected.
- If it shows **"No Data"** or is empty:
  - Go to **JMeter Menu → Options → Log Level → DEBUG**, then rerun the test to see more logs.

2. Check HTTP Request Configuration

Double-check these settings:

- **Server Name or IP**: No `http://` — only domain, e.g., `jsonplaceholder.typicode.com`
- **Protocol**: Must be `http` or `https`
- **Path**: Make sure it's valid, e.g., `/posts`
- **Method**: Use correct one — `GET`, `POST`, etc.

3. Add HTTP Header Manager (for APIs)

Some APIs won't respond unless headers are set correctly:

- Add HTTP Header Manager under the HTTP Request:
- `Content-Type: application/json`
- `Accept: application/json`

4. Check Response Code

In **View Results Tree → Sampler Result**, look for:

- **Response code**: `200` means OK.
- If it's `403`, `401`, `500`, or `404`, there's a config issue or server problem.

5. Increase Response Size Display Limit

JMeter truncates large responses:

- Go to: **jmeter.properties** file
- Search and set:
- `view.results.tree.max_size=0`

  (This removes the size limit. Restart JMeter after saving.)

6. Test the URL Outside JMeter

Paste your complete URL (e.g., `https://jsonplaceholder.typicode.com/posts`) into a browser or use `curl`:

```
curl https://jsonplaceholder.typicode.com/posts
```

If this doesn't return data, the problem is with the URL or endpoint.

7. Try a Working Public API

As a test, use this configuration:

- Protocol: `https`
- Server: `jsonplaceholder.typicode.com`
- Path: `/posts`
- Method: `GET`

You should get JSON response in the body.

JDBC Performance Improvement Tips

Here are 20 actionable techniques to optimize JDBC performance in Java applications:

1. Connection Pooling

Problem: Creating new connections is expensive (~100ms per connection).
Solution: Use a connection pool like HikariCP (fastest), Tomcat JDBC, or c3p0.

```
// HikariCP Configuration (Recommended)
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
config.setUsername("user");
config.setPassword("pass");
config.setMaximumPoolSize(20);  // Optimal for most apps
HikariDataSource ds = new HikariDataSource(config);
```

2. Batch Processing

Problem: Executing individual INSERT/UPDATE statements is slow.
Solution: Use addBatch() and executeBatch().

```
try (Connection con = ds.getConnection();
    PreparedStatement ps = con.prepareStatement("INSERT INTO users
VALUES (?, ?)")) {

   for (User user : users) {
      ps.setInt(1, user.getId());
      ps.setString(2, user.getName());
      ps.addBatch();  // Add to batch
   }
   ps.executeBatch();  // Execute all at once
}
```

3. Fetch Size Optimization

Problem: Default fetch size (e.g., 10 rows) causes multiple roundtrips.
Solution: Increase fetch size for large queries.

```
Statement stmt = con.createStatement();
stmt.setFetchSize(1000);  // Fetch 1000 rows per trip
ResultSet rs = stmt.executeQuery("SELECT  FROM large_table");
```

4. Use Prepared Statements

Problem: SQL parsing overhead for repeated queries.
Solution: Cache and reuse PreparedStatement.

```
// Reuse across requests
PreparedStatement ps = con.prepareStatement("SELECT  FROM users
WHERE id = ?");
ps.setInt(1, userId);
ResultSet rs = ps.executeQuery();
```

5. Disable Auto-Commit

Problem: Auto-commit mode creates overhead for transactions.
Solution: Disable for batch operations.

```
con.setAutoCommit(false);
// Perform multiple operations
con.commit();  // Commit once
```

6. ResultSet Optimization

Problem: Unnecessary memory usage for large results.
Solution: Use TYPE_FORWARD_ONLY + CONCUR_READ_ONLY.

```
Statement stmt = con.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_READ_ONLY
);
```

7. Connection Timeout Tuning

Problem: Stale connections waste resources.
Solution: Configure timeouts.

```
// HikariCP example
config.setConnectionTimeout(30000);  // 30s
config.setIdleTimeout(600000);       // 10m
config.setMaxLifetime(1800000);      // 30m
```

8. Indexing & Query Optimization

Problem: Slow queries due to missing indexes.
Solution:
- Add indexes on frequently queried columns.
- Use EXPLAIN to analyze query plans.

9. Limit Data Retrieval

Problem: Fetching unnecessary columns/rows.
Solution: Be specific in SELECT.

```
SELECT  FROM users;
```

Good

```
SELECT id, name FROM users WHERE status = 'ACTIVE' LIMIT 100;
```

10. Use Try-With-Resources

Problem: Resource leaks from unclosed connections.
Solution: Auto-close with try-with-resources.

```
try (Connection con = ds.getConnection();
     PreparedStatement ps = con.prepareStatement("...");
     ResultSet rs = ps.executeQuery()) {
    // Process results
}
```

11. Disable Debug Logging
Problem: JDBC driver logging slows performance.
Solution: Disable in production.

properties
For MySQL
logging.level.jdbc=OFF

12. Connection Validation
Problem: Dead connections cause failures.
Solution: Validate connections before use.

```
config.setConnectionTestQuery("SELECT 1");  // HikariCP
```

13. Use Stored Procedures
Problem: Network overhead for complex logic.
Solution: Move logic to database.

```
CallableStatement cs = con.prepareCall("{call get_user_details(?)}");
cs.setInt(1, userId);
cs.execute();
```

14. Avoid SELECT FOR UPDATE
Problem: Unnecessary row locking.
Solution: Use optimistic locking or smaller transactions.

15. Metadata Caching
Problem: Repeated DatabaseMetaData calls are slow.
Solution: Cache metadata locally.

```
// Cache table schema at startup
DatabaseMetaData meta = con.getMetaData();
ResultSet tables = meta.getTables(null, null, "%", null);
```

16. Network Tuning
Problem: Slow network impacts JDBC.
Solution:
- Place app server close to DB.
- Use socketTimeout:
// MySQL example
jdbc:mysql://localhost:3306/db?socketTimeout=30000

17. Use Statement.setFetchDirection()
Problem: Inefficient result traversal.
Solution: Optimize fetch direction.

stmt.setFetchDirection(ResultSet.FETCH_FORWARD);

18. Connection Pool Sizing
Formula:

Pool size = (Core count 2) + Effective disk spindles

- For 16-core CPU + SSD: ~20-30 connections.
19. Upgrade JDBC Driver
Problem: Old drivers lack optimizations.
Solution: Use the latest version.

```
<!-- MySQL Connector -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

Summary Checklist
Connections	Use HikariCP, disable auto-commit
Queries	Batch processing, fetch size, prepared statements
Network	Timeouts, keepalive, driver upgrade
Monitoring	JMX, logging control

Key Rule:
"Fetch only what you need, reuse what you can, and batch the rest."

Sample Exercise

```
package performancepack.database;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
/*
 * Statement stmt = connection.createStatement();
stmt.setQueryTimeout(30); // Timeout after 30 seconds
 */
public class BlobPerformanceDemo {
	public static void main(String[] args) {
		//init();
		blobtest();
	}
	public static void init() {
		 String createTableSQL = "CREATE TABLE IF NOT EXISTS
files (" +
		"id INT AUTO_INCREMENT PRIMARY KEY, " +
		"file_name VARCHAR(255), " +
		"blob_column BLOB)";
		 String insertSQL = "INSERT INTO files (file_name,
blob_column) VALUES (?, ?)";
		 File fileToInsert = new File("pom.xml"); // Make sure this
file exists
		HikariConfig config = new HikariConfig();
	config.setJdbcUrl("jdbc:mysql://localhost:3306/testdb");
	config.setUsername("root");
	config.setPassword("root");
	config.setMaximumPoolSize(10); // Maximum connections in
the pool
// Create the DataSource
	HikariDataSource dataSource = new HikariDataSource(config);
	try (Connection connection = dataSource.getConnection()) {
	 try (Statement stmt = connection.createStatement()) {
```

```
            stmt.execute(createTableSQL);
            System.out.println("Table 'files' created successfully.");
        }
        // 2. Insert a BLOB into the table
        try (PreparedStatement ps =
connection.prepareStatement(insertSQL)) {
            ps.setString(1, fileToInsert.getName());
            try (FileInputStream fileInputStream = new
FileInputStream(fileToInsert)) {
                ps.setBinaryStream(2, fileInputStream, (int)
fileToInsert.length());
                ps.executeUpdate();
                System.out.println("BLOB data inserted successfully.");
            }
        }
    }catch(Exception e) {
    e.printStackTrace();
    }
    }
    public static void blobtest() {
          HikariConfig config = new HikariConfig();
    config.setJdbcUrl("jdbc:mysql://localhost:3306/testdb");
    config.setUsername("root");
    config.setPassword("root");
    config.setMaximumPoolSize(10); // Maximum connections in
the pool
    // Create the DataSource
    HikariDataSource dataSource = new HikariDataSource(config);

          long startTime=System.nanoTime();
          //try-with-resources for JDBC (JDK 7) - Improves
Performance
          try (Connection connection = dataSource.getConnection())
{
                Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT blob_column
FROM files WHERE id = 1");
if (rs.next()) {
          byte[] data = rs.getBytes("blob_column");
          System.out.println("Data length: " + data.length);
        }
          }catch(Exception e) {
```

```
                    e.printStackTrace();
                }
                long endTime = System.nanoTime();
        System.out.println("with normal blob...: " + (endTime -
startTime) + " ns");

        long startTime2=System.nanoTime();
                //try-with-resources for JDBC (JDK 7) - Improves
Performance
                try (Connection connection = dataSource.getConnection())
{
                    Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT blob_column
FROM files WHERE id = 1");
            if (rs.next()) {
                try (InputStream blobStream =
rs.getBinaryStream("blob_column");
                     FileOutputStream fileOut = new
FileOutputStream("output_file.xml")) {
                    byte[] buffer = new byte[1024];
                    int bytesRead;
                    // Read the data in chunks and write to the file
                    while ((bytesRead = blobStream.read(buffer)) != -1) {
                        fileOut.write(buffer, 0, bytesRead);
                    }
System.out.println("File saved successfully!");
                }
            }
                }catch(Exception e) {
                    e.printStackTrace();
                }
                long endTime2 = System.nanoTime();
        System.out.println("with Performance blob...: " + (endTime2 -
startTime2) + " ns");
        }
}
```