Java Application Performance and Memory Management - Notes

1. Introduction to JVM Performance

a. JVM Internals

- **JVM Components:**
  - **Class Loader** – Loads Java classes into memory.
  - **Runtime Data Areas** – Includes Heap, Stack, and Metaspace.
  - **Execution Engine** – Includes JIT Compiler and Interpreter.
  - **Garbage Collector (GC)** – Manages automatic memory allocation/deallocation.
- **JVM Memory Areas:**
  - **Heap Memory:** Stores objects, divided into Young & Old generations.
  - **Stack Memory:** Stores method execution details and local variables.
  - **Metaspace:** Stores class metadata (replaces PermGen).
  - **Native Memory:** Used for OS-level interactions.

2. Java Performance Optimization

a. Writing Efficient Code

- Avoid **redundant object creation** (reuse objects where possible).
- Use **StringBuilder** instead of String for concatenation inside loops.
- **Prefer primitive types** (int, double) over wrapper types (Integer, Double).
- Use **efficient collections**:
  - ArrayList over LinkedList (for fast random access).
  - HashMap over TreeMap (for faster key-based lookups).

b. Optimizing Multi-threaded Applications

- Use **Thread Pools (ExecutorService)** instead of creating new threads.
- Prefer **Concurrent Collections** (ConcurrentHashMap, CopyOnWriteArrayList).
- Avoid unnecessary synchronization (synchronized, locks).
- Use **CompletableFuture** for asynchronous programming.

c. I/O Performance Tuning

- Use **BufferedReader & BufferedWriter** for reading/writing files.
- Use **Memory-mapped files (MappedByteBuffer)** for large file handling.
- Prefer **NIO (java.nio)** over standard I/O for non-blocking operations.

3. Java Garbage Collection (GC) & Memory Management

a. Garbage Collection Basics

- **Garbage Collection (GC) Mechanisms:**
  - **Minor GC (Young Gen):** Cleans short-lived objects.
  - **Major GC (Old Gen):** Cleans long-lived objects.
  - **Full GC:** Cleans entire Heap, causes application pauses.
- **Types of Garbage Collectors:**
  - **Serial GC** – Best for small apps, single-threaded.
  - **Parallel GC** – Multi-threaded, high throughput.
  - **G1 GC (Garbage First GC)** – Balances throughput and latency, recommended for large heaps.
  - **ZGC, Shenandoah GC** – Ultra-low-latency, for large-scale applications.

b. GC Tuning & JVM Options

- **Heap Sizing:**
  - -Xms512m -Xmx4g → Set initial and max heap size.
  - -XX:NewRatio=2 → Adjust ratio of Young to Old Gen.
- **Choosing a Garbage Collector:**
  - -XX:+UseG1GC → Use G1 GC.
  - -XX:+UseParallelGC → Use Parallel GC.
  - -XX:+UseZGC → Use ZGC (low-latency).
- **Logging GC Activity:**
  - -Xlog:gc* → Enable GC logging.

4. JVM Monitoring & Profiling Tools

a. JVM Built-in Tools

- jstat – Monitor JVM statistics (jstat -gc <pid>).
- jmap – Heap dump analysis (jmap -dump:format=b,file=heap.bin <pid>).
- jstack – View thread dumps (jstack <pid>).
- jcmd – Run diagnostic commands (jcmd <pid> GC.run).

b. Profiling & Performance Tools

- **JVisualVM** – Real-time monitoring of memory & CPU usage.
- **JConsole** – Lightweight monitoring tool for JVM.
- **Java Mission Control (JMC)** – Advanced profiling and diagnostics.
- **Eclipse MAT (Memory Analyzer Tool)** – Analyzes heap dumps.
- **YourKit, JProfiler** – Commercial profiling tools.

5. Memory Leak Detection & Prevention

a. Common Causes of Memory Leaks

- **Unclosed resources** (FileReader, Socket, JDBC Connections).
- **Static references holding objects** (static List<Object> leaks).
- **ThreadLocal variables** not cleared properly.
- **Improper Caching Strategy** (storing excessive objects in a Map).

b. Detecting Memory Leaks

- **Heap Dump Analysis** (jmap -dump:live,format=b,file=heap.bin <pid>).
- **Using Eclipse MAT** to detect memory leaks.
- **Profiling Tools (YourKit, JProfiler)** to identify excessive memory usage.

c. Best Practices to Avoid Memory Leaks

- Use **try-with-resources** (try (BufferedReader br = new BufferedReader(...)) {}).
- Clear **ThreadLocal** values manually (threadLocal.remove()).
- Use **WeakReference / SoftReference** for caching.

6. Performance Best Practices

- **Use parallel streams** (list.parallelStream().map(...)).
- **Optimize SQL Queries** (use indexing, avoid SELECT *).
- **Reduce synchronization overhead** in multi-threaded apps.
- **Use object pooling** for expensive objects (JDBC connections, Thread pools).
- **Cache expensive calculations** using ConcurrentHashMap, Guava Cache.

1. JVM Memory Structure & Garbage Collection (GC)

JVM Memory Areas

- **Heap Memory (Managed by GC)**
  - **Young Generation (Eden, Survivor Spaces S1 & S2)** – Short-lived objects.
  - **Old Generation (Tenured Space)** – Long-lived objects.
  - **Metaspace** – Stores class metadata (replaces PermGen).
- **Stack Memory**
  - Stores method execution details, local variables.
  - Each thread has its own stack.
- **Native Memory**
  - Used by **JNI (Java Native Interface)**, threads, and off-heap storage.

Garbage Collection (GC) Process

1. **Minor GC (Young Gen)**

    a. Occurs frequently, moves surviving objects to Survivor space.
    b. Objects that survive multiple Minor GCs move to the Old Gen.
2. **Major GC (Old Gen)**
    a. Less frequent but more time-consuming.
    b. If the Old Gen is full, a **Full GC** occurs, causing application pause.
3. **Full GC**
    a. Cleans both Young & Old Gen.
    b. Can cause performance bottlenecks if not optimized.

Types of Garbage Collectors

GC Type	Best For	Characteristics
Serial GC	Small apps	Single-threaded, good for small heaps.
Parallel GC	Multi-threaded apps	Uses multiple threads for Young Gen GC.
G1 GC	Large-scale apps	Balances throughput and latency, divides heap into regions.
ZGC	Low-latency apps	Scales up to **TBs of memory**, minimal pause times.
Shenandoah GC	Real-time apps	Low-latency like ZGC, pauses <10ms even for large heaps.

2. GC Tuning & JVM Options

Setting Heap Size

- -Xms512m -Xmx4g → Min & Max heap size.
- -XX:NewRatio=2 → New Gen to Old Gen ratio.

Choosing a Garbage Collector

- -XX:+UseG1GC → Recommended for large applications.
- -XX:+UseParallelGC → High throughput, multi-threaded.
- -XX:+UseZGC → For ultra-low-latency applications.

Tuning GC Performance

- -XX:MaxGCPauseMillis=200 → GC pause target in ms.
- -XX:+UseStringDeduplication → Reduces memory by reusing identical strings.

Enabling GC Logs

- -Xlog:gc* → Enables detailed GC logs.

3. Performance Optimization

Efficient Code Practices

- **Use StringBuilder** instead of + for concatenation in loops.
- **Avoid unnecessary object creation**, reuse existing objects.

- **Use primitive types** (int, double) over wrappers (Integer, Double).
- **Use efficient collections**:
    - ArrayList > LinkedList (better cache locality).
    - HashMap > TreeMap (faster lookups).

Optimizing Multi-threading

- **Use thread pools (ExecutorService)** instead of manually managing threads.
- **Prefer Concurrent Collections** (ConcurrentHashMap, CopyOnWriteArrayList).
- **Minimize synchronization overhead** (synchronized, locks).
- **Use CompletableFuture** for non-blocking async operations.

Optimizing I/O Performance

- **Use Buffered Streams** (BufferedReader, BufferedWriter) for reading/writing files.
- **Use memory-mapped files (MappedByteBuffer)** for fast file access.
- **Use NIO (java.nio)** for non-blocking I/O.

4. Profiling & Monitoring Java Applications

JVM Monitoring Tools

Tool	Usage
JVisualVM	Real-time JVM monitoring, heap analysis.
JConsole	Lightweight JVM monitoring, threads, CPU, memory usage.
Java Mission Control (JMC)	Advanced profiling for production apps.
Eclipse MAT	Deep memory analysis for detecting leaks.
YourKit, JProfiler	Commercial tools for deep profiling.

Command-line Tools

Command	Description
jstat -gc <pid>	Monitor GC activity.
jmap -dump:format=b,file=heap.bin <pid>	Dump heap memory for analysis.
jstack <pid>	Take thread dump.
jcmd <pid> GC.run	Force GC manually.

5. Memory Leak Detection & Prevention

Common Causes of Memory Leaks

- **Unclosed resources** (FileReader, Socket, JDBC Connection).
- **Static references holding objects** (static List<Object> leaks).
- **ThreadLocal variables** not cleared properly.

- **Improper caching strategy** (storing excessive objects in memory).

Detecting Memory Leaks

- **Heap Dump Analysis** (jmap -dump:format=b,file=heap.bin <pid>).
- **Using Eclipse MAT** to detect memory leaks.
- **Profiling Tools (YourKit, JProfiler)** for live memory analysis.

Preventing Memory Leaks

- **Use try-with-resources** (try (BufferedReader br = new BufferedReader(...)) {}).
- **Clear ThreadLocal** values manually (threadLocal.remove()).
- **Use WeakReference / SoftReference** for caching.

6. Concurrency & Multithreading Optimization

Best Practices

- **Use thread-safe collections** (ConcurrentHashMap, CopyOnWriteArrayList).
- **Minimize locks** to reduce contention.
- **Prefer ReadWriteLock over synchronized** when reads dominate.
- **Use CompletableFuture** for async programming.

Common Threading Issues

Issue	Solution
Thread contention	Reduce shared resource access.
Deadlocks	Use timeout-based locks (tryLock).
Race conditions	Use Atomic variables (AtomicInteger).
Thread starvation	Use fair locking (ReentrantLock(true)).

7. Caching Strategies for Performance

Cache Type	Use Case
In-memory caching (e.g., Guava Cache, Caffeine)	Fast lookups within JVM.
Distributed caching (e.g., Redis, Memcached)	Shared caching for microservices.
SoftReference caching	Allows GC to reclaim memory when needed.

Best Practices for Caching

- **Eviction Strategies**: LRU (Least Recently Used), TTL (Time To Live).
- **Use ConcurrentHashMap** for in-memory caching.
- **Use Guava Cache or Caffeine** for efficient caching.

Conclusion

- **Optimize GC settings** based on application needs.
- **Monitor JVM performance** using built-in tools (jstat, jmap, jstack).
- **Use efficient collections & avoid memory leaks**.
- **Leverage multithreading best practices** to avoid contention & deadlocks.
- **Profile applications regularly** using tools like JVisualVM & YourKit.

Java Application Performance-

Performance consideration:

Memory constraints

Application speed

Aspects:

Design time

Runtime

Vendors:

Oracle jdk

Open jdk

Optimize -

Code/program - which runs 1000 times in a day
Rather than
Code/program - which runs 1 time in a day
Logically

Micro bench marking -
Measuring the performance of the code on small pieces of code.
Not always an ideal choice.

In Java, **Logging** is an API that provides the ability to trace out the errors of the applications.

Need for Logging

- It provides the complete tracing information of the application.
- It records the critical failure if any occur in an application.

Logging Formatters or Layouts

- SimpleFormatter - It generates a text message that has general information.
- XMLFormatter - The XMLFormatter generates the log message in XML format.

Java Logging Levels

S. No.	Standard Log Levels	Level	Value	Used For
1		FINEST	300	Specialized Developer Information
2		FINER	400	Detailed Developer Information
3		FINE	500	General Developer Information
4		CONFIG	700	Configuration Information
5		INFO	800	General Information
6		WARNING	900	Potential Problem
7		SEVERE	1000	Represents serious failure
8	**Special Log Levels**	OFF	Integer.MAX_VALUE	Turns off the logging
9		ALL	Integer.MIN_VALUE	Captures everything

Method 1:

1. logger.log (Level.INFO, "Display message");

Method 2:

2. logger.info ("Display message");

Logging Frameworks

We can also use the logging framework to make the logging concept easy. There is the following popular logging framework used for logging:

- Log4j: Apache Log4j is an open-source Java-based logging utility.
- SLF4J: It stands for Simple Logging Facade for Java (SLF4J). It is an abstract layer for multiple logging frameworks such as Log4j, Logback, and java.util.logging.
- Logback: It is an open-source project designed as a successor to Log4j version 1 before Log4j version 2 was released.
- Tinylog (tinylog): It is a light weighted and open-source logger.

Example-

```
import java.io.IOException;
```

```
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.logging.*;

class DemoLogger {
private final static Logger LOGGER =
Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
        public void makeSomeLog()
{
LOGGER.log(Level.INFO, "My first Log Message");
}
}

public class GfG {
public static void main (String [] args)
{
DemoLogger obj = new DemoLogger();
obj.makeSomeLog();
                LogManager lgmngr = LogManager.getLogManager();
Logger log = lgmngr.getLogger(Logger.GLOBAL_LOGGER_NAME);
log.log(Level.INFO, "This is a log message");
}
}
```