Week-04

01. Understand JVM Memory Structure.

The Java virtual machine divides the memory into

1. Method Area
2. Stack
3. Heap Memory
4. Native Method Stack

* Method Area :- Method Area is a part of the heap memory which is shared among all the threads. It creates when the JVM starts up. It is used to store data-class structure, superclass, name, interface, name, and constructor. The JVM stores the following kinds of information in the method area :

- A fully qualified name of a type
- The type's modifiers
- Type's direct superclass name
- A structure list of the fully qualified names of super interfaces.

* Stack Memory :- The stack memory allocation in java is used for static memory and thread execution. The values contained in their memory are temporary and limited to specific methods as they keep getting referenced in Last-In-First-out fashion.

As soon as the memory is called and a new block gets created in the stack memory, the stack memory then holds primitive values and references until the method lasts. After its ending, the block is flushed and is available for a new process to take place

* Java Heap Space :- Mainly used by java runtime. Java Heap space comes into play every time an object is created and allocated in it. The discrete function, like Garbage collection, keeps flushing the memory used by the previous objects that hold no reference. For an object created in the Heap Space can have free access across the application.

We can break this memory model down into smaller parts, called generations, which are :

01) Young Generation :- This is where all new objects are allocated and aged. A minor Garbage collection occurs when this fills up.

02.) Old or Tenured Generation :- This is where long surviv-ing objects are allocated stored. When objects are stored in the young Generation, a threshold for the object's age is set, and when that threshold is reached, the object is moved to the old generation.

02.) Permanent Generation :- This consists of JVM metadata for the runtime classes and application methods.

* Native Method Stack :- It is also known as C stack. It is a stack for native code written in a langu-age other than Java. Java Native Interface (JNI) calls the native stack. The performance of the native stack depends on the OS

02. Understand and differentiate between concurrent mark and sweep with Parallel mark and sweep.

Concurrent Mark And Sweep	Parallel Mark And Sweep
* CMS performs garbage collection concurrently with the application threads, aiming to minimize pause times by running some garbage collection tasks concurrently with the application. It achieves this by executing the marking phase concurrently with the application threads.	* similar to CMS, PMS also conducts garbage collection threads, but it further utilizes multiple threads to parallelize the garbage collection tasks. This parallelism enhances overall garbage collection throughput by distributing the workload across multiple threads.
* CMS consists of two main phases: marking and sweeping. During the marking phase, reachable objects are identified concurrently with the application. After marking, the sweeping phase reclaims memory occupied by garbage objects.	* PMS also follows the mark and sweep approach. It begins by marking reachable objects concurrently with the applications, similar to CMS. However, the sweeping phase may involve parallel execution across multiple threads, enabling faster reclamation of memory.
* CMS prioritizes minimizing pause times to reduce the impact on application responsiveness. It achieves this by running the marking phase concurrently with the appli-	* PMS aims to maximize overall garbage collection throughput by leveraging parallelism. While it may not achieve the same low pause times as CMS, it excels in

-cation, thus reducing the duration of stop-the-world pauses.	scenarios where maximizing overall throughput is crucial, such as batch processing or high-throughput systems.
* CMS is well-suited by latency sensitive applica-tions where minimizing pause time is critical. Its commonly used in interact-ive applications, web servers and real time system.	* PMS is preferred for appli-cation initiates its parallel phases at predefined points in the garbage collection cycle or when certain conditions are met.
* Generally optimized for applications with a low pause time requirement, sacrificing some through-put.	* Emphasizes throughput by utilizing more system resour-ces, potentially resulting in longer pause times but higher overall application throughput.
* consumes fewer CPU resour-ces during normal applic-ation execution due to concurrent marking.	* PMS can utilize more CPU resources during both marking and sweeping phases, potentially impacting application performance.
* Tends to have a smaller memory footprint because it doesn't require as much memory for parallel processing.	* may require more memory due to the overhead of managing multiple threads and larger data structures.

03. What is the difference Dynamic and Static Garbage collection modes.

Dynamic Garabage collection Modes	Static Garabage collection Modes
* Memory allocation and deallocation are handled automatically by the runtime environment.	* Memory allocation and deallocation are explicity controlled by the programmer.
* Memory management is automatic, reducing the need for manual inter- -vention.	* Memory management requires manual interv- -ention from the programmer
* Utilizes garbage collection algorithms like mark-and-sweep or generational collection.	* Relies on manual memory allocation and deallocati- -on by the programmer.
* Less error-prone as memory management is automated, reducing the risk of memory leaks or dangling pointers.	* More error-prone as programmers need to ensure proper memory allocation and deallocation to avoid memory leaks and other issues.
* Provides flexibility by automatically handling memory management tasks, allowing progra- -mmers to focus on application logic.	* Offers more control over memory management, allowi- -ng programmers to fine-tune memory usage but requiring more manual effort.
* can impact performance due to occasional pauses	* Generally offers more predictable performance since

during garbage collection cycles	memory management is under direct programmer control.
* commonly found in higher level languages like java, python and c#	* More prevalent in level level languages like c and c++, though dynamic memory management may still be available through libraries.
* Easier for developers as they don't have to worry about memory management details.	* Requires more effect from developers to manage memory properly, but offers finer control over memory usage.

04. What is compaction ? How it different Garbage collection ?

compaction is a memory management technique used to reduce fragmentation in a memory heap. When objects are allocated and deallocation dynamically during program execution, they leave gaps of free memory in between. Over time, these gaps can lead to fragmentation where the free memory is scattered throughout the heap in small chunks, making it challenging to allocate large contiguous block of memory. compaction works by rearra-nging the live objects in memory to fill in the gaps left by deallocated objects. This process involves moving live objects closer together, effectively consolidating free space into larger contiguous blocks.

* Difference between compaction and Garbage collection

1.) Definition :-

• compaction :- Involves rearranging memory to reduce fragmentation by moving allocated objects closer together and filled in the gaps left by reclaimed memory.

• Garbage collection :- Involves reclaiming memory occupied by objects that are no longer in use (garbage), typically using algorithms like mark-and-sweep or copying collection.

2.) Objective :-

• compaction :- Aims to reduce memory fragmentation, which can lead to more efficient memory usage and

improved performance by reducing the overhead of memory allocation and deallocation.

- Garbage collection :- Focuses on reclaiming memory occupied by unreachable objects to prevent memory leaks and ensure efficient memory utilization

3) Process :-

- compaction :- Requires identifying free memory blocks and moving live objects to eliminate fragmentation, which may involve updating references to the moved objects.
- Garbage collection :- Involves identifying unreachable objects and reclaiming their memory, typically without rearranging the memory layout.

4.) Impact on Performance :-

- compaction :- can introduce overhead due to the need to move objects and update references, but can lead to improved performance by reducing memory fragmentation
- Garbage collection :- may introduce pauses or overhead during collection cycles, but its primary goal is to reclaim memory rather than optimizing memory layout.

05. What is the purpose of the finalize() method in Java, how does it relate to object cleanup and garbage collection?

The 'finalize()' method in Java is a method provided by the 'Object' class that allows an object to perform cleanup operations just before it is garbage collected. Its purpose is to give the object a chance to release any non-memory resources it may hold, such as file handles, database connections, or network sockets. However, it's important to note that the 'finalize()' method is not guaranteed to be called promptly or at all by the garbage collector, so it should not be relied upon for critical resource cleanup.

* How the 'finalize()' method related to object cleanup and garbage collection:

1.) Object cleanup:

- When an object is no longer referenced by any part of the program, it becomes eligible for garbage collection.
- Before the object is reclaimed by the garbage collector, the 'finalize()' method, if overridden, will be invoked by the garabage collector.

2) 'finalize()' Method Invocation:

- The 'finalize()' method is called by the garbage collector just before reclaiming the memory occupied by the object.
- This gives the object an opportunity to release any non-memory resources it holds, such as closing files or releasing network connections.

3) Finalization Process:

- When the 'finalize()' method is called, the object can perform cleanup operations and prepare for garbage collection.
- After the 'finalize()' method completes, the object becomes eligible for garbage collection, and its memory is reclaimed.

4.) Limitations and Caution:

- The invocation of the 'finalize()' method is not guaranteed to happen promptly or at all, as it depends on the garbage collectors behavior.
- Relying solely on 'finalize()' for critical resource cleanup is not recommended, as it may lead to resource leaks if the method is not invoked in a timely manner.

5.) Superseded by 'try-with-resources' and 'AutoCloseable':

- In modern Java, the preferred approach for resource cleanup is to use 'try-with-resource' or implement the 'AutoCloseable' interface.
- These mechanisms ensure that resources are properly released regardless of whether garbage collection occurs, offering more deterministic resource cleanup.