**What is Thread?**

A thread is smallest unit of execution within a program.

**What is Multithreading?**

Multithreading allows concurrent execution of two or more threads, enabling maximum utilization of CPU.

Each thread runs in parallel, performing separate tasks and improving the efficiency and performance of an application.

**How to create a thread?**

There are two main ways to create a thread in Java.

one way is by extending Thread class.

another way is by implementing Runnable interface.

overriding the run method to specify the task that thread will perform.

**Explain Thread life cycle?**

A thread several states during its lifecycle like new, runnable, running, block or wait and termination.

New:

A thread in new state means thread is created but not yet started.

Runnable:

When call start(), the thread is ready to run and is waiting for the CPU.

Running:

The thread is executing its run method.

Blocked/Waiting:

The thread is blocked or waiting for a resource or event.

Terminated:

The thread has finished executing.

**Explain some important methods provided by Thread class?**

start

Start method begins the execution of the thread by calling its run method.

run

run method contains the code that defines the behavior of the thread.

sleep

sleep method pauses the thread for a specified period, releasing the CPU.

Join

join method waits for the thread to complete its execution.

yield

yield method temporarily pauses the currently executing thread and allows other threads to execute.

interrupt

interrupt method interrupts the thread, typically used to signal the thread to stop.

**How inter thread communication works?**

Threads communicate using methods like wait, notify, and notifyAll.

wait method pauses the current thread to wait until another thread invokes notify method or notifyAll method on the same object.

notify method wakes up a single thread that is waiting on the object's monitor.

notifyAll method wakes up all threads that are waiting on the object's monitor.

**What is synchronization?**

Synchronization is used to control the access of multiple threads to shared resources.

Synchronization prevents thread interference and memory consistency errors.

Synchronization can be achieved using synchronized methods or blocks.

Example for Synchronized Method:

public synchronized void synchronizedMethod() {

// synchronized code

}

Example for Synchronized Block:

public void method() {

synchronized (this) {

// synchronized code

}

}

**How Synchronization works internally?**

Synchronization in Java is built around the concept of an intrinsic lock (or monitor lock).

Each object in Java has an intrinsic lock associated with it.

When a thread enters a synchronized method or block, it acquires the lock associated with the object.

If another thread attempts to enter a synchronized method/block on the same object, it will be blocked until the lock is released.

A monitor is a mechanism that is used to coordinate the actions of multiple threads.

When a thread holds a lock on an object, it is said to be "in the monitor" for that object.

In java monitorenter and monitorexit instructions used to implement synchronization.

monitorenter is executed when a thread enters a synchronized block/method, acquiring the lock.

monitorexit is executed when the thread exits the synchronized block/method, releasing the lock.

In the HotSpot JVM, every object has a header that includes a mark word.

The mark word contains information about the object's state, including the lock status.

The mark word can be in various states such as unlocked, lightweight locked, inflated (heavyweight) locked, and biased.

Java supports lightweight and heavyweight locks.

Initially, the JVM tries to use lightweight locks, which are cheaper and faster.

If contention occurs, the JVM inflates the lock to a heavyweight lock, which uses a monitor object to manage synchronization.

**Example:**

When t1 enters the increment method, it acquires the intrinsic lock of the counter object.

Similarly, when t2 enters the decrement method's synchronized block, it tries to acquire the same lock.

If t1 holds the lock, t2 will be blocked until t1 releases the lock and vice versa.

This ensures that only one thread can execute the synchronized code at a time, preventing race conditions.

When the thread exits the synchronized method/block, it releases the lock, allowing other waiting threads to acquire it.

**What are advantages and disadvantages of synchronization?**

**Advantages:**

Synchronization ensures that only one thread accesses at a time.

Synchronization preventing data inconsistency and corruption.

Synchronized blocks/methods ensure that a series of operations are executed as a single, uninterruptible unit. This is called Atomicity.

**Disadvantages:**

Synchronization has performance overhead issue due to the locking mechanism, especially with heavyweight locks.

Improper synchronization can lead to deadlocks so two or more threads are waiting indefinitely for each other to release locks.

Excessive synchronization can limit the scalability of an application, as it reduces the parallelism.

**What is atomicity?**

Operations that are performed as a single unit of work.

Atomic operations completes in a single step.

The atomic operation is executed completely or not at all. There is no intermediate state.

The atomic operation is isolated from other operations, means execution is independent and does not interfere with other concurrent operations.

Java provides several classes in the java.util.concurrent.atomic package to handle atomic operations.AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference, etc.

Methods are ensuring that each operation completes entirely before another one begins.

**What kind of issues will generate if there is no atomicity?**

Without atomicity race condition and data corruption issue will generate.

Without atomic operations, two threads might read the same value and then write back different results, leading to inconsistent states. This is called race condition.

When threads interleave their operations on shared data, it can lead to corrupted data if the operations are not atomic.

**What is Compare-And-Swap(CAS)?**

CAS is a hardware-supported atomic instruction that compares the current value of a memory location to an expected value and, if they match, updates the memory location to a new value.

Retry if the current value does not match the expected value.

CAS is atomic and is the foundation of many non-blocking algorithms.

**What is semaphore?**

Semaphore restricts the number of threads that can access a resource.

The Semaphore class in the java.util.concurrent package provides concurrent access to resources.

Semaphore control the number of permits available for use.

Threads acquire permits before accessing the resource and release them after they are done.

Threads can block until a permit is available and immediately return if a permit is not available. acquire, tryAcquire methods are used for this.

A binary semaphore is a special case with only one permit, functioning similarly to a mutex lock.

The Semaphore class provides constructors to create semaphores with a specified number of permits and an option to specify if the semaphore should be fair (first-in-first-out granting of permits).

Threads can acquire permits using the acquire method and Threads release permits using the release method.

A Semaphore object is created with 2 permits, meaning only two threads can access the shared resource concurrently.

Fair Semaphore granted permits in the order they are requested (FIFO order).

Non-fair Semaphore does not guarantee any particular order.

**Explain thread locking System?**

A thread must acquire the lock before entering the critical section and release the lock after it has finished using the resource.

If another thread has already acquired the lock, any subsequent threads attempting to acquire the same lock will be blocked until the lock is released.

Locks are implemented using the synchronized keyword and the Lock interface.

Java provides more advanced locking mechanisms through the Lock interface and it’s implemented by ReentrantLock class.

**What is mutex lock?**

A mutex lock is short for mutual exclusion lock.

It prevents multiple threads from simultaneously accessing a shared resource.

It provides only one thread can access the resource at a time.

It maintains data consistency and prevent race conditions.

In Java, mutex locks are implemented using the synchronized keyword and the Lock interface.

**What is dead lock?**

A deadlock is a situation in concurrent programming where two or more threads are unable to proceed with their execution because each thread is waiting for a resource that is held by another thread, creating a circular dependency. So none of the threads can continue, and the system effectively becomes stuck.

**How to avoid dead locks?**

Implement lock ordering. Always acquire locks in a consistent order. Ensure that all threads acquire resources in the same order to prevent circular wait.

Use timeout mechanisms when trying to acquire locks. If a thread cannot acquire a lock within a specified time, it releases any locks it holds and retries.

Periodically check dead lock cycles in the resource allocation graph and take corrective actions.

Design the system such that resources can only be requested in a hierarchical manner, eliminating circular dependencies.

Minimize the use of nested locks whenever possible. If nested locks are necessary, ensure they are always acquired in the same order.

**what is thread pool?**

A thread pool is a collection of pre-instantiated reusable threads that are available for performing tasks.

Instead of creating a new thread for each task, tasks are assigned to a pool of existing threads.

Threads in the pool are reused for executing multiple tasks, reducing the overhead associated with creating and destroying threads.

Tasks submitted to the thread pool are queued if no threads are available tasks are executed as soon as a thread becomes available.

The thread pool controls the level of concurrency, balancing the load and preventing too many threads from running simultaneously.

**How to create Thread pool?**

Java provides built-in support for thread pools through the java.util.concurrent package, which includes the Executor, ExecutorService, and Executors classes.

The Executors utility class provides several factory methods for creating different types of thread pools.

newFixedThreadPool method creates a thread pool with a fixed number of threads.

newCachedThreadPool method creates a thread pool that creates new threads as needed but will reuse previously constructed threads when they are available.

newSingleThreadExecutor method creates a thread pool with a single thread that executes tasks sequentially.

newScheduledThreadPool method creates a thread pool that can schedule commands to run after a given delay or to execute periodically.

**what are multi threading enhancements in java8?**

CompletableFuture handle asynchronous programming. It allows you to build complex pipelines of asynchronous tasks.

You can chain multiple asynchronous tasks together using methods like thenApply, thenAccept, thenRun, thenCombine, etc.

CompletableFuture provides methods to combine multiple futures.

Handle exceptions in the asynchronous pipeline using exceptionally() and handle() methods.

Create a parallel stream by calling parallelStream() on a collection or by using the parallel() method on a stream.

Parallel streams automatically partition the stream’s data and process it in parallel, potentially improving performance for large datasets.

The ForkJoinPool, which is a framework for parallel execution of tasks that can be divided into smaller tasks.

The common pool is a default ForkJoinPool that can be used by parallel streams and other concurrent tasks.

We can create custom ForkJoinPool instances for more control over parallel execution.

StampedLock is a new kind of lock introduced in Java 8 that supports three modes for controlling read/write access to a resource.

Java 8 introduced LongAdder and DoubleAdder classes, which are designed to handle high contention scenarios more efficiently than AtomicLong and AtomicDouble.

**Explain Java Memory Model (JMM)?**

The Java Memory Model (JMM) describes how threads interact through memory.

JMM defines rules for thread visibility to share data safely.

It ensure series of operations are performed as a single, indivisible step.

It defines a partial ordering of read and write operations for compiler optimizations.

If one action happens-before another, then the first is visible and ordered before the second.

If A happens-before B, and B happens-before C, then A happens-before C.

**Explain Happens-Before Relationship?**

If one action happens-before another, then the first is visible and ordered before the second.

Each action in a thread happens-before every subsequent action in that thread.

An unlock on a monitor lock happens-before every subsequent lock on that monitor.

A write to a volatile field happens-before every subsequent read of that field.

A call to Thread.start() on a thread happens-before any actions in the started thread.

Any action in a thread happens-before another thread detects that the thread has terminated, either by joining it via Thread.join() or by Thread.isAlive() returning false.

If A happens-before B, and B happens-before C, then A happens-before C it is called as transitivity.

**Explain Memory Barriers?**

Memory barriers (or fences) are low-level primitives used to enforce ordering constraints on memory operations.

Acquire Barrier: ensures that all reads and writes before the barrier are completed before any read or write after the barrier.

Release Barrier: ensures that all reads and writes before the barrier are visible to other processors before any read or write after the barrier.

**Explain Volatile Variables?**

the volatile variable is always visible to other threads.

A read of a volatile variable always returns the most recent write by any thread.

Volatile writes have the same memory semantics as a release barrier.

Volatile reads have the same memory semantics as an acquire barrier.