Java Threads:

Threads are a lightweight process for executing a-;;;;; task in a process and multiple threads can be executed in a process and runs in a same shared memory.

**Two ways of achieving multithreading in java**

Extends Thread.class

Implementing Runnable Interface

Thread Life Cycle:

New() à Running()/Runnable() à Blocked()/ Waiting() à Terminated()

Types of Thread in JVM:

**Worker Thread** àThis type of Thread is a normal thread created for executing a task with a customized priority and cannot by terminated by JVM

**Deamon Thread** à Low priority threads terminated by JVM whenever the worker thread completed its execution and it runs in background for running small tasks

Thread scheduler decides the priority of thread execution

When Threads with same priority are executed by FIFO

T1.join à waits for the thread to complete the t1 thread execution

**Diff Between sleep and wait:**

* you call wait on the **Object**while on the other hand you call sleepon the **Thread**itself
* wait can be interrupter (this is why we need the *InterruptedException*) while on the other hand sleep can not
* wait (and notify) must happen in a synchronized  block on the monitor object whereas sleep does not
* sleep operation does not release the locks it holds while on the other hand wait releases the lock on the object that wait() is called on

**What is a Lock in Java?**

**Locks allows us to restrict access to a shared resource such that only one thread can access that resource.**

Let's clarify the difference between class-level locking and object-level locking in Java. This distinction is crucial for understanding how synchronized blocks and methods control concurrency.

**The Core Concept: What is a Lock in Java?**

In Java, every object (including Class objects) has an **intrinsic lock** (also known as a monitor lock). When a thread enters a synchronized block or method, it attempts to acquire this intrinsic lock.

* If the lock is available, the thread acquires it and proceeds.
* If the lock is already held by another thread, the current thread blocks until the lock is released.

# **Thread starvation** is a phenomenon in concurrent programming where a thread is continuously denied access to a shared resource or CPU time, even though the resource or CPU might be available. As a result, the thread is unable to make progress and complete its tasks.

**DeadLock** occurs when two or more threads waiting each other for a lock or resource indefinitely.

Deadlock can be avoided by eliminating cyclic dependencies among the threads.

**Livelock** is a state in concurrent programming where two or more processes or threads continuously change their state in response to changes in the other processes, but without making any actual progress on their intended task.

**Atomic Variable:** The primary purpose of atomic variables is to provide a **thread-safe, non-blocking alternative** to using synchronized blocks or (java.util.concurrent.locks.Lock) for simple, single-variable updates.

In Java concurrency, a **Semaphore** is a synchronization primitive that controls access to a shared resource. It maintains a set of permits. A thread that wants to access the resource must first acquire a permit. If permits are available, the thread takes one. If no permits are available, the thread blocks until a permit becomes available. When the thread is finished with the resource, it releases the permit, allowing another thread to acquire it.(i.e used to control access to a shared resource)

In summary, a **Mutex** in Java concurrency is a fundamental mechanism to ensure that only one thread can execute a particular section of code (the critical section) at a time, preventing data corruption and maintaining consistency in multithreaded applications. The synchronized keyword and ReentrantLock are the primary ways to achieve mutex behaviour in Java.

**ReentrantLock** provides a more flexible and powerful alternative to the traditional synchronized keyword for controlling access to shared resources in a multithreaded environment.

his is the most defining characteristic, and where the name comes from. A thread that has already acquired a ReentrantLock can acquire it again multiple times without blocking itself. For each successful lock() call, there must be a corresponding unlock() call for the lock to be truly released. This is similar to how intrinsic (synchronized) locks behave. If a thread acquires the lock 3 times, it must release it 3 times before another thread can acquire it.

**Advanced Features over synchronized:** ReentrantLock offers several capabilities that synchronized does not:

Fairness Policy:

Timed Lock Acquisition

Non-Blocking Lock Acquisition

Condition Variables

**ExecutorService** simplifies managing concurrent tasks, especially when you need to execute multiple task and has thread pools to eliminate manual thread creations.

Java Collections can be synchronized by using **Collections.synchronizedMap** method to make it thread safe, but it’s not efficient

**Blocking Queue:**

It's a thread-safe queue

BlockingQueue provides a safe and efficient way for threads to communicate by exchanging data. It automatically manages the flow control:

* **Consumers stop (block) when there's nothing to do.**
* **Producers stop (block) when they can't do any more (queue is full).**

This "blocking" behavior makes BlockingQueue ideal for implementing the **Producer-Consumer pattern**, where different parts of your application produce data and other parts consume it, without needing complex manual synchronization or explicit wait()/notify() logic. The BlockingQueue handles all that for you.

**Priority Queue:**

**Thread-Safe:** This is its most important feature. All operations (adding, removing, peeking) are internally synchronized, meaning you can safely use it with multiple producer and consumer threads concurrently without needing external locks or synchronization mechanisms.

* Instead, they are ordered based on their **priority**. The element with the *highest priority* (which means the "smallest" value according to the queue's ordering) will always be at the head of the queue, ready to be retrieved.
* Priority is determined either by:
  + The **natural ordering** of the elements (if they implement Comparable).
  + A **custom Comparator** provided when the PriorityBlockingQueue is created.
* Heap-Based Implementation (Min Heap).
* No Null Values are allowed.

**conCurrentMap:**

The key goal of ConcurrentMap implementations is to achieve high concurrency by allowing multiple threads to perform read operations concurrently.

While Hashtable and Collections.synchronizedMap() provide thread safety, they do so by synchronizing on the *entire map* for almost every operation. This means only one thread can access the map at a time (even for read operations), leading to **poor performance and scalability** under high contention.

Historically, ConcurrentHashMap used a technique called **segment-level locking**. The map was divided into several segments, and each segment had its own lock. A thread would only need to acquire the lock for the specific segment it was operating on. This allowed multiple threads to operate on different segments simultaneously

ConcurrentMap (and specifically ConcurrentHashMap) is explicitly designed to allow and manage **concurrent write operations** from multiple threads. This is its core value proposition, providing a highly efficient and scalable solution for shared map access in concurrent Java applications.

It guarantees Atomic operations

The **CopyOnWriteArrayList** in Java's java.util.concurrent package is a specialized, thread-safe variant of ArrayList designed for scenarios where **reads vastly outnumber writes**. Its name, "Copy-On-Write," perfectly describes its core mechanism.

**Core Principle: Copy-On-Write**

1. **Reads are Fast and Concurrent:** When you perform a read operation (like get(), iterator(), size(), contains()), CopyOnWriteArrayList does *not* acquire any locks. Threads can read the list concurrently without blocking each other. This is because reads operate on a stable, immutable snapshot of the array.
2. **Writes Create a New Copy:** When a **modifying operation** occurs (like add(), set(), remove(), clear(), addAll()), CopyOnWriteArrayList does the following:
   * It takes a **fresh copy** of the underlying array.
   * It performs the modification on this **new copy**.
   * Once the modification is complete, it **replaces the old array with the new, modified array** (an atomic operation, typically using a volatile reference).