**Day\_6(Multithreding)**

# ****What is Multithreading in Java?****

* **Multithreading** = process of executing **multiple threads** concurrently within a program.
* A **thread** is the **smallest unit of execution** (lightweight subprocess).
* Java provides **built-in support** for multithreading using the Thread class and the Runnable interface.

💡 Example in real life:

* Imagine you are listening to music 🎶 while downloading a file ⬇️ and typing a document ⌨️.
* All 3 tasks run **at the same time** → that’s **multithreading**.

# ****Why Multithreading?****

**Better CPU utilization** – threads share the same process memory, so CPU is not idle.

**Concurrent execution** – multiple tasks run at the same time.

**Improved performance** – especially in I/O heavy tasks (networking, DB calls).

**Resource sharing** – threads share memory of the process (no need for heavy inter-process communication).

# ****How to Create a Thread in Java****

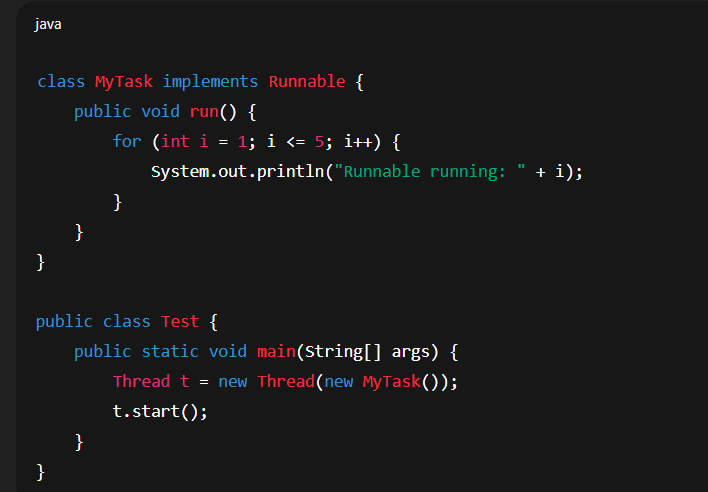
* There are 2 ways:

### ✅ 1. Extending Thread class

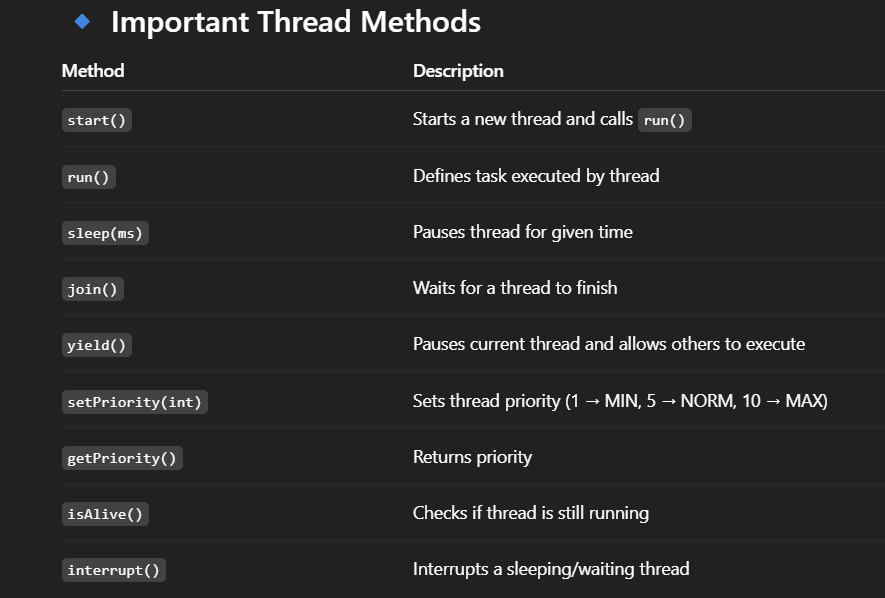


👉 start() → creates a new thread & calls run() internally.

### ✅ 2. Implementing Runnable interface



👉 Better approach (because Java supports **multiple inheritance via interfaces**).



# ****Thread Lifecycle (States)****

* A thread goes through these states:

**New** → created but not started (Thread t = new Thread();)

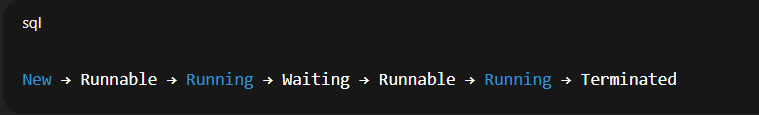
**Runnable** → ready to run, waiting for CPU.

**Running** → executing task (run() method).

**Waiting / Timed Waiting** → thread temporarily inactive (sleep(), join(), wait()).

**Terminated** → task finished.

👉 Diagram:



# ****Advantages of Multithreading****

✅ Efficient CPU utilization  
✅ Faster execution (when tasks are independent)  
✅ Saves memory (threads share same address space)  
✅ Useful in games, animations, servers, networking apps

# **Disadvantages / Challenges**

❌ **Complexity** → hard to debug and maintain  
❌ **Thread Interference** → when 2 threads access same resource at same time  
❌ **Deadlock** → when 2 threads wait forever on each other’s resources  
❌ **Context switching overhead** → frequent switching slows performance

# ****Real-Life Use Cases****

Web servers (handle multiple requests simultaneously)

Banking systems (multiple transactions at once)

Games (background music, player actions, rendering)

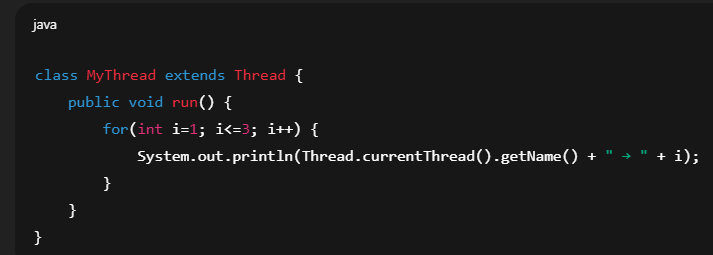
Chat applications (sending + receiving messages simultaneously)

* **Example for** understand **how threads work internally in Java**. Let's break it down **step by step**, from creation to execution, and explain what happens inside the JVM and OS.



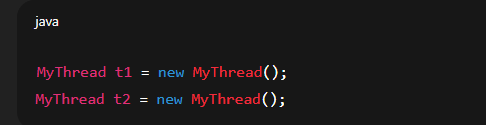
## ✅ ****Step-by-step Execution****

### ✅ 1. ****Thread Class Declaration****



* define a class MyThread that extends Thread.
* override the run() method → this is the code the thread will execute.

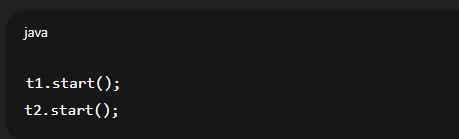
### ✅ 2. ****Thread Objects are Created****



· Two separate Thread objects are created → each with its own internal thread stack.

· At this point, the threads are in the **NEW** state → not yet started.

### ✅ 3. ****Starting the Threads****



* start() method is called on both threads → this triggers the JVM to:

1. Create a new thread in the operating system.
2. Schedule the thread for execution.
3. Move the thread from **NEW** state to **RUNNABLE** state.

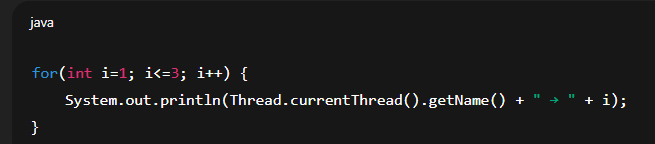
### ✅ 4. ****Thread Scheduling****

* JVM hands over thread management to the underlying OS scheduler.
* The OS decides which thread runs first → this can vary each time you run the program.
* Both threads are eligible → they compete for CPU time.

### ✅ 5. ****Running the Threads****

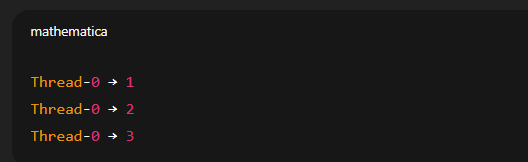
* Once the OS picks a thread:

#### For example, t1 is picked first:

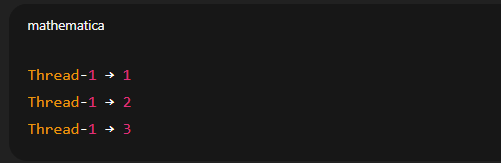


· The run() method is executed for t1.

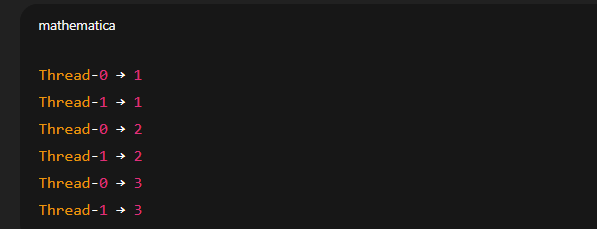
· It prints:



* Meanwhile, t2 is waiting or running depending on the scheduler.
* Or, if t2 is picked first, it prints:



* Sometimes, the output is **interleaved**, like:



This happens because the scheduler switches between threads after executing a few instructions.

### ✅ 6. ****Thread States****

* Throughout execution, the thread goes through the following states:
* **NEW** → object created, not started
* **RUNNABLE** → after start(), waiting for CPU
* **RUNNING** → actively executing run()
* **TERMINATED** → after run() completes, thread is dead

### ✅ 7. ****Thread Resources****

* Each thread has its own:
* **Program Counter** – keeps track of where it is in execution.
* **Stack** – stores local variables like i.
* **Thread Name** – default names like Thread-0, Thread-1.
* Both threads share:
* **Heap memory** – where objects are stored (e.g., MyThread instances).

## ✅ ****Key Points****

* Calling start() creates a new thread and schedules it for execution.
* The JVM and OS cooperate to manage threads.
* run() is executed in parallel when possible.
* Threads share the heap but have separate stacks.
* The output may differ due to the OS scheduling threads differently.

# ✅ ****What is Context Switching in Threads?****

* **Context Switching** in threads is the process where the CPU **pauses the execution of one thread** and **resumes the execution of another thread**. It involves saving the current thread’s state and loading the next thread’s state.
* This allows **multithreading**, where multiple threads share CPU time and appear to run in parallel.

# ✅ ****Why Context Switching Happens?****

* There are more threads than CPU cores → CPU time must be shared.
* A thread may:
* Complete its task.
* Wait for I/O (file read, network, etc.).
* Be preempted by the scheduler to allow other threads to run.
* **Interview questions**

# ****1.What are**** notify() ****and**** notifyAll() ****in Java?****

* Both methods belong to the **Object class** (not Thread class).
* Used in **inter-thread communication** → threads communicate about the availability of a resource.
* Always used inside a **synchronized block or synchronized method**, otherwise IllegalMonitorStateException.

# ****Key Methods****

**1.wait()**

* Tells the current thread to **release the lock** and go into **waiting state** until notified.
* Example: A consumer thread waits if no product is available.

**2.notify()**

* Wakes up **one** waiting thread (chosen randomly by JVM).
* Example: Producer notifies one waiting consumer when product is ready.

**3.notifyAll()**

* Wakes up **all waiting threads** (but they still compete for lock → only one runs at a time).
* Example: Producer notifies all consumers that resource is available.

# ****Example: Producer-Consumer Problem****

class SharedResource {

private int data;

private boolean hasData = false;

// Produce method

public synchronized void produce(int value) throws InterruptedException {

while (hasData) {

wait(); // wait if data already produced

}

data = value;

System.out.println("Produced: " + data);

hasData = true;

notify(); // wake up one consumer

}

// Consume method

public synchronized void consume() throws InterruptedException {

while (!hasData) {

wait(); // wait if no data

}

System.out.println("Consumed: " + data);

hasData = false;

notifyAll(); // wake up all waiting producers

}

}

public class Test {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

// Producer thread

Thread producer = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

try {

resource.produce(i);

} catch (InterruptedException e) {}

}

});

// Consumer thread

Thread consumer = new Thread(() -> {

for (int i = 1; i <= 5; i++) {

try {

resource.consume();

} catch (InterruptedException e) {}

}

});

producer.start();

consumer.start();

}

}

# ****Output Example****

Produced: 1

Consumed: 1

Produced: 2

Consumed: 2

...

**2.If I have three threads t1, t2, and t3, and I want to make sure that t2 starts only after t1 has finished, and t3 starts only after t2.**

* **Solution 1: Using Thread.join()**
* In Java, join() allows one thread to wait until another thread has finished execution. So in my main method, I would first start t1, then call t1.join(). This ensures that the main thread will pause until t1 completes. After that, I start t2, then call t2.join(), and finally do the same for t3.
* This guarantees the order t1 → t2 → t3. It’s very easy to implement .

**class MyThread extends Thread {**

**private String name;**

**MyThread(String name) {**

**this.name = name;**

**}**

**@Override**

**public void run() {**

**System.out.println(name + " started");**

**try {**

**Thread.sleep(1000); // simulate work**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.out.println(name + " finished");**

**}**

**}**

**public class ThreadOrderExample {**

**public static void main(String[] args) {**

**Thread t1 = new MyThread("T1");**

**Thread t2 = new MyThread("T2");**

**Thread t3 = new MyThread("T3");**

**try {**

**t1.start();**

**t1.join(); // Wait until t1 finishes**

**t2.start();**

**t2.join(); // Wait until t2 finishes**

**t3.start();**

**t3.join(); // Wait until t3 finishes**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

### Output (order always maintained)

T1 started

T1 finished

T2 started

T2 finished

T3 started

T3 finished

**3.why do we call start() method first ,what will hapeen run() call directly?**

### 1. start() ****creates a new thread****

When you call t1.start(), the JVM internally:

Creates a **separate call stack** for that thread.

Calls the run() method **inside that new thread**.

So execution is truly **multithreaded**.

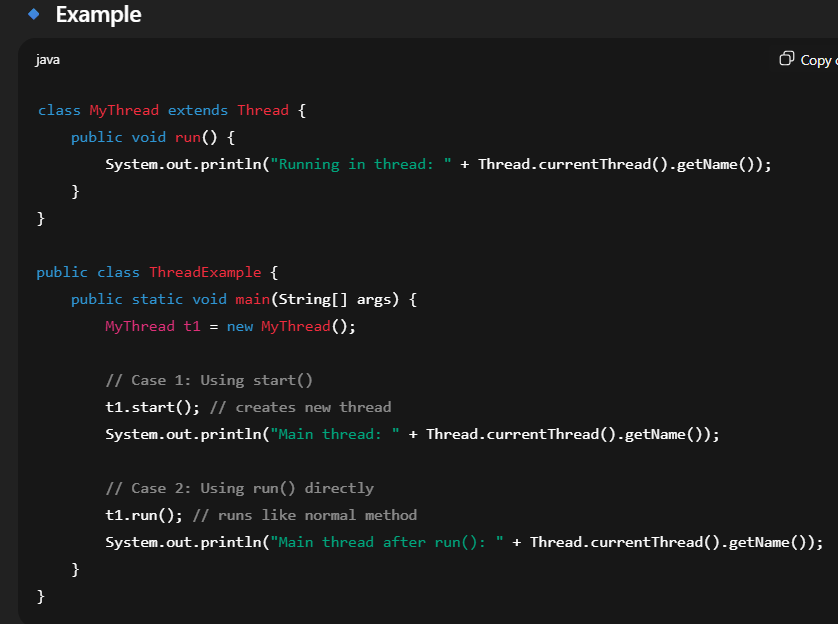
2. **run**() **called directly = just a normal method call**

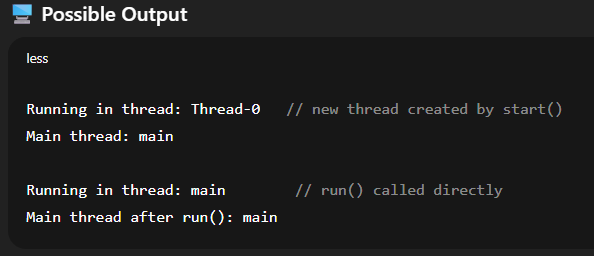
If you call t1.run() directly:

It **does not create a new thread**.

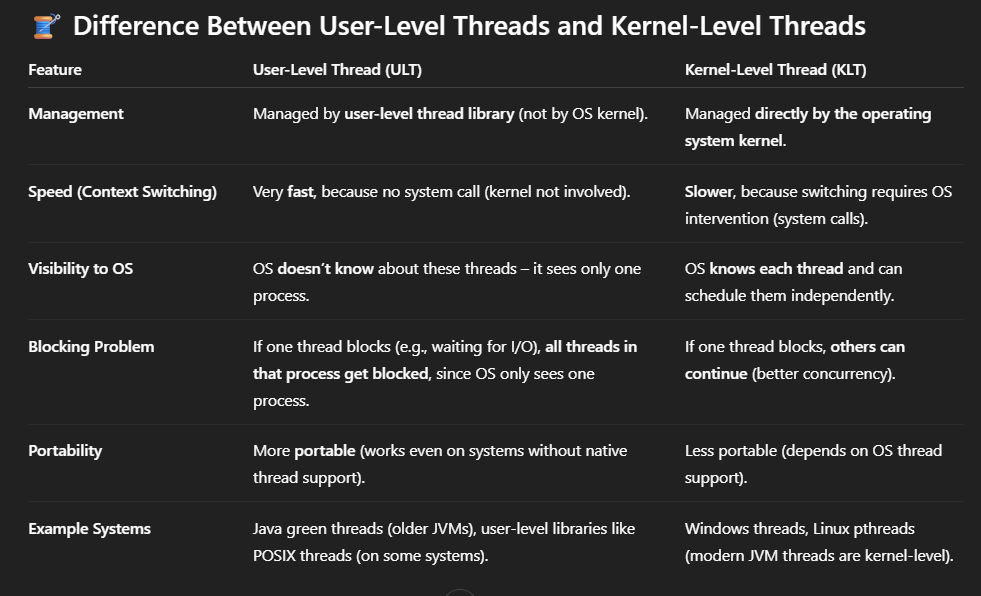
The run() method just executes in the **current thread (main thread)**.

So no multithreading happens, it behaves like a **normal method call**.





**4.difference between user level and kernel level thread**



# ****5.How to Wake a Blocked Thread in Java****

### 1. ****First, understand what “blocked” means****

In Java, a thread can be in **BLOCKED**, **WAITING**, or **TIMED\_WAITING** state:

* **BLOCKED** → waiting to acquire a lock (synchronized).
* **WAITING / TIMED\_WAITING** → waiting for notify() or timeout.

### 2. ****Ways to Wake a Thread****

#### a) If Thread is in WAIT() state

#### A thread goes into WAITING if it calls wait() inside a synchronized block.

* To wake it up, another thread must call notify() or notifyAll() on the same lock object.

#### b) If Thread is in SLEEP or JOIN (TIMED\_WAITING)

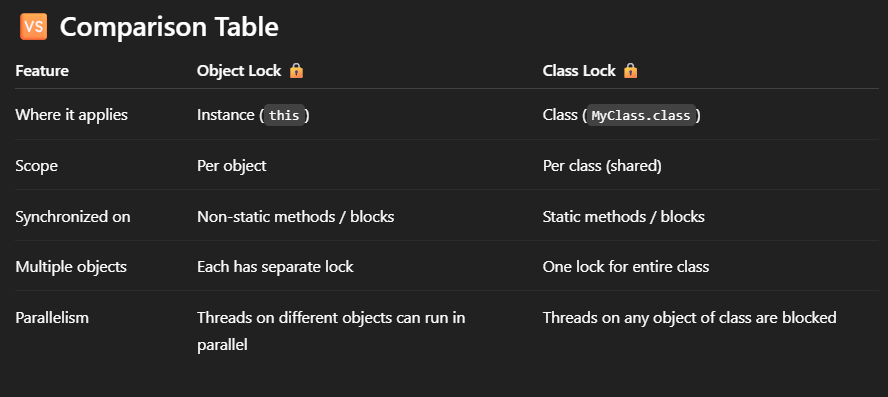
* You can’t directly wake it up, but you can call interrupt() on it.
* This throws InterruptedException and wakes the thread.

#### c) If Thread is in BLOCKED state (waiting for lock)

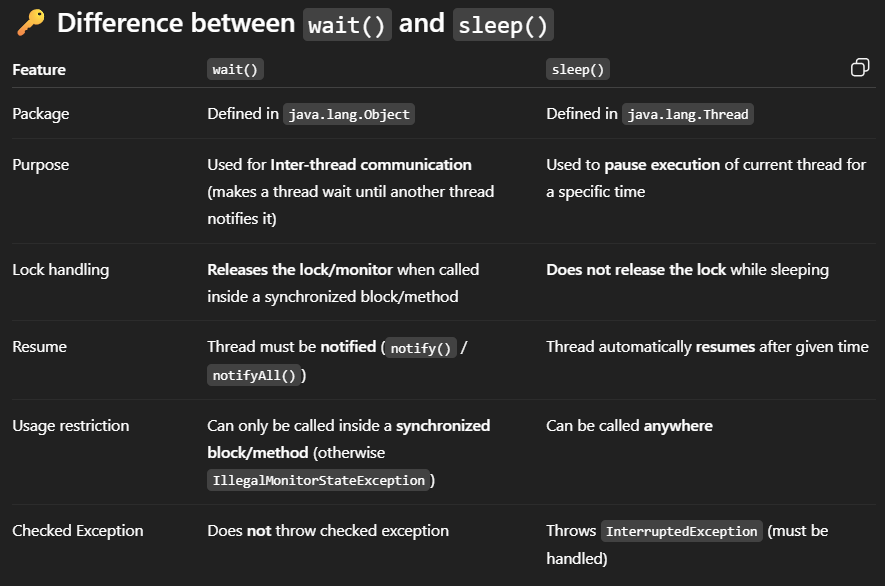
* You **cannot directly wake it** until the thread holding the lock releases it.
* Only when the lock becomes available, the blocked thread will automatically resume.
* **which is better to implement thread? extending Thread class or implementing Runnable innterface**
* Using Runnable is generally better than extending Thread because it provides more flexibility. It allows multiple inheritance (since you can extend another class), separates the task from the thread execution, and integrates well with modern concurrency utilities like ExecutorService. Extending Thread is only preferred for very simple cases or quick demos.

**6.what is difference between class lock and object lock**

An object lock is acquired when we use synchronized instance methods, and it applies to a specific object. A class lock is acquired when we use synchronized static methods, and it applies to the Class object, so it’s shared by all instances. That means object locks allow more concurrency, while class locks are stricter and serialize access across all objects of the class.



**7.what is difference between wait and sleep**



**8.which method will realease lock**

* In **Java multithreading**, the wait() **method** (from Object class) is the one that **releases the lock (monitor)**.

### 🔑 Key Points:

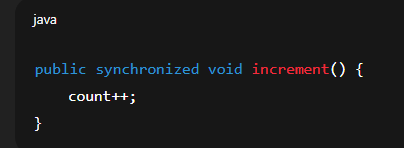
* **wait()**
* Must be called inside a synchronized block/method.
* When a thread calls wait(), it:
* **Releases the lock** on the object.
* Moves to the **waiting state**.
* Will only resume when another thread calls notify() or notifyAll() on the same object.
* **sleep()**
* Does **not release the lock**.
* It just **pauses the execution** of the thread for the given time, but if the thread was holding a lock, it still holds it during sleep.
* **yield()**
* Also **does not release lock**.
* Just hints to the scheduler to give CPU to other threads of the same priority.

**9.what is race condition?**

* A **race condition** occurs when **two or more threads access shared data at the same time**, and the **final outcome depends on the order in which the threads are scheduled**.  
  Since thread scheduling is not predictable, the result becomes **inconsistent or incorrect**.

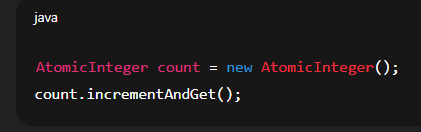
# ****How to Prevent Race Conditions****

1. ****Synchronization (synchronized keyword)****

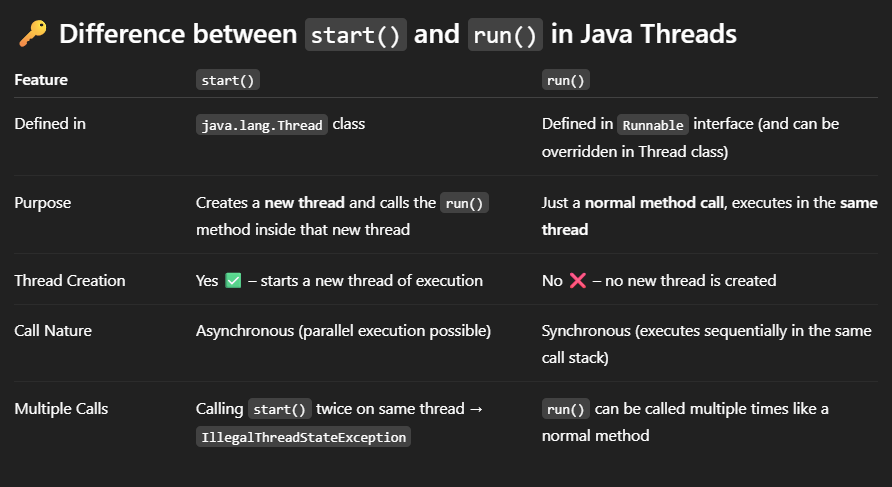


2. **Lock API (**ReentrantLock**)**  
More advanced control over locking.

3. **Atomic Variables (e.g.,** AtomicInteger**)**

·

**10.difference between start() and run() method**



* start() actually creates a new thread and then internally calls the run() method in that new thread, while calling run() directly does not create a new thread — it just executes like a normal method in the current thread.

# ****11.What is common Problems you have faced in Multithreading Environment? And how to resolve it?****

### 1. ****Race Condition****

* Multiple threads access shared data at the same time without proper synchronization.
* Leads to **inconsistent or incorrect results**.  
  ✅ Solution: Use synchronized, Lock, or Atomic classes.

### 2. ****Deadlock****

* Two or more threads wait for each other’s resources, so **none can proceed**.
* Example: Thread A holds Lock1 and waits for Lock2, while Thread B holds Lock2 and waits for Lock1.  
  ✅ Solution: Avoid nested locks, use lock ordering, or tryLock().

### 3. ****Starvation****

* A thread **never gets CPU or resources** because other threads keep taking them.
* Example: If higher priority threads keep running, a lower priority thread may starve.  
  ✅ Solution: Use **fair locks** (ReentrantLock(true)), avoid too much priority bias.

### 4. ****Livelock****

* Threads keep **responding to each other** but no progress is made.
* Example: Two people trying to cross a narrow passage, both step aside repeatedly without crossing.  
  ✅ Solution: Better coordination logic.

### 5. ****Thread Interference****

* When two threads **update shared data simultaneously**, updates are lost.
* Example: Two threads incrementing a counter at the same time.  
  ✅ Solution: Synchronization, atomic variables.

### 6. ****Context Switching Overhead****

* Too many threads cause frequent **context switches** → performance drops.  
  ✅ Solution: Use thread pools (ExecutorService) instead of creating too many threads.

### 7. ****Memory Consistency Errors****

* Threads may **not see updated values** of shared variables because of CPU caching.  
  ✅ Solution: Use volatile keyword or synchronization.

# ****What is Thread Priority in Java?****

👉 **Thread priority** is a value that helps the **thread scheduler** decide the order in which threads are executed when multiple threads are ready to run. Higher priority threads are more likely to be executed before lower priority threads, though it depends on the OS and JVM implementation.

### ✅ Key Points

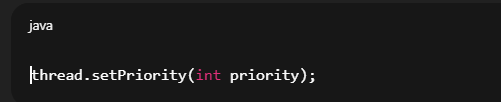
* Each thread has a **priority value between 1 and 10**:

Thread.MIN\_PRIORITY = 1 (lowest priority)

Thread.NORM\_PRIORITY = 5 (default priority)

Thread.MAX\_PRIORITY = 10 (highest priority)

* By default, every thread is assigned **normal priority (5)**.
* You can set the priority using:



* A higher priority thread may get more CPU time, but **it's not guaranteed** that it will always run before others.