Multithreading

1. What is Multithreading?

Multithreading = **Ek program me multiple tasks (threads) ko parallel chalana**.

Thread = smallest unit of execution

Example (real world):

- YouTube app
  - 1 thread: video play
  - 1 thread: audio play
  - 1 thread: buffering
  - 1 thread: ad load

All run **together** → smooth experience.

2. Process vs Thread (MOST asked)

Feature	Process	Thread
Memory	Separate memory	Shared memory
Communication	Slow	Fast
Resource Heavy	Yes	No
Crash	Process crash does NOT affect others	Thread crash can affect whole app
Example	Chrome, VSCode	Chrome tabs, VSCode extensions

Real-world example:

- Process = poora ghar
- Thread = ghar ke andar ke rooms
  Rooms **same resources** share karte hain (electricity, WiFi)

3. Why Multithreading? (Benefits)

1. Faster execution

Tasks parallel run hote hain → performance boost.

2. Resource sharing

Threads same memory share karte hain → efficient.

3. Better user experience

UI freeze nahi hota.

4. CPU utilization

Multicore CPUs ka full use hota hai.

4. Types of Multitasking

A. Process-based multitasking

(Multiple programs chal rahe hain)

- Chrome
- VSCode
- Spotify

B. Thread-based multitasking

Ek hi program ke andar multiple threads kaam karte hain.

5. Thread Lifecycle (Interview-Favorite)

Thread states:

1. **New**
2. **Runnable**
3. **Running**
4. **Blocked/Waiting**
5. **Terminated**

Diagram samajhne layak hai, code hum agle part me karenge.

Thread Creation in Java (VERY IMPORTANT)

Java me threads banane ke **2 tareeke** hain:

1. Extending Thread class

Yeh simple aur beginner-friendly method hai.

Step 1 — Thread class ko extend karo

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running...");
    }
}
```

`run()` = thread ka job

Java automatically `run()` ko thread ke andar execute karta hai

Step 2 — Thread start karna (IMPORTANT!)

```
MyThread t = new MyThread();
t.start();
```

`start()` vs `run()` difference (INTERVIEW FAVOURITE)

start()	run()
New thread create karta hai	Normal method call
CPU scheduling hota hai	No multithreading
Truly parallel execution	No parallel execution

Agar start() nahi use karoge → thread kabhi create nahi hoga.

Full Example : Thread class extend karke

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Child Thread: " + i);
        }
    }
}
public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        for(int i = 1; i <= 5; i++) {
            System.out.println("Main Thread: " + i);
```

```
        }
    }
}
```

✓ Output random interleaved hoga (child + main mix)

✓ Isse proof milta hai ki **threads parallel run ho rahe hain**

2. Implementing Runnable interface

Industry me ye method prefer kiya jata hai.

Kyun?

Because Java **multiple inheritance allow nahi karta**, to:

- aap class ko dusre class se extend nahi kar paoge
- but `implements` multiple time allowed hai

Step 1 — Runnable implement karo

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}
```

Step 2 — Thread object me Runnable pass karo

```
Thread t = new Thread(new MyTask());
t.start();
```

✓ Runnable me thread create nahi hota

✓ Thread class hi thread banata hai

✓ Runnable sirf job deta hai

Full Example — Runnable Interface

```
class MyTask implements Runnable {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Child Thread: " + i);
        }
```

```
    }
}


public class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new MyTask());
        t.start();

        for(int i = 1; i <= 5; i++) {
            System.out.println("Main Thread: " + i);
        }
    }
}
```

Thread class vs Runnable interface

Extending Thread	Implementing Runnable
Easy	Professional way
Cannot extend another class	Can extend other classes
run() in subclass	run() in separate class
Tight coupling	Loose coupling (better design)
Not recommended	Recommended

Thread Naming, Thread Priority & Thread Scheduler

1. Thread Naming (VERY IMPORTANT)

Java me har thread ka ek naam hota hai.

Default thread names:

- Main thread → `"main"`
- New threads → `"Thread-0"`, `"Thread-1"` ...

A. Thread ka naam kaise set kare?

Method 1 — Constructor me

```
Thread t = new Thread("WorkerThread");
```

Method 2 — setName()

```
t.setName("Downloader");
```

B. Thread ka naam kaise get kare?

```
System.out.println(t.getName());
```

Full Example: Thread Name

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread Running: " +
Thread.currentThread().getName());
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setName("Worker-1");
        t1.start();

        System.out.println("Main Thread: " +
Thread.currentThread().getName());
    }
}
```

2. Thread Priority

Thread priority = Thread ko diya gaya **importance level**.

Priority Range:

```
1  (MIN_PRIORITY)
5  (NORM_PRIORITY) → default
10 (MAX_PRIORITY)
```

A. setPriority()

```
t.setPriority(Thread.MAX_PRIORITY);  // 10
t.setPriority(Thread.MIN_PRIORITY);  // 1
t.setPriority(7);
```

B. getPriority()

```
System.out.println(t.getPriority());
```

Priority ka effect kya hota hai?

Higher priority → Higher chance to run

GUARANTEE nahi hoti

Java thread scheduler OS ke hisaab se decide karta hai.

This is an important interview line:

"Priority increases the probability of execution, not the guarantee."

Example — Thread Priority

```
class Task extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() +
                           " Priority: " +
Thread.currentThread().getPriority());
    }
}

public class Test {
    public static void main(String[] args) {

        Task t1 = new Task();
        t1.setName("Low");
        t1.setPriority(1);

        Task t2 = new Task();
        t2.setName("High");
        t2.setPriority(10);

        t1.start();
        t2.start();
    }
}
```

3. Thread Scheduler (MOST IMPORTANT)

Thread Scheduler = OS + JVM ka program

jo decide karta hai kaun sa thread kab chalega.

Java me scheduler ke liye koi guarantee nahi hoti:

- Kaun sa thread pehle chalega?
- Kitna time chalega?
- Kaun sa thread pause hoga?

All depends on **OS + JVM + CPU load**

Interviewer asks: Is Java thread scheduling preemptive or time-sliced?

Answer:

"Depends on OS, no guarantee."

Summary

Topic	1-Line Interview Definition
Thread Name	Human-readable identifier
Thread Priority	Importance suggestion for scheduler
Scheduler	OS-based thread manager
Priority Guarantee?	NO

Thread.sleep(), Thread.yield(), Thread.join()

Yeh teen methods interview me **100% pooche jaate hain**.

Inme se join() sabse important hota hai.

1. Thread.sleep() — Pause the thread

`sleep()` thread ko **temporary rokh deta hai**.

```
Thread.sleep(milliseconds);
```

Example: 1 second ka delay

```
Thread.sleep(1000);
```

sleep() kya karta hai?

- Thread **running state → timed waiting state** me chala jata hai
- CPU dusre thread ko de diya jata hai
- Time complete hone ke baad thread wapas runnable state me aa jata hai

Example — sleep()

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(i);
            try { Thread.sleep(500); }  // 0.5 sec pause
            catch(Exception e) {}
        }
    }
```

```
}

public class Test {
    public static void main(String[] args) {
        new MyThread().start();
    }
}
```

✓ Output slow-slow print hoga

✓ UI apps me loading delay ke liye use hota hai

Interview Question on sleep()

Q: sleep() thread lock release karta hai?

Ans: ❌ **NO**

sleep() **lock release nahi karta**.

2. Thread.yield() — Give chance to other threads

Yield scheduler ko request karta hai:

“Mujhe abhi CPU mat do… kisi aur thread ko de do.”

But it is **not guaranteed**.

yield() kab use hota hai?

- Low priority thread wants to give chance to high priority thread
- Testing me
- Debugging me

Example — yield()

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName());
            Thread.yield();
        }
    }
```

```
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        for(int i = 1; i <= 5; i++) {
            System.out.println("Main");
        }
    }
}
```

Output random hoga
yield = “offer”, not “force”.

Interview Question

Q: Does yield guarantee other threads will run?

Ans: ❌ **NO**

3. Thread.join() — Wait for a thread to finish

join() =
“Main thread ruk ja — jab tak ye thread apna kaam complete nahi kar leta.”

join() syntax:

```
t.join(); // Main thread will wait for t to finish
```

Example — join()

```
class Worker extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println("Worker: " + i);
            try { Thread.sleep(500); } catch(Exception e) {}
```

```
        }
    }
}

public class Test {
    public static void main(String[] args) throws Exception {
        Worker w = new Worker();
        w.start();

        w.join();  // main waits here

        System.out.println("Main thread resumes...");
    }
}
```

Output order (always same):

```
Worker: 1
Worker: 2
Worker: 3
Worker: 4
Worker: 5
Main thread resumes...
```

join() **guarantee** deta hai
main thread tab tak wait karega

join(time) — Timed wait

```
w.join(2000);   // 2 seconds tak wait karo
```

Interview Table — sleep vs yield vs join

Method	Purpose	Guarantee?	Lock release?
sleep()	Pause thread	YES (time complete)	NO
yield()	Suggest others to run	NO	NO
join()	Wait for another thread	YES	NO

Synchronization (MOST IMPORTANT in Multithreading)

Iss topic se 100% interview me sawal aata hai.

Synchronization ka purpose hota hai:

Multiple threads ko ek shared resource ka access control dena
taaki data corrupt na ho.

1. Race Condition (Core Problem)

Race Condition tab hota hai jab:

- Do ya zyada threads
- Same data ko
- Same time par update/modify kar rahe ho

Aur output **random / wrong** aata hai.

Real Example (Bank Account)

Balance = 1000

Thread 1 → withdraw(500)
Thread 2 → withdraw(500)

Agar dono ek sath chale:

Expected balance = **0**
BUT incorrect output ho sakta hai = **500** OR **-500**

Isko prevent karne ke liye synchronization hota hai.

2. What is Synchronization?

Synchronization =
"Ek time par **sirf ek thread** shared resource ko access kare."
Java me synchronization kaise hota hai?

3 ways:

1. `synchronized` method
2. `synchronized` block
3. Static synchronization

3. Synchronized Method

Agar aap method ko synchronized bana do:

Ek time par ek hi thread method execute karega.

Example — Without synchronization (Wrong output)

```
class Counter {
    int count = 0;

    public void increment() {
        count++;
    }
}

class MyThread extends Thread {
    Counter c;
    MyThread(Counter c) { this.c = c; }

    public void run() {
        for(int i = 0; i < 1000; i++) {
            c.increment();
        }
    }
}

public class Test {
    public static void main(String[] args) throws
Exception {
        Counter c = new Counter();

        MyThread t1 = new MyThread(c);
        MyThread t2 = new MyThread(c);

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Count = " + c.count);
    }
}
```

Expected output: 2000

But actual can be: **1500, 1700, 1980... random**

Reason → Race condition

Fix using synchronized method

```
synchronized void increment() {
    count++;
}
```

Now output will ALWAYS be:

```
Final Count = 2000
```

4. Synchronized Block (Better & Faster)

Aap pura method lock nahi karna chahte
→ sirf critical part lock karna chahte ho.

Syntax:

```
synchronized(objectReference) {
    // critical section
}
```

Example

```
public void increment() {
    synchronized(this) {
        count++;
    }
}
```

✓ Faster
✓ Better control
✓ Less locking → better performance

5. Static Synchronization (MOST ASKED)

Agar resource static hai → lock bhi static hona chahiye.

```
static synchronized void show() { }
```

Static sync ka lock: **class-level lock**, object-level nahi.

6. Lock kya hota hai?

Java me har object + class ka ek lock hota hai.

- Synchronized method/block me enter karne ke liye
  → thread ko lock chahiye
- Exit karte hi lock release ho jata hai

7. Interview Table — Method vs Block

Feature	Synchronized Method	Synchronized Block
Lock	Entire method	Only required block
Performance	Slow	Faster
Usage	Easy	Recommended
Flexibility	Low	High

Inter-thread Communication (wait, notify, notifyAll)

Yeh multithreading ka **sabse important + sabse confusing** topic hai.
Interview me isse direct question, example aur theory har cheez poochi jaati hai.

Main tumhe is topic ko **real-world examples + deep explanation + proper code** ke saath samjhaunga.

1. What is Inter-thread Communication?

Jab do threads **ek dusre ka wait** karte hain, ya **signal** dete hain ki kaam ho gaya:
Isko inter-thread communication kehte hain.

Example (Real life):

Producer — Consumer Problem

- Producer → data banata hai
- Consumer → data use karta hai
- Dono ko ek-dusre ke signal ki zarurat hoti hai

- “Wait, data banta hoon”
- “OK, data mil gaya, ab tum aur banao!”

Java me yeh communication hota hai:

Using:

- `wait()`
- `notify()`
- `notifyAll()`

2. wait(), notify(), notifyAll() kis object ke lock par kaam karte hain?

Yeh **object ke intrinsic lock (monitor lock)** par kaam karte hain.
So yeh methods **Object class ke part** hain (Thread class ke nahi).

Interview Line (must say):

✓ “wait(), notify() and notifyAll() belong to Object class because they work on intrinsic object locks.”

3. wait() — Thread ko rok deta hai

`wait()` ka meaning:

👉 “Main abhi kaam nahi kar sakta. Mujhe dusre thread ke signal ka wait hai.”

IMPORTANT RULE:

wait() sirf **synchronized block/method ke andar** hi chal sakta hai.
Agar synchronized ke bahar chalaya →
→ `IllegalMonitorStateException`

4. notify() — Ek waiting thread ko jagata hai

- notify() **ek random waiting thread** ko wake-up karta hai
- But wake-up hone ke baad thread turant execute nahi karta
- Pehle lock release hona jaruri hai

5. notifyAll() — Saare waiting threads ko jagata hai

- notifyAll() **all waiting threads** ko wake up karta hai
- Lekin lock sirf ek ko milega
- Baaki queue me wait karenge

Producer–Consumer Example (Most Asked)

Goal:

Producer → values dalta hai
Consumer → values nikalta hai
Dono ko synchronize + wait/notify chahiye.

Full Example — Best Explanation

```
class Data {
    int value;
    boolean isProduced = false;

    synchronized void produce(int v) throws Exception {
        if(isProduced == true) {
            wait();     // wait until consumed
        }

        value = v;
        isProduced = true;
        System.out.println("Produced: " + value);

        notify();        // tell consumer to consume
    }

    synchronized int consume() throws Exception {
        if(isProduced == false) {
            wait();      // wait if nothing produced
        }

        System.out.println("Consumed: " + value);
        isProduced = false;

        notify();       // tell producer to produce more
```

```
        return value;
    }
}
```

Producer Thread

```
class Producer extends Thread {
    Data d;

    Producer(Data d) { this.d = d; }

    public void run() {
        int i = 1;
        while(true) {
            try {
                d.produce(i++);
                Thread.sleep(500);
            } catch(Exception e) {}
        }
    }
}
```

Consumer Thread

```
class Consumer extends Thread {
    Data d;

    Consumer(Data d) { this.d = d; }

    public void run() {
        while(true) {
            try {
                d.consume();
                Thread.sleep(500);
            } catch(Exception e) {}
        }
    }
}
```

Main Class

```
public class Test {
    public static void main(String[] args) {
        Data d = new Data();
        new Producer(d).start();
        new Consumer(d).start();
    }
}
```

OUTPUT (Always Correct)

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
...
```

Summary Table — wait, notify, notifyAll

Method	Meaning	Wakes Up
wait()	Thread waits & releases lock	Nobody
notify()	Wake up one waiting thread	Only one
notifyAll()	Wake up all waiting threads	All

Multithreading — Deadlock (Deep & Interview-Level Explanation)

Ye multithreading ka sabse **dangerous + interview favorite** topic hai.

Main tumhe deadlock ko itna clear kar dunga ki tum kisi bhi interview me 100% confidently answer doge.

1. Deadlock Kya Hota Hai? (Simple Definition)

Deadlock tab hota hai jab:

👉 **Do ya zyada threads ek dusre ka lock chhodne ka wait karte rehte hain**

👉 Aur koi bhi kabhi lock nahi chhodta

👉 Result = System freeze

Ek real life example:

- Person A ke paas **pen** hai, usse **book** chahiye
- Person B ke paas **book** hai, usse **pen** chahiye

Dono wait kar rahe hain

→ Kabhi kaam nahi hoga

→ Ye deadlock hai

2. Java Deadlock Example (Most Asked)

Ye example interview me 90% log galat batate. Tum sahi bata paoge.

```
class A {
    synchronized void methodA(B b) {
        System.out.println("Thread 1: methodA ke andar");
        try { Thread.sleep(100); } catch(Exception e) {}

        // Thread 1 waiting for B's lock
        b.last();
    }

    synchronized void last() {
        System.out.println("Inside A.last()");
    }
}
class B {
    synchronized void methodB(A a) {
        System.out.println("Thread 2: methodB ke andar");
        try { Thread.sleep(100); } catch(Exception e) {}

        // Thread 2 waiting for A's lock
```

```
        a.last();
    }

    synchronized void last() {
        System.out.println("Inside B.last()");
    }
}

public class DeadlockDemo {
    A a = new A();
    B b = new B();

    DeadlockDemo() {
        Thread t1 = new Thread(() -> a.methodA(b));
        Thread t2 = new Thread(() -> b.methodB(a));

        t1.start();
        t2.start();
    }

    public static void main(String[] args) {
        new DeadlockDemo();
    }
}
```

3. Explanation (Very Clear)

Thread 1:

- `a.methodA(b)` chala raha hai
- It means: **A ka lock** hold kiya hua
- End me `b.last()` call karega
- Means: **B ka lock chahiye**

Thread 2:

- `b.methodB(a)` chala raha hai
- It means: **B ka lock** hold kiya hua
- End me `a.last()` call karega
- Means: **A ka lock chahiye**

Final Situation:

Thread	Lock held	Lock needed
Thread 1	A	B

Thread 2	B	A

👉 Both waiting

👉 No one releases lock

👉 Program freeze

👉 **Deadlock**

4. Deadlock Kaise Identify Hota Hai?

👉 Program freeze ho jata hai

👉 CPU usage low

👉 Logging shows last executed line

👉 Thread dump (jstack) me “Found one Java-level deadlock” likha hota hai

5. Deadlock Prevention Techniques (Industry-Wise)

Ye sabse important part hai. Interviews me 100% poochte hain.

Always lock resources in same order

Example:

- Always lock A → then B
- Never lock B → then A

If ordering same hai → deadlock impossible

Use tryLock() (ReentrantLock)

```
if (lock1.tryLock()) {
    if (lock2.tryLock()) {
        // work
    } else {
        lock1.unlock();
    }
}
```

→ tryLock() timeout de sakta hai

→ Deadlock nahi hota

Avoid nested synchronized blocks

Nested locks = high chance of deadlock

Unko minimize karo

Use wait/notify carefully

Kabhi-kabhi wrong wait/notify logic bhi deadlock jaisa freeze create karti hai

6. Deadlock vs Livelock vs Starvation

Concept	Meaning
Deadlock	Threads wait forever for locks
Livelock	Threads keep changing state but no progress
Starvation	One thread never gets CPU (always preempted)

7. Interview-Perfect Answer (2 lines)

“Deadlock occurs when two threads hold locks in opposite order and wait for each other, resulting in permanent waiting. Prevent it by consistent lock ordering, avoiding nested locks, or using tryLock().”

Multithreading — Race Condition (Deep Detail + Interview-Level + Real Examples)

Deadlock ke baad sabse important topic = **Race Condition**.

Interview me ye hamesha pucha jata hai.

Chalo ise crystal-clear samajhte hain.

1. Race Condition Kya Hota Hai? (Simple Definition)

Race Condition tab hota hai jab:

👉 **Multiple threads simultaneously same shared data ko modify karte hain**

👉 Aur execution order unpredictable hota hai

👉 Result = **Wrong output**

Example real life:
Do log **Google Sheet** me same cell ko ek hi time edit kar rahe ho → final value random hogi.

2. Java Race Condition Example (Most Asked)

Is code ka output **1000 hona chahiye** par nahi hota.

```
class Counter {
    int count = 0;

    void increment() {
        count++;
    }
}

public class RaceDemo {
    public static void main(String[] args) throws
Exception {
        Counter c = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 500; i++) {
                c.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count = " + c.count);
    }
}
```

Expected output:

```
Final Count = 1000
```

Actual output:
Kabhi 950, kabhi 930, kabhi 980...
Random

3. Ye Race Condition Kyun Hoti Hai? (Perfect Explanation)

`count++;` actually 3 steps me hota hai:
1️⃣ Read count
2️⃣ count = count + 1
3️⃣ Write count back
Suppose:

- Thread 1 → value read = 10
- Thread 2 → value read = 10

Dono ne 10 read kiya → dono 11 write kar denge.
Correct result: **12**
Actual result: **11**
→ Data corrupt ho gaya
→ This is **Race Condition**

4. Solution 1: synchronized (Most common)

```
class Counter {
    int count = 0;

    synchronized void increment() {
        count++;
    }
}
```

Ab dono threads ek time me ek hi entry karenge
No race condition

5. Solution 2: AtomicInteger (Best for performance)

Industry me Atomic classes ka use hota hai.

```
import java.util.concurrent.atomic.AtomicInteger;

class Counter {
    AtomicInteger count = new AtomicInteger(0);
}
```

Increment:
`count.incrementAndGet();`

✓ Super fast
✓ Non-blocking
✓ Thread-safe
✓ Recommended in high-performance apps

6. synchronized VS AtomicInteger (Interview Table)

Feature	synchronized	AtomicInteger
Lock	Uses monitor lock	No lock
Speed	Slow	Fast
Context switching	Yes	No
Best for	Complex code	Simple counters

7. Race Condition vs Deadlock vs Livelock

Term	Meaning
Race Condition	Wrong output due to parallel modification
Deadlock	Threads stuck forever
Livelock	Threads keep working but no progress

Very common interview confusion — ab clear.

Perfect Interview Answer (2 Lines)

“Race condition occurs when multiple threads modify shared data simultaneously causing inconsistent results. It can be prevented using synchronized, volatile, or Atomic classes.”

Multithreading — `volatile` Keyword (Most Confusing + Most Asked Topic)

Ab hum **volatile** ko deep detail me padhenge — interviews me iska question hamesha trap hota hai.

1. volatile kya hota hai? (Simple Definition)

`volatile` keyword ensures:

Visibility Guarantee

Ek thread ne variable me change kiya →
Dusre thread ko **turant** dikhega.

No Caching

Variable **CPU cache** me store nahi hota
→ Hamesha **main memory** se read/write hota hai.

BUT volatile does NOT give atomicity

Volatile = **thread-safe nahi banata**
(Ye interview me trick question hota hai!)

2. Visibility Problem WITHOUT volatile

Suppose ek thread continuously loop chala raha hai:

```
class Task implements Runnable {
    boolean running = true;

    public void run() {
        while (running) {
        }
        System.out.println("Stopped!");
```

```
    }
}
```

Thread-1: `run()` loop me hai
Thread-2: `running = false;` set karta hai

BUT Thread-1 ko ye change pata hi nahi chalega!

Why?
Because Thread-1 ne `running` ka **cached copy** rakha hai.
Result → Infinite loop
Problem → **Visibility Issue**

3. Fix with volatile

```
volatile boolean running = true;
```

Ab:

- Thread-2 ne false kiya
- Thread-1 ko **turant** dikhega
- Loop stop ho jayega

4. volatile Deep Theory (Interview-Level)

Java Memory Model (JMM) fix karta hai:

✓ Happens-before relation
A volatile variable write →
all following reads **must** get updated value.

✓ No reordering allowed
Compiler + JVM + CPU
koi bhi instruction reorder nahi kar sakte.
Example without volatile:

```
ready = true;
number = 10;
```

CPU is order ko ulta kar sakta hai:

```
number = 10;
ready = true;
```

volatile lagate hi:
→ Reordering forbidden

5. volatile does NOT solve race conditions (VERY IMPORTANT)

Bahut log galat samajhte hain:
`volatile int count;`
↓ Ye **atomic nahi hota**
`count++` ka output **inconsistent** rahega
(same as race condition)

Kyun?

Because `count++` = 3 steps:

1. Read
2. Add
3. Write

volatile sirf **visibility** deta hai
atomicity nahi deta.

6. volatile VS synchronized

Feature	volatile	synchronized
Visibility	✓ Yes	✓ Yes
Atomicity	❌ No	✓ Yes
Locking	❌ No	✓ Has lock
Performance	Fast	Slow
Prevent Race Condition	❌ No	✓ Yes

Interview trap:
"Is volatile enough for thread safety?"
→ **No**.

7. volatile real-world examples

Example 1: Stop a thread safely

```
volatile boolean stop = false;
```

Example 2: Configuration flags

Feature ON/OFF
Logging enable/disable

Example 3: Double-checked locking (Singleton)

Must use volatile
(Interview favorite)

8. volatile with Singleton (Most asked)

```
class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {            // First check
            synchronized (Singleton.class) {
                if (instance == null) {    // Second check
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

If volatile nahi lagate →
CPU constructed object ka **half state** memory me publish kar deta hai →
Thread-safe nahi hota.

Final Interview Answer (2–3 Lines)

"volatile ensures visibility and prevents instruction reordering, but it does not provide atomicity. It is useful for flags, state checks, and double-checked locking in Singleton, but not for operations like count++."

Multithreading — Thread Pools (ExecutorService, Callable, Future)

Ye Java multithreading ka **most practical + industry-level** topic hai.
Almost **every real project** me thread pool use hota hai.
Aaj hum **Thread Pools** ko deep detail me samjhenge.

1. Why Thread Pool? (Real-world reason)

Agar hum naya thread baar-baar banaye:

- Thread creation cost = **very high**
- Memory zyada use hoti hai
- Context switching bahut hota hai
- System slow ho jata hai

👉 Isliye real projects me **Thread Pools** use hote hain.
Thread pool = Pehle se ready threads ka group
→ Kaam aate hi assign ho jata hai
→ Thread khatam hone ke baad reuse hota hai

2. ExecutorService — Thread Pool Manager

Thread pools banana ka main interface:

```
ExecutorService service = Executors.newFixedThreadPool(5);
```

Yeh 5 threads ka pool banata hai.

3. Types of Thread Pools (Most Asked)

✓ 1. Fixed Thread Pool

```
Executors.newFixedThreadPool(5);
```

→ 5 fixed threads
→ CPU-bound tasks

✓ 2. Cached Thread Pool

```
Executors.newCachedThreadPool();
```

→ Unlimited threads
→ I/O tasks, network calls, task jisme wait time zyada ho

✓ 3. Single Thread Executor

```
Executors.newSingleThreadExecutor();
```

→ Tasks sequentially run
→ Logging systems, order-sensitive tasks

✓ 4. Scheduled Thread Pool

```
Executors.newScheduledThreadPool(3);
```

→ Timer tasks
→ job scheduling
→ Periodic tasks

4. Thread Pool Basic Example (Runnable)

```
ExecutorService service = Executors.newFixedThreadPool(3);

Runnable task = () -> {
    System.out.println(Thread.currentThread().getName() +
" executing task");
};

for (int i = 1; i <= 5; i++) {
    service.submit(task);
}

service.shutdown();
```

Output example:

```
pool-1-thread-1 executing task
pool-1-thread-2 executing task
pool-1-thread-3 executing task
pool-1-thread-1 executing task
pool-1-thread-2 executing task
```

→ Same threads reused!

5. Callable vs Runnable (VERY IMPORTANT)

Feature	Runnable	Callable
Return value	❌ No	✓ Yes
Throws exception	❌ No	✓ Yes
Method	run()	call()

Example:

```
Callable<Integer> task = () -> {
    return 10 * 2;
};
```

6. Future — Get Value from Thread

```
Callable<Integer> task = () -> {
    Thread.sleep(1000);
    return 50;
};

ExecutorService service = Executors.newFixedThreadPool(2);

Future<Integer> future = service.submit(task);

System.out.println("Result: " + future.get());

service.shutdown();
```

`future.get()` → Thread ka result return karta hai
But note: **future.get() blocks until result is available**

7. ScheduledExecutorService (Timer Tasks)

```
ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(2);

scheduler.schedule(() -> {
    System.out.println("Run after 5 seconds");
}, 5, TimeUnit.SECONDS);
```

Repeat every second:

```
scheduler.scheduleAtFixedRate(
    () -> System.out.println("Every 1 sec"),
    0,
    1,
    TimeUnit.SECONDS
);
```

8. shutdown vs shutdownNow

Method	Meaning
shutdown()	Pool close karo, pending tasks complete hone do
shutdownNow()	Saare threads forcefully stop kar do

9. Thread Pool Best Practices (Industry-Level)

✓ Never create thread manually in enterprise apps
✓ Always use **ExecutorService**
✓ Fixed thread pool for CPU tasks
✓ Cached pool for I/O tasks
✓ NEVER forget `shutdown()`
✓ For returning values → prefer **Callable + Future**

Final Interview Answer (2 Lines)

"Thread Pool reuses threads to reduce creation cost. We create pools using ExecutorService, and for returning values we use Callable with Future."

Multithreading — Scheduled Thread Pool + ForkJoinPool + Parallel Streams

Is part ke baad **Multithreading complete** ho jayega.
Saare interview concepts covered ho jayenge.

1. Scheduled Thread Pool (Deep Detail)

Ye ek special thread pool hai jo **tasks ko delay** ya **periodically (repeat)** chalane ke kaam aata hai.
Use cases:

✓ Cron jobs
✓ Auto-backup
✓ Email notifications
✓ Repeat tasks
✓ Timer systems

1.1 ScheduledExecutorService create karna

```
ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(3);
```

3 threads jo scheduling handle karenge.

1.2 Run task after delay

```
scheduler.schedule(() -> {
    System.out.println("Run after 5 sec");
}, 5, TimeUnit.SECONDS);
```

1.3 Run task repeatedly — fixed rate

```
scheduler.scheduleAtFixedRate(
    () -> System.out.println("Every 2 sec"),
    0,                  // delay
    2, TimeUnit.SECONDS
);
```

fixedRate means:

- Start time important
- Delay count nahi hota
- Example: 0 sec → 2 sec → 4 sec → 6 sec

Agar task slow chalta hai → overlap ho sakta hai.

1.4 Run task repeatedly — fixed delay

```
scheduler.scheduleWithFixedDelay(
    () -> System.out.println("Delay of 3 sec after task
completes"),
    0,
    3,
    TimeUnit.SECONDS
);
```

fixedDelay means:
Always wait **after task completes**.

1.5 Which one to use?

Use Case	Use
Real-time scheduling	fixedRate
Repeat after finishing task	fixedDelay
Cron-like tasks	fixedRate
Heavy tasks	fixedDelay

2. ForkJoinPool (Advanced + Interview Topic)

Java 7 se introduce hua.
Ye “divide-and-conquer” parallelism ke liye bana hai.
Used in:
✓ Sorting algorithms
✓ Large computations
✓ Recursive tasks
✓ Parallel Streams

2.1 Fork/Join Concept

Divide big task → child tasks → run parallel → combine result
Example:
100000 numbers sum karna
→ Split into 2 tasks
→ Fir 4
→ Fir 8

→ Parallel run
→ Final result combine

2.2 RecursiveTask Example

```
class SumTask extends RecursiveTask<Integer> {
    int[] arr;
    int start, end;

    SumTask(int[] arr, int start, int end) {
        this.arr = arr;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= 10) {
            int sum = 0;
            for (int i = start; i < end; i++) sum +=
arr[i];
            return sum;
        }

        int mid = (start + end) / 2;

        SumTask left = new SumTask(arr, start, mid);
        SumTask right = new SumTask(arr, mid, end);

        left.fork();
        int rightResult = right.compute();
        int leftResult = left.join();

        return leftResult + rightResult;
    }
}
```

Run with ForkJoinPool:

```
ForkJoinPool pool = new ForkJoinPool();
int result = pool.invoke(new SumTask(arr, 0, arr.length));
```

2.3 ForkJoinPool Benefits

✓ Uses **work stealing** (idle thread steals task)
✓ Automatically balances load
✓ Very fast for recursive splitting tasks
✓ Core of **Parallel Streams**

3. Parallel Streams (Simple + Powerful)

Java 8 ne multithreading ko extremely easy banaya.
Example:

```
List<Integer> list = Arrays.asList(1,2,3,4,5);

list.parallelStream()
    .forEach(i -> System.out.println(i + " " +
Thread.currentThread().getName()));
```

Output (random threads):

```
1 ForkJoinPool.commonPool-worker-1
3 ForkJoinPool.commonPool-worker-3
5 ForkJoinPool.commonPool-worker-2
```

3.1 When to use parallel streams?

✓ Large data (big lists, arrays)
✓ CPU-heavy tasks
✓ Multi-core processors

3.2 When NOT to use parallel streams?

❌ Small datasets
❌ I/O tasks
❌ Database calls
❌ Tasks with shared mutable state (risk of race condition)

4. Difference: ExecutorService vs ForkJoinPool vs ParallelStream

Feature	ExecutorService	ForkJoinPool	Parallel Stream
Best for	general async tasks	recursive tasks	data processing
Thread type	Normal threads	Worker threads with work-stealing	ForkJoinPool.commonPool
Code complexity	Medium	High	Very easy

Multithreading Completed

You now know:

✓ Threads & lifecycle
✓ Thread creation
✓ Synchronization
✓ Locks
✓ Deadlock
✓ Race condition
✓ volatile
✓ Thread pools
✓ Callable/Future
✓ Scheduled executors
✓ ForkJoinPool
✓ Parallel Streams

Ye tamam topics interview me **directly pooche jate hain**.