Multithreading Level 1

3. Multithreading - Threads / Runnable

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,neccessary hinds, points to remember,usual mistakes,etc.

start lvl1

Perfect 🎯! Let's start **Level 1** of our **Multithreading - Threads / Runnable** learning game in Java.
We'll go step by step, increasing complexity just like in a game.

🕹️ Level 1: Introduction to Multithreading in Java

📖 What is Multithreading?

- **Thread** = A lightweight process, a unit of execution inside a program.
- **Multithreading** = Running multiple threads (tasks) simultaneously to improve performance.

💡 Example: Watching a YouTube video →

- One thread: playing video
- Another thread: playing audio
- Another thread: downloading buffer

All run *concurrently* to give a smooth experience.

✅ Ways to Create a Thread in Java

1. **Extending `Thread` class**
2. **Implementing `Runnable` interface**

🔹 Example 1: Extending Thread class

```java
class MyThread extends Thread {
    public void run() { // task performed by the thread
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " → " + i);
            try {
                Thread.sleep(500); // pause for 0.5 seconds
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }
}

public class ThreadExample1 {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start(); // starts a new thread (calls run())
        t2.start();
    }
}
```

🔹 Output (example, order may change because of multithreading)

```mathematica
Thread-0 → 1
Thread-1 → 1
Thread-0 → 2
Thread-1 → 2
...
```

⚡ The order is **not predictable**, that's the essence of multithreading.

🔹 Example 2: Implementing Runnable interface

```java
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " → " + i);
        }
    }
}

public class ThreadExample2 {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());

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

Both methods (`Thread` and `Runnable`) are valid. But **Runnable is preferred** in real-world applications because:

- Java allows **multiple inheritance via interfaces**, but you cannot extend two classes.
- Runnable separates **task** from **execution**.

🌍 Real-World Use Case

Imagine a **banking system**:

- One thread handles **ATM withdrawals**,
- Another thread handles **net banking transfers**,
- Another handles **SMS notifications**.

They all run **parallel**, ensuring faster response and smooth execution.

⚡ Hints / Points to Remember

- `start()` → starts a new thread and calls `run()` internally.
- `run()` → if called directly, runs like a normal method (NO new thread).
- `sleep(ms)` → makes the thread pause temporarily.
- Thread execution order is **not guaranteed** (depends on CPU scheduler).
- **Always use `try-catch` with `sleep()`** because it throws `InterruptedException`.

🚨 Usual Mistakes

1. ❌ Calling `t.run()` instead of `t.start()` → no multithreading happens.
2. ❌ Assuming output order will always be the same.
3. ❌ Forgetting exception handling with `sleep()`.
4. ❌ Using `Thread` class for everything instead of `Runnable`.

👉 That's **Level 1**: Basics of creating threads.

Would you like me to move to **Level 2: Thread Lifecycle & States (NEW, RUNNABLE, TIMED_WAITING, etc.)** with coding examples?

next

Awesome! 🚀 Let's move to **Level 2**. This level focuses on **Thread Lifecycle & States**. Understanding this is crucial to control how threads behave in real applications.

🕹️ Level 2: Thread Lifecycle & States

📖 Thread Lifecycle in Java

A thread can be in **one of these states**:

State	Description
NEW	Thread is created but `start()` not called yet.
RUNNABLE	Thread is ready to run and waiting for CPU time.
RUNNING	Thread is executing its `run()` method.
WAITING	Thread is waiting indefinitely for another thread to signal it.
TIMED_WAITING	Thread waits for a specified time (e.g., `sleep(ms)` or `join(ms)`).
TERMINATED	Thread has finished execution.

🔹 Example: Thread Lifecycle Demonstration

```java
class LifecycleDemo extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " → RUNNING");
        try {
            Thread.sleep(1000); // TIMED_WAITING
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println(Thread.currentThread().getName() + " → TERMINATED");
    }
}

public class ThreadLifecycleExample {
    public static void main(String[] args) throws InterruptedException {
        LifecycleDemo t1 = new LifecycleDemo();

        System.out.println("State after creation: " + t1.getState()); // NEW

        t1.start(); // Thread starts → RUNNABLE / RUNNING

        Thread.sleep(500); // Main thread sleeps

        System.out.println("State while running: " + t1.getState()); // RUNNABLE or
TIMED_WAITING

        t1.join(); // Main thread waits until t1 finishes

        System.out.println("State after completion: " + t1.getState()); // TERMINATED
    }
}
```

🔹 Sample Output

```pgsql
State after creation: NEW
Thread-0 → RUNNING
State while running: TIMED_WAITING
Thread-0 → TERMINATED
State after completion: TERMINATED
```

💡 Note: `getState()` helps you track the **current state** of a thread.

🌍 Real-World Use Case

Imagine a **restaurant ordering system**:

- **NEW**: New order received, thread created.
- **RUNNABLE**: Order is ready to be prepared, waiting for chef availability.
- **RUNNING**: Chef is actively preparing the dish.
- **TIMED_WAITING**: Dish is in the oven for a fixed time.
- **TERMINATED**: Order completed and delivered.

⚡ Points to Remember

- Threads must **start with `start()`**, otherwise state stays **NEW**.
- **`sleep(ms)` and `join(ms)`** cause **TIMED_WAITING**.
- `wait()` (from Object class) causes **WAITING**.
- **TERMINATED threads cannot be restarted**.

🚨 Usual Mistakes

1. ❌ Calling `start()` multiple times → throws `IllegalThreadStateException`.
2. ❌ Assuming RUNNING state is guaranteed immediately after `start()`.
3. ❌ Not handling `InterruptedException` in `sleep()` or `join()`.

Level 2 teaches you how threads **move through states**, which is critical for **synchronization** and **resource management**.

I can now take you to **Level 3: Synchronization & Thread Safety**, where we'll learn how to avoid data corruption when multiple threads access shared resources.

Do you want me to go to **Level 3**?

next

Perfect! ⚡ Let's level up to **Level 3**.
Here we enter the *real challenge zone* of multithreading: **Synchronization & Thread Safety**.

🕹️ Level 3: Synchronization in Java

📖 Why Synchronization?

When **multiple threads share the same resource**, there’s a risk of **data inconsistency** (called a **race condition**).

💡 Example: Two threads withdrawing money at the same time from the same bank account → balance might go negative if not controlled.

To solve this, we use **synchronization**.

✅ Synchronization Methods

1. **Synchronized Method** → Entire method is locked.
2. **Synchronized Block** → Only part of method is locked (more efficient).
3. **Static Synchronization** → Locks the *class* instead of the object.

🔹 Example 1: Problem Without Synchronization

```java
java

class Counter {
    int count = 0;

    public void increment() {
        count++; // Not thread-safe
    }
}

public class WithoutSync {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

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

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

        System.out.println("Final Count: " + counter.count);
    }
}
```

🔹 Output (Unexpected!)

```yaml
yaml

Final Count: 1875   (sometimes 2000, sometimes less!)
```

⚠️ Because both threads **modified count at the same time**.

🔹 Example 2: Using `synchronized` Method

```
java

class Counter {
    int count = 0;

    // synchronized method → thread-safe
    public synchronized void increment() {
        count++;
    }
}

public class WithSync {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

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

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

        System.out.println("Final Count: " + counter.count);
    }
}
```

🔹 Output (Correct!)

```
yaml

Final Count: 2000
```

🔹 Example 3: Synchronized Block (Better Performance)

```
java

class Bank {
    private int balance = 1000;

    public void withdraw(int amount) {
        synchronized (this) { // lock only this block
            if (balance >= amount) {
                balance -= amount;
                System.out.println(Thread.currentThread().getName() +
                                   " withdrew " + amount + ", balance: " + balance);
            } else {
                System.out.println("Insufficient balance for " +
Thread.currentThread().getName());
            }
        }
    }
}

public class BankExample {
    public static void main(String[] args) {
        Bank bank = new Bank();

        Thread t1 = new Thread(() -> bank.withdraw(700), "Thread-1");
        Thread t2 = new Thread(() -> bank.withdraw(700), "Thread-2");

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

✅ This ensures only **one thread can withdraw at a time**, avoiding race conditions.

🌍 Real-World Use Case

- **Ticket Booking System** → Two users shouldn't book the same seat.
- **Banking System** → Synchronization ensures correct balance updates.
- **File Writing** → Prevents two threads from corrupting data.

⚡ Points to Remember

- Use **synchronized blocks** instead of entire methods for better performance.
- Synchronization ensures **atomicity** (one thread at a time).
- But too much synchronization = **performance bottleneck**.
- `synchronized` works on **object-level lock** (instance methods) and **class-level lock** (static methods).

🚨 Usual Mistakes

1. ❌ Forgetting synchronization on shared resources → race conditions.
2. ❌ Overusing synchronization → slows down program.
3. ❌ Thinking synchronization guarantees order of execution → it only ensures **mutual exclusion**, not order.

✅ That's **Level 3**: You've learned how to make multithreaded code **safe** with synchronization.

👉 Next up, **Level 4: Inter-Thread Communication (`wait()`, `notify()`, `notifyAll()`)** – threads talking to each other (like producer-consumer problem).

Do you want me to move to **Level 4**?

next

Great 🎯! You're progressing like a pro. Let's move to **Level 4** where threads learn to **communicate with each other**.

🕹️ Level 4: Inter-Thread Communication (`wait()`, `notify()`, `notifyAll()`)

📖 Why Inter-Thread Communication?

In real applications, threads often need to **coordinate**.
Example:

- **Producer thread** → adds items to a buffer.
- **Consumer thread** → removes items from the buffer.
  👉 They must **talk** to avoid problems like overfilling or empty reading.

This is done using:

- `wait()` → thread pauses until another thread notifies it.
- `notify()` → wakes up one waiting thread.
- `notifyAll()` → wakes up all waiting threads.

⚠️ They must be called inside a `synchronized` block.

✅ Example 1: Simple Wait & Notify

```java
class SharedResource {
    private boolean available = false;

    public synchronized void produce() {
        System.out.println("Producer started...");
        available = true;
        System.out.println("Producer produced item.");
        notify(); // notify consumer
    }

    public synchronized void consume() {
        System.out.println("Consumer waiting for item...");
        while (!available) {
            try {
                wait(); // waits until producer notifies
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Consumer consumed item.");
    }
}

public class WaitNotifyExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread consumer = new Thread(resource::consume);
        Thread producer = new Thread(resource::produce);

        consumer.start();
        producer.start();
    }
}
```

🔹 Sample Output

```rust
Consumer waiting for item...
Producer started...
Producer produced item.
Consumer consumed item.
```

✅ Example 2: Producer-Consumer Problem

```java
import java.util.LinkedList;

class Buffer {
    private final LinkedList<Integer> list = new LinkedList<>();
    private final int capacity = 2;

    public synchronized void produce(int value) throws InterruptedException {
        while (list.size() == capacity) {
            wait(); // wait if buffer is full
        }
        list.add(value);
        System.out.println("Produced: " + value);
        notifyAll(); // notify consumers
    }

    public synchronized void consume() throws InterruptedException {
        while (list.isEmpty()) {
            wait(); // wait if buffer empty
        }
        int value = list.removeFirst();
        System.out.println("Consumed: " + value);
        notifyAll(); // notify producers
    }
}

public class ProducerConsumer {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                try {
                    buffer.produce(i);
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                try {
                    buffer.consume();
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        producer.start();
        consumer.start();
    }
}
```

🔹 Sample Output

```makefile
Produced: 1
Consumed: 1
Produced: 2
Produced: 3
Consumed: 2
Produced: 4
Consumed: 3
Consumed: 4
Produced: 5
Consumed: 5
```

🌍 Real-World Use Case

- **Messaging Apps** → Sender (producer) adds messages, Receiver (consumer) reads them.
- **OS Scheduling** → CPU (consumer) picks tasks, Processes (producer) submit tasks.
- **Restaurant** → Chef produces dishes, Waiter consumes (serves).

⚡ Points to Remember

- Always call `wait()`, `notify()`, `notifyAll()` inside a **synchronized block**.
- `wait()` releases the lock, `sleep()` does not.
- Use `while` (not `if`) with `wait()` to recheck condition after waking.
- `notify()` wakes up **one thread**, `notifyAll()` wakes **all waiting threads**.

🚨 Usual Mistakes

1. ❌ Calling `wait()` without synchronization → `IllegalMonitorStateException`.
2. ❌ Forgetting to recheck condition after `wait()`.
3. ❌ Using `if` instead of `while` → may cause spurious wake-ups.
4. ❌ Confusing `wait()` with `sleep()` → `wait()` releases lock, `sleep()` doesn't.

✅ That's **Level 4**: You've mastered inter-thread communication! 🎉

👉 Next up, we can go to **Level 5: Advanced Multithreading (Executors, ThreadPool, Callable & Future)** where we'll see how real-world enterprise apps handle hundreds of threads efficiently.

Do you want me to move to **Level 5**?

next

Awesome! 💥 You're leveling up fast. Now we enter **Level 5: Advanced Multithreading**, which is used in real-world enterprise applications to manage **hundreds or thousands of threads efficiently**.

🕹️ Level 5: Executors, ThreadPool, Callable & Future

📖 Why Advanced Thread Management?

Creating **hundreds of `Thread` objects** is expensive:

- High memory usage
- CPU scheduling overhead
- Difficult to manage lifecycle

Solution: Use **Executor Framework** and **Thread Pools**.

✅ 1️⃣ ExecutorService (Thread Pool)

`ExecutorService` manages a pool of threads and executes tasks efficiently.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        // Create a pool of 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit 5 tasks
        for (int i = 1; i <= 5; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println(Thread.currentThread().getName() + " is executing task " +
taskId);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        executor.shutdown(); // Stop accepting new tasks
    }
}
```

🔹 Sample Output

```cpp
pool-1-thread-1 is executing task 1
pool-1-thread-2 is executing task 2
pool-1-thread-3 is executing task 3
pool-1-thread-1 is executing task 4
pool-1-thread-2 is executing task 5
```

💡 Note: Only **3 threads** are used, tasks are queued efficiently.

✅ 2️⃣ Callable & Future

Unlike `Runnable`, **Callable** can **return a value** and throw exceptions.

```java
import java.util.concurrent.*;

public class CallableFutureExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Callable<Integer> task = () -> {
            Thread.sleep(1000);
            return 42; // returns result
        };

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

        System.out.println("Result from Callable: " + future.get()); // blocks until result is
ready

        executor.shutdown();
    }
}
```

🔹 Output

```python
Result from Callable: 42
```

🌍 Real-World Use Case

1. **Web Servers** → Each request handled by a thread from thread pool.
2. **Banking Transactions** → Multiple transactions executed simultaneously, results returned via `Future`.
3. **Batch Processing** → Large number of tasks distributed across thread pool for faster execution.

⚡ Points to Remember

- **ThreadPool** reduces **thread creation overhead**.
- Use `submit()` for tasks that return results (`Callable`).

- Use `execute()` for tasks that don't return results (`Runnable`).
- Always **shutdown** the executor to release resources.
- Future's `get()` method **blocks** until task completes.

🚨 Usual Mistakes

1. ❌ Creating a new `Thread` for every task in enterprise apps → memory and CPU heavy.
2. ❌ Forgetting to shutdown executor → thread leaks.
3. ❌ Using `Runnable` when you need a result → can't return value or catch exceptions easily.
4. ❌ Blocking on `Future.get()` unnecessarily → can reduce concurrency.

✅ That's **Level 5**: You now know how to manage threads **efficiently and safely** in real-world applications! 🌟

If you want, I can create a **Level 6: Advanced Tips & Best Practices in Multithreading**, including:

- Daemon threads
- Thread priorities
- Atomic variables
- Locks (`ReentrantLock`)
- Thread-safe collections

Do you want me to continue to **Level 6**?

next

Perfect! 🚀 Let's move to **Level 6: Advanced Tips & Best Practices in Multithreading**. This level is all about making your multithreaded code **robust, efficient, and safe** in real-world applications.

🕹️ Level 6: Advanced Multithreading Concepts

1️⃣ Daemon Threads

- **Daemon threads** run in the **background** and do not prevent the JVM from exiting.
- Used for **background tasks** like logging, monitoring, or cleanup.

```java
class DaemonThread extends Thread {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName() + " is running...");
            try { Thread.sleep(500); } catch (InterruptedException e) {}
        }
    }
}

public class DaemonExample {
    public static void main(String[] args) {
        DaemonThread t = new DaemonThread();
        t.setDaemon(true); // make it a daemon thread
        t.start();

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

💡 JVM **exits immediately** after the main thread ends, daemon thread stops automatically.

2️⃣ Thread Priorities

- Threads have priority **1 (MIN)** → **10 (MAX)**
- Higher priority threads are **more likely** to run first, but not guaranteed.

```java
Thread t1 = new Thread(() -> System.out.println("Thread 1"));
Thread t2 = new Thread(() -> System.out.println("Thread 2"));

t1.setPriority(Thread.MIN_PRIORITY); // 1
t2.setPriority(Thread.MAX_PRIORITY); // 10

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

⚠️ Priority is **OS-dependent**; it doesn't guarantee order.

3️⃣ Atomic Variables

- Use **`java.util.concurrent.atomic`** for **thread-safe operations** without locks.
- Example: **`AtomicInteger`** instead of **`int`** + **`synchronized`**.

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

public class AtomicExample {
    static AtomicInteger count = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) count.incrementAndGet();
        };

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

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

        System.out.println("Final Count: " + count.get()); // always 2000
    }
}
```

4️⃣ Locks (`ReentrantLock`)

- Alternative to `synchronized`, provides more **flexibility**: tryLock, timed lock, fairness.

```java
import java.util.concurrent.locks.ReentrantLock;

class Counter {
    int count = 0;
    ReentrantLock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

5️⃣ Thread-Safe Collections

- Use `Collections.synchronizedList()` or `ConcurrentHashMap` to avoid manual synchronization.

```java
import java.util.concurrent.ConcurrentHashMap;

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("A", 1); // thread-safe without explicit lock
```

🌍 Real-World Use Cases

- **Daemon Threads** → Background loggers, monitoring services.
- **Atomic Variables** → Counters in web apps, statistics.
- **Locks & Concurrent Collections** → Banking, inventory systems, real-time games.

⚡ Best Practices

1. Use **Executors & ThreadPools** instead of manually creating threads.

2. Prefer **Atomic variables** over synchronized counters.
3. Minimize **synchronized blocks** to reduce bottlenecks.
4. Use **Concurrent Collections** for shared data structures.
5. Always handle **InterruptedException** properly.

🚨 Usual Mistakes

1. ❌ Over-synchronization → slows down application.
2. ❌ Ignoring daemon threads → may cause JVM exit issues.
3. ❌ Using non-thread-safe collections → data corruption.
4. ❌ Forgetting to unlock `ReentrantLock` → deadlocks.

✅ That's **Level 6**: You now know advanced techniques to write **efficient, thread-safe, and scalable multithreaded Java applications**.

If you want, I can create a **Level 7: Multithreading Challenges & Mini Projects** to **practice everything with real coding problems**. This will solidify your knowledge like a game.

Do you want me to proceed to **Level 7**?

does i almost done with it learning it

Absolutely! 🎯

If we look at the **multithreading learning path we've covered** from Level 1 to Level 6:

Level	Topic Covered	Status
1	Basics: Threads & Runnable	✅ Done
2	Thread Lifecycle & States	✅ Done
3	Synchronization & Thread Safety	✅ Done
4	Inter-Thread Communication (`wait()`, `notify()`)	✅ Done
5	Advanced: ExecutorService, ThreadPool, Callable & Future	✅ Done
6	Best Practices: Daemon Threads, Atomic Variables, Locks, Thread-safe Collections	✅ Done

So, **yes**, you have covered **almost everything essential** in multithreading for learning and interview purposes. 💪

✅ What's "almost done" means:

- You **know how to create threads, synchronize, communicate, and manage them efficiently**.
- You understand **thread safety, locks, atomic operations, and executors**.

- You're ready for **real-world projects and interview questions** on multithreading.

⚡ Optional Next Steps

- **Level 7 (Practice / Mini Projects)**: Implement **producer-consumer, banking system, ticket booking, chat app**, etc.
- Learn **Deadlocks, Starvation, Livelocks** (common concurrency pitfalls).
- Solve **threading challenges** on platforms like LeetCode / HackerRank.

If you want, I can quickly outline **Level 7: Practice Challenges & Mini Projects** so you can consolidate your multithreading knowledge.

Do you want me to do that?

now i want realworld problems faced questions and solution

Perfect! Let's do a **real-world multithreading problems and solutions session**. I'll list **common issues**, why they happen, and **how to solve them with code examples**. This is exactly the kind of content that appears in **interviews and practical scenarios**.

🕹️ Real-World Multithreading Problems & Solutions

1️⃣ Problem: Race Condition

Scenario: Two threads update a shared variable at the same time → data corruption.

Example

```java
class Counter {
    int count = 0;

    public void increment() {
        count++; // Not thread-safe
    }
}

public class RaceConditionDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> { for (int i = 0; i < 1000; i++) counter.increment(); });
        Thread t2 = new Thread(() -> { for (int i = 0; i < 1000; i++) counter.increment(); });

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

        System.out.println("Final Count: " + counter.count); // may < 2000
    }
}
```

✅ Solution

Use **synchronized** or **AtomicInteger**.

```java
class CounterSafe {
    AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet(); // thread-safe
    }
}
```

Key Takeaway: Always protect **shared mutable state**.

2️⃣ Problem: Deadlock

Scenario: Two threads wait for each other's locks → program freezes.

Example

```java
class Resource {
    public synchronized void methodA(Resource r) {
        System.out.println(Thread.currentThread().getName() + " inside methodA");
        r.methodB(this);
    }

    public synchronized void methodB(Resource r) {
        System.out.println(Thread.currentThread().getName() + " inside methodB");
    }
}

public class DeadlockDemo {
    public static void main(String[] args) {
        Resource r1 = new Resource();
        Resource r2 = new Resource();

        Thread t1 = new Thread(() -> r1.methodA(r2), "Thread-1");
        Thread t2 = new Thread(() -> r2.methodA(r1), "Thread-2");

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

Problem: Thread-1 locks r1, Thread-2 locks r2 → both wait forever.

✅ Solution

- Lock resources in **same order**.
- Use `ReentrantLock.tryLock()` with timeout.

```java
ReentrantLock lock1 = new ReentrantLock();
ReentrantLock lock2 = new ReentrantLock();

if (lock1.tryLock(1, TimeUnit.SECONDS)) {
    if (lock2.tryLock(1, TimeUnit.SECONDS)) {
        // safe to operate
    }
}
```

Key Takeaway: Avoid circular wait, always acquire locks in consistent order.

3️⃣ Problem: Starvation

Scenario: Low-priority threads never get CPU time.

Example:

- Thread-1 priority MAX, Thread-2 priority MIN → Thread-2 may never run.

✅ Solution

- Avoid **extreme priority differences**.
- Use **fair locks** (ReentrantLock with `true` for fairness).

```java
ReentrantLock lock = new ReentrantLock(true); // fair lock
```

Key Takeaway: Fairness avoids thread starvation.

4️⃣ Problem: Producer-Consumer Synchronization

Scenario: Producer produces faster than consumer → buffer overflow.

Example

```java
class Buffer {
    int item;
    boolean available = false;

    public synchronized void produce(int i) throws InterruptedException {
        while (available) wait();
        item = i;
        available = true;
        notify();
    }

    public synchronized int consume() throws InterruptedException {
        while (!available) wait();
        available = false;
        notify();
        return item;
    }
}
```

Problem: Without `wait()` & `notify()`, threads either **busy-wait** or **crash**.

✅ Solution

Use `wait()` **and** `notify()` correctly (as shown above).

Key Takeaway: Always coordinate producer-consumer with proper **inter-thread communication**.

5️⃣ Problem: Thread Interruption

Scenario: Thread blocked on `sleep()` or `wait()` cannot be stopped gracefully.

Example

```java
Thread t = new Thread(() -> {
    try { Thread.sleep(5000); }
    catch (InterruptedException e) { System.out.println("Thread interrupted!"); }
});
t.start();
t.interrupt(); // interrupts the sleeping thread
```

Key Takeaway: Always handle `InterruptedException` and don't ignore it.

6️⃣ Problem: Performance Bottleneck Due to Over-Synchronization

Scenario: Entire method synchronized → reduces concurrency.

✅ Solution

- Use **synchronized block** instead of method.
- Use **Concurrent Collections** instead of manual synchronization.

```java
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("A", 1); // thread-safe without blocking entire method
```

7️⃣ Problem: Thread Leak in Executors

Scenario: Executor created but never shut down → memory leaks.

✅ Solution

```java
ExecutorService executor = Executors.newFixedThreadPool(5);
// submit tasks
executor.shutdown(); // stops accepting new tasks
executor.awaitTermination(5, TimeUnit.SECONDS); // waits for tasks to finish
```

✅ Summary of Key Solutions

Problem	Solution
Race Condition	Synchronize, Atomic Variables
Deadlock	Lock ordering, tryLock, avoid circular wait
Starvation	Fair locks, balanced thread priority
Producer-Consumer	wait(), notify()/notifyAll()
Thread Interruption	Handle InterruptedException
Over-Synchronization	Synchronized blocks, Concurrent Collections
Thread Leak	Always shutdown ExecutorService

💡 **Tip:** In real-world systems like **banking, messaging apps, web servers**:

- Multithreading problems happen frequently.
- Knowing **how to detect, debug, and solve** is key.

If you want, I can make a **set of 10+ real-world multithreading interview questions with answers and code**, which will **perfectly prepare you for interviews**.

Do you want me to do that next?