**Perfect! Let's simulate this buggy scenario using notify() instead of notifyAll() — so you can see for yourself how a deadlock can happen when one producer wakes another producer mistakenly.**

**🧪 Problem Setup**

* **Use 3 producer threads**
* **Use 1 consumer thread**
* **Use notify() (not notifyAll())**
* **Buffer size = 5**
* **Each producer produces 5 items**
* **Only 1 consumer is consuming slowly**

**🧨 Code: Deadlock Scenario with notify()**

**import java.util.LinkedList;**

**import java.util.Queue;**

**class Store {**

**private final Queue<Integer> buffer = new LinkedList<>();**

**private final int capacity = 5;**

**private int value = 1;**

**public synchronized void produce(String producerName) throws InterruptedException {**

**while (buffer.size() == capacity) {**

**System.out.println(producerName + " waiting (buffer full)");**

**wait();**

**}**

**buffer.add(value);**

**System.out.println(producerName + " produced: " + value++);**

**notify(); // ❌ can wake another producer accidentally**

**}**

**public synchronized void consume() throws InterruptedException {**

**while (buffer.isEmpty()) {**

**System.out.println("Consumer waiting (buffer empty)");**

**wait();**

**}**

**int item = buffer.poll();**

**System.out.println("Consumed: " + item);**

**notify(); // ❌ can wake another consumer (if multiple consumers)**

**}**

**}**

**public class DeadlockDemo {**

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

**Store store = new Store();**

**Runnable producerTask = () -> {**

**String name = Thread.currentThread().getName();**

**try {**

**for (int i = 0; i < 5; i++) {**

**store.produce(name);**

**Thread.sleep(200); // simulate some delay**

**}**

**} catch (InterruptedException e) {**

**Thread.currentThread().interrupt();**

**}**

**};**

**Runnable consumerTask = () -> {**

**try {**

**for (int i = 0; i < 15; i++) {**

**store.consume();**

**Thread.sleep(1000); // slow consumer**

**}**

**} catch (InterruptedException e) {**

**Thread.currentThread().interrupt();**

**}**

**};**

**// Start 3 producers**

**new Thread(producerTask, "Producer-1").start();**

**new Thread(producerTask, "Producer-2").start();**

**new Thread(producerTask, "Producer-3").start();**

**// Start 1 consumer**

**new Thread(consumerTask, "Consumer").start();**

**}**

**}**

**🧯 What Happens**

1. **Producers keep producing until buffer fills (5 items).**
2. **One producer tries to produce → hits wait() (buffer full).**
3. **Another producer is notified → wakes up, but still sees buffer full → waits again.**
4. **Cycle repeats → all producers waiting ☠️**
5. **Only consumer can help by consuming → but if timing is off, producers starve.**

**✅ Fix It Using notifyAll()**

**Just change both notify() calls to:**

**notifyAll();**

**This ensures:**

* **Any thread waiting — producer or consumer — has a chance to wake up**
* **Deadlock is avoided 💥**

**💡 Key Learning**

**🧠 When you have multiple producers or consumers, always prefer notifyAll() over notify() unless you're 100% sure who is waiting.**

**Want me to simulate the same thing with BlockingQueue or ReentrantLock and Condition to show the better way?**

Great question, bhai! Let’s clearly understand what happens when you use \*\*notifyAll()\*\* instead of notify() in a wait-notify scenario 👇

**🔁 notify() vs notifyAll() – Real Difference**

| **Behavior** | **notify()** | **notifyAll()** |
| --- | --- | --- |
| Notifies | Only **one random waiting thread** | **All waiting threads** on that monitor |
| Risk | Might wake up **same type of thread** (e.g., producer waking another producer) → 💀 deadlock | Everyone gets a chance, **deadlock avoided** |
| Use Case | When **only one thread** is waiting | When **multiple threads** are waiting (common case) |
| Example Problem | Multiple producers, one gets notified → it checks condition, still false → goes back to wait → nothing progresses | Everyone wakes up, checks condition, only **eligible** one proceeds |

**✅ What Happens When You Use notifyAll() in Your Code**

Let’s go back to your code logic:

public synchronized void produce(int value) throws InterruptedException {

while (buffer.size() == capacity) {

wait();

}

buffer.add(value);

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

notifyAll(); // 🔥 instead of notify()

}

public synchronized void consume() throws InterruptedException {

while (buffer.isEmpty()) {

wait();

}

int val = buffer.poll();

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

notifyAll(); // 🔥 instead of notify()

}

**🎯 Output Behavior with notifyAll()**

* If the buffer becomes full → all waiting producers are notified.
  + Only the one **who finds space in buffer** will proceed.
  + Others will check the condition → it's still full → go back to wait.
* If buffer becomes empty → all consumers are notified.
  + Only the **first** to see non-empty buffer consumes.
  + Others go back to wait.

So, even though **all threads wake up**, **only the eligible one proceeds**, making it **safe and deadlock-free**.

**🔍 Example Console Output with notifyAll():**

(Assuming 3 producers and 1 consumer)

Producer-1 produced: 1

Producer-2 produced: 2

Producer-3 produced: 3

Producer-1 produced: 4

Producer-2 produced: 5

Producer-3 waiting (buffer full)

Producer-1 waiting (buffer full)

Producer-2 waiting (buffer full)

Consumed: 1

Producer-3 produced: 6

...

Here, no one gets stuck. As soon as consumer consumes, **one waiting producer wakes up and continues**.

**🔐 Summary:**

* ✅ Use notifyAll() when you have **multiple producers/consumers**.
* ❌ notify() is risky in such cases — can cause **starvation** or **deadlock**.
* ✅ Always check the condition in a while() loop before proceeding (to avoid spurious wake-ups).

Do you want to try converting this to use **ReentrantLock + Condition** next? It’s a clean alternative to synchronized + wait/notify.