Great! Let's simulate your **Producer-Consumer problem** using **both approaches**—this will help you clearly see the benefits over using wait() and notify() manually.

**✅ Version 1: Using BlockingQueue (Best for simple use cases)**

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.BlockingQueue;

class Store {

BlockingQueue<Integer> buffer = new ArrayBlockingQueue<>(5); // capacity = 5

public void produce(int value) throws InterruptedException {

buffer.put(value); // waits if full

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

}

public void consume() throws InterruptedException {

int val = buffer.take(); // waits if empty

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

}

}

public class Factory {

public static void main(String[] args) {

Store store = new Store();

Thread producer = new Thread(() -> {

try {

for (int i = 1; i <= 10; i++) {

store.produce(i);

Thread.sleep(300); // Simulate delay

}

} catch (InterruptedException e) {}

});

Thread consumer = new Thread(() -> {

try {

for (int i = 1; i <= 10; i++) {

store.consume();

Thread.sleep(500); // Simulate delay

}

} catch (InterruptedException e) {}

});

producer.start();

consumer.start();

}

}

🧠 **Key Points**:

* No need to use synchronized, wait(), or notify().
* put() and take() automatically block/wait until space or item is available.
* Cleaner, safer, preferred for most use cases.

**✅ Version 2: Using ReentrantLock and Condition**

This gives you **maximum control**.

import java.util.LinkedList;

import java.util.Queue;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

class Store {

Queue<Integer> buffer = new LinkedList<>();

int capacity = 5;

Lock lock = new ReentrantLock();

Condition notFull = lock.newCondition();

Condition notEmpty = lock.newCondition();

public void produce(int value) throws InterruptedException {

lock.lock();

try {

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

notFull.await(); // wait until not full

}

buffer.add(value);

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

notEmpty.signal(); // notify consumer

} finally {

lock.unlock();

}

}

public void consume() throws InterruptedException {

lock.lock();

try {

while (buffer.isEmpty()) {

notEmpty.await(); // wait until not empty

}

int val = buffer.poll();

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

notFull.signal(); // notify producer

} finally {

lock.unlock();

}

}

}

public class Factory {

public static void main(String[] args) {

Store store = new Store();

Thread producer = new Thread(() -> {

try {

for (int i = 1; i <= 10; i++) {

store.produce(i);

Thread.sleep(300);

}

} catch (InterruptedException e) {}

});

Thread consumer = new Thread(() -> {

try {

for (int i = 1; i <= 10; i++) {

store.consume();

Thread.sleep(500);

}

} catch (InterruptedException e) {}

});

producer.start();

consumer.start();

}

}

🧠 **Key Points**:

* ReentrantLock gives you finer control than synchronized.
* Condition objects (notFull, notEmpty) separate the logic and reduce unnecessary wakeups.
* Useful when multiple producers and consumers are involved.

Let me know which one you want to explore deeper:

* Want me to **add multiple producers and consumers**?
* Or show a case where notify() causes deadlock but notifyAll() solves it?

Your call, bhai 🔥