Wait(), notify() – Producer Consumer in Java Thread – 2021-2022

Threre are two threads, one is producers which will produce and another thread is consumer which will consume.

public class Consumer implements Runnable {  
 private Store q;  
  
 public Consumer(Store q) {  
 this.q = q;  
 }  
  
 public void run() {  
 while (true) {  
 try {  
 Thread.*sleep*(1000);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 **q.get();**  
 }  
 }  
}

public class Producer implements Runnable {  
 private Store q;  
  
 public Producer(Store q) {  
 this.q = q;  
 }  
  
 public void run() {  
 int i = 0;  
 while (true) {  
 try {  
 Thread.*sleep*(1000);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 **q.put(i++);**  
 }  
 }  
}

public class Store {  
 private int n;  
 **private boolean valueSet = false;**  
  
 **synchronized** void get() {  
 **if (valueSet == false)** {  
 try {  
 System.*out*.println("Going to wait on get Method");  
 **wait();** } catch (InterruptedException e) {  
 System.*out*.println("InterruptedException…");  
 }  
 } 🡸 **End of If**  
 **System.*out*.println("Got: " + n);  
 valueSet = false;  
 notify();**  
 }  
  
 **synchronized** void put(int n) {  
 **if (valueSet == true)** {  
 try {  
 System.*out*.println("Going to wait on put Method");  
 **wait();** } catch (InterruptedException e) {  
 System.*out*.println("InterruptedException…");  
 }  
 } 🡸 **End of If**  
 **this.n = n;  
 valueSet = true;  
 System.*out*.println("Put: " + n);  
 notify();** }  
}

public class Test {  
 public static void main(String args[]) {  
 Store q = new Store();  
 Producer producer = new Producer(q);  
 Consumer consumer = new Consumer(q);  
 new Thread(producer, "Producer").start();  
 new Thread(consumer, "Consumer").start();  
 }  
}

Test program. 🡺

**Producer Consumer using Java 8 format**

public class TestPC1 {

    public void produce(Storage s) {

        for(int i = 0; i < 5; i++) {

            s.put(i);

        }

    }

    public void consume(Storage s) {

        for (int i = 0; i < 5; i++) {

            s.get();

        }

    }

    public void test() {

        Storage s = new Storage();

        Runnable pRunnable = () -> produce(s);

        Runnable cRunnable = () -> consume(s);

        new Thread(pRunnable).start();

        new Thread(cRunnable).start();

    }

    public static void main(String[] args) {

        new TestPC1().test();

    }

}

**Producer Consumer using BlockingQueue**

**import** java.util.concurrent.BlockingQueue;

**public** **class** **Producer** **extends** Thread {

**private** BlockingQueue<Integer> queue;

**public** Producer(BlockingQueue<Integer> queue) {

**this**.queue = queue;

}

**public** **void** run() {

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

System.***out***.println("Produced: " + i);

**try** {

queue.put(i);

Thread.*sleep*(1000);

} **catch** (InterruptedException ex) { ex.printStackTrace(); }

}

}

}

**import** java.util.concurrent.BlockingQueue;

**public** **class** **Consumer** **extends** Thread {

**private** BlockingQueue<Integer> queue;

**public** Consumer(BlockingQueue<Integer> queue) {

**this**.queue = queue;

}

**public** **void** run() {

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

**try** {

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

} **catch** (InterruptedException ex) { ex.printStackTrace(); }

}

}

}

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.LinkedBlockingQueue;

**public** **class** BlockingQueueDemo {

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

BlockingQueue<Integer> queue = **new** LinkedBlockingQueue<Integer>();

Producer produce = **new** Producer(queue);

Consumer consume = **new** Consumer(queue);

produce.start();

consume.start();

}

}

You can use LinkedBlockingQueue or ArrayBlockingQueue.

Reference : <http://stackoverflow.com/questions/9532923/producer-consumer-using-blocking-queue>

**Producer Consumer using Java 8**

**public class** TestBlockingQueue {  
  
 **public void** produce(BlockingQueue<Integer> bq) {  
 **for** (**int** i = 0; i < 5; i++) {  
 **try** {  
 bq.put(i);  
 System.***out***.println(**"PUT : "** + i);  
 } **catch** (InterruptedException e) { e.printStackTrace();}  
 }  
 }  
  
 **public void** consume(BlockingQueue<Integer> bq) {  
 **for** (**int** i = 0; i < 5; i++) {  
 Integer val = **null**;  
 **try** {  
 val = bq.take();  
 System.***out***.println(**"GOT : "** + val);  
 } **catch** (InterruptedException e) { e.printStackTrace();}  
 }  
 }  
  
 **public static void** main(String[] args) {  
 TestBlockingQueue tbq = **new** TestBlockingQueue();  
 BlockingQueue<Integer> bq = **new** LinkedBlockingQueue<>(1);  
 Runnable rp = () -> tbq.produce(bq);  
 Runnable cr = () -> tbq.consume(bq);  
 **new** Thread(rp).start();  
 **new** Thread(cr).start();  
 }  
}

**Producer and Consumer using Semaphore**

[**Semaphore**](http://www.javamadesoeasy.com/2015/03/semaphore-in-java.html) **on producer is created with permit =1**. So, that **producer can get the permit to produce**.

**Semaphore on consumer is created with permit =0**. So, that **consumer could wait for permit to consume**.

**Producer gets permit by** calling **semaphoreProducer.acquire()** and **starts producing**, **after producing** it calls **semaphoreConsumer.release()**. So, that **consumer could get the  permit to consume**.

|  |
| --- |
| **semaphoreProducer.acquire();**  **System.*out*.println("Produced : "+i);**  **semaphoreConsumer.release();** |

**Consumer gets permit by** calling **semaphoreConsumer.acquire()** and **starts consuming**, **after consuming** it calls **semaphoreProducer.release()**. So, that **producer could get the  permit to produce**.

|  |
| --- |
| **semaphoreConsumer.acquire();**  **System.*out*.println("Consumed : "+i);**  **semaphoreProducer.release();** |

**Producer**

**import** java.util.concurrent.Semaphore;  
**public class** Producer **implements** Runnable {  
  
 Semaphore **semaphoreProducer**;  
 Semaphore **semaphoreConsumer**;  
  
 **public** Producer(Semaphore semaphoreProducer, Semaphore semaphoreConsumer) {  
 **this**.**semaphoreProducer** = semaphoreProducer;  
 **this**.**semaphoreConsumer** = semaphoreConsumer;  
 }  
  
 **public void** run() {  
 **for** (**int** i = 1; i <= 5; i++) {  
 **try** {  
 **semaphoreProducer**.acquire();  
 System.***out***.println(**"Produced : "** + i);  
 Thread.*sleep*(1000);  
 **semaphoreConsumer**.release();  
  
 } **catch** (InterruptedException e) { e.printStackTrace(); }  
 }  
 }  
}

**Consumer**  
**public class** Consumer **implements** Runnable {  
  
 Semaphore **semaphoreConsumer**;  
 Semaphore **semaphoreProducer**;  
  
 **public** Consumer(Semaphore semaphoreConsumer, Semaphore semaphoreProducer) {  
 **this**.**semaphoreConsumer** = semaphoreConsumer;  
 **this**.**semaphoreProducer** = semaphoreProducer;  
 }  
  
 **public void** run() {  
  
 **for** (**int** i = 1; i <= 5; i++) {  
 **try** {  
 **semaphoreConsumer**.acquire();  
 System.***out***.println(**"Consumed : "** + i);  
 Thread.*sleep*(1000);  
 **semaphoreProducer**.release();  
 } **catch** (InterruptedException e) { e.printStackTrace(); }  
 }  
 }  
}

**ProducerConsumerTest.java**  
**import** java.util.concurrent.Semaphore;  
**public class** ProducerConsumerTest {  
  
 **public static void** main(String[] args) {  
 Semaphore semaphoreProducer = **new** Semaphore(1);  
 Semaphore semaphoreConsumer = **new** Semaphore(0);  
 System.***out***.println(**"semaphoreProducer permit=1 | semaphoreConsumer permit=0"**);  
  
 Producer producer = **new** Producer(semaphoreProducer, semaphoreConsumer);  
 Consumer consumer = **new** Consumer(semaphoreConsumer, semaphoreProducer);  
  
 Thread producerThread = **new** Thread(producer, **"ProducerThread"**);  
 Thread consumerThread = **new** Thread(consumer, **"ConsumerThread"**);  
  
 producerThread.start();  
 consumerThread.start();  
 }  
}

**Producer Consumer using Semaphore as per Java 8**

**public class** TestSemaphore {

**private Semaphore ps = new Semaphore(1);**

**private Semaphore cs = new Semaphore(0);**

**public void** consume() {  
 **try** {  
 **for**(**int** i = 0; i < 5; i++) {  
 **cs.acquire();**  
 System.***out***.println(**"GOT : "**+i);  
 **ps.release();**  
 }  
 } **catch**(Exception ex) {  
 ex.printStackTrace();  
 }  
 }

**public void** produce() {  
 **try** {  
 **for**(**int** i = 0; i < 5; i++) {  
 **ps.acquire();**  
 System.***out***.println(**"PUT : "**+i);  
 **cs.release();**  
 }  
 } **catch**(Exception ex) {  
 ex.printStackTrace();  
 }  
 }

**public void** test() {  
 Runnable p = () -> produce();  
 Runnable c = () -> consume();  
  
 **new** Thread(p).start();  
 **new** Thread(c).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestSemaphore().test();  
 }  
   
}

Producer Consumer using Reentrant Lock Condition

**public class** MyBuffer {  
  
 **private int data**;  
 **private** Lock **lock** = **new** ReentrantLock();  
 **private** Condition **full** = **lock**.newCondition();  
 **private** Condition **empty** = **lock**.newCondition();  
 **private boolean flag** = **false**;

**public void** put(**int** x) {  
 **lock**.lock();  
 **try** {  
 **if** (**flag** == **true**) {  
 **full**.await();  
 }  
 TimeUnit.***SECONDS***.sleep(1);  
 System.***out***.println(**"PUT : "** + x);  
 **this**.**data** = x;  
 **flag** = **true**;  
 **empty**.signal();  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
}

**public void** get() {  
 **lock**.lock();  
 **try** {  
 **if** (**flag** == **false**) {  
 **empty**.await();  
 }  
 TimeUnit.***SECONDS***.sleep(1);  
 System.***out***.println(**"GOT : "** + **this**.**data**);  
 **flag** = **false**;  
 **full**.signal();  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
}

}

**public class** ReentrantProducerConsumer {  
  
 **private** Storage **s** = **new** Storage();  
 **public void** produce() {  
 **for** (**int** i = 0; i < 5; i++)  
 **s**.put(i);  
 }  
 **public void** consume() {  
 **for** (**int** i = 0; i < 5; i++)  
 **s**.get();  
 }  
 **public void** test() {  
 Runnable rp = () -> produce();  
 Runnable cp = () -> consume();  
  
 **new** Thread(rp).start();  
 **new** Thread(cp).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** ReentrantProducerConsumer().test();  
 }  
}

The Buffer class can also be written in the following manner using Queue.

**public class** MyBuffer {  
  
 **private** Lock **lock** = **new** ReentrantLock();  
 **private** Condition **qFull** = **lock**.newCondition();  
 **private** Condition **qEmpty** = **lock**.newCondition();  
  
 **private** Queue<Integer> **queue** = **new** LinkedList<>();  
 **private int CAPACITY** = 1;

**public void** get() {  
 **lock**.lock();  
 **try** {  
 **while** (**queue**.size() == 0) {  
 TimeUnit.***SECONDS***.sleep(1);  
 **qEmpty.await();**  
 }  
 System.***out***.println(**"Got : "** + **queue**.poll());  
 **qFull.signal();**  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 }

**public void** put(**int** x) {  
 **lock**.lock();  
 **try** {  
 **while** (**queue**.size() == **CAPACITY**) {  
 TimeUnit.***SECONDS***.sleep(1);  
 **qFull.await();**  
 }  
 System.***out***.println(**"PUT : "** + x);  
 **queue**.offer(x);  
 **qEmpty.signal();**  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }

}

**Print Even and Odd using wait() and notify() (Correct and Better)**

**public class** EvenOddPrinter {  
 **private volatile boolean isOdd**;  
  
 **public synchronized void** printOdd(**int** val) {  
 **if** (**isOdd** == **false**) {  
 **try** {  
 wait();  
 TimeUnit.***SECONDS***.sleep(1);  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 }  
 System.***out***.println(**"Odd : "** + val);  
 **isOdd** = **false**;  
 notify();  
 }  
  
 **public synchronized void** printEven(**int** val) {  
 **if** (**isOdd** == **true**) {  
 **try** {  
 wait();  
 TimeUnit.***SECONDS***.sleep(1);  
 } **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 }  
 System.***out***.println(**"Even : "** + val);  
 **isOdd** = **true**;  
 notify();  
 }   
}

**Test Program**

EvenOddPrinter printer = **new** EvenOddPrinter();  
EvenThread even = **new** EvenThread(printer);  
OddThread odd = **new** OddThread(printer);  
even.start();  
odd.start();

**public class** EvenThread **extends** Thread {  
 **private** EvenOddPrinter **printer**;  
  
 **public** EvenThread(EvenOddPrinter printer) {  
 **this**.**printer** = printer;  
 }  
  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i < 10; i = i + 2) {  
 **printer**.printEven(i);  
 }  
 }  
}

**public class** OddThread **extends** Thread {  
 **private** EvenOddPrinter **printer**;  
  
 **public** OddThread(EvenOddPrinter printer) {  
 **this**.**printer** = printer;  
 }  
  
 @Override  
 **public void** run() {  
 **for** (**int** i = 1; i < 10; i = i + 2)  
 **printer**.printOdd(i);  
 }  
}

**Print Even and Odd using Semaphore**

import java.util.concurrent.Semaphore;  
public class **EvenOddPrinter1** {  
 private Semaphore semEven = new Semaphore(0);  
 private Semaphore semOdd = new Semaphore(1);  
  
 public void printEven(int number) {  
 try {  
 semEven.acquire();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 System.*out*.println("Even:" + number);  
 semOdd.release();  
 }  
  
 public void printOdd(int number) {  
 try {  
 semOdd.acquire();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 System.*out*.println("Odd:" + number);  
 semEven.release();  
 }  
}

public class **EvenOddTaskThread2** implements Runnable {  
 private EvenOddPrinter1 printer1;  
 private String oddEvenType;  
  
 public EvenOddTaskThread2(EvenOddPrinter1 printer1, String oddEvenType) {  
 this.printer1 = printer1;  
 this.oddEvenType = oddEvenType;  
 }  
  
 @Override  
 public void run() {  
 if (oddEvenType.equalsIgnoreCase("odd")) generateOddNumbers();  
 else generateEvenNumbers();  
 }  
  
 public void generateEvenNumbers() {  
 for (int i = 0; i < 10; i = i + 2) {  
 printer1.printEven(i);  
 }  
 }  
  
 public void generateOddNumbers() {  
 for (int i = 1; i < 10; i = i + 2) {  
 printer1.printOdd(i);  
 }  
 }  
}

public class **TestOddEven1** {  
 public static void main(String[] args) {  
 EvenOddPrinter1 printer1 = new EvenOddPrinter1();  
 Thread oddThread = new Thread(new EvenOddTaskThread2(printer1, "Odd"));  
 Thread evenThread = new Thread(new EvenOddTaskThread2(printer1, "Even"));  
 oddThread.start();  
 evenThread.start();  
 }  
}

**Blocking Queue Implementation using wait() and notify()**

wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ). notify() wakes up the first thread that called wait() on the same object.

For this I assume you're wanting to write a blocking queue implementation, where you have some fixed size backing-store of elements. The first thing you have to do is to identify the conditions that you want the methods to wait for. In this case, you will want the put() method to block until there is free space in the store, and you will want the take() method to block until there is some element to return.

public class BlockingQueue<T> {

private Queue<T> queue = new LinkedList<T>();

private int capacity;

public BlockingQueue(int capacity) {

this.capacity = capacity;

}

public synchronized void put(T element) throws InterruptedException {

while(queue.size() == capacity) { 🡸 You can also use **if** instead of **while**

wait();

}

queue.add(element);

notify();

}

public synchronized T take() throws InterruptedException {

while(queue.isEmpty()) { 🡸 You can also use **if** instead of **while**

wait();

}

T item = queue.remove();

notify();

return item;

}

}

**Firstly, you need to ensure that any calls to wait() or notify() are within a synchronized region of code (with the wait() and notify() calls being synchronized on the same object).** The reason for this (other than the standard thread safety concerns) is due to something known as a missed signal.