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

**Producer Consumer using Java 8 format**

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

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

**public class** TestPC {  
  
 **public void** produce(ProducerConsumer pc) {  
 **try** {  
 **for** (**int** i = 0; i < 5; i++)  
 pc.put(i);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** consume(ProducerConsumer pc) {  
 **try** {  
 **for** (**int** i = 0; i < 5; i++)  
 pc.get();  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 } **finally** {  
 }  
 }  
  
 **public void** check() {  
 ProducerConsumer pc = **new** ProducerConsumer();  
  
 **new** Thread(() -> produce(pc)).start();  
 **new** Thread(() -> consume(pc)).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestPC().check();  
 }  
}

You can use LinkedBlockingQueue or ArrayBlockingQueue.

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

**Producer Consumer using BlockingQueue in Java 8**

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

**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**.

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

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

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

**public class** TestSemaphore {  
 **private** Semaphore producerSemafore = **new** Semaphore(1);  
 **private** Semaphore consumerSemafore = **new** Semaphore(0);  
  
 **public void** produce() {  
 **try** {  
 **for** (**int** i = 0; i < 5; i++) {  
 **producerSemafore.acquire();**  
 System.***out***.println("PUT : " + i);  
 **consumerSemafore.release();** }  
 } **catch** (Exception ex) {  
 ex.printStackTrace();  
 }  
 }  
  
 **public void** consume() {  
 **try** {  
 **for** (**int** i = 0; i < 5; i++) {  
 **consumerSemafore.acquire();** System.***out***.println("GOT : " + i);  
 **producerSemafore.release();** }  
 } **catch** (Exception ex) {  
 ex.printStackTrace();  
 }  
 }  
  
 **public void** check() {  
 **new** Thread(() -> produce()).start();  
 **new** Thread(() -> consume()).start();  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestSemaphore().check();  
 }  
  
}

**Producer Consumer using Reentrant Lock Condition**

**public class** ProducerConsumerReentrant {  
 **private int** value;  
 **private** Lock lock = **new** ReentrantLock();  
 **private Condition full = lock.newCondition();  
 private Condition empty = lock.newCondition();**  
 **private boolean** flag = **false**;  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** put(**int** x) {  
 **lock.lock();** **try** {  
 **if** (flag == **true**) {  
 **full.await();** }  
 sleep(1);  
 System.***out***.println("PUT : " + x);  
 **this**.value = x;  
 flag = **true**;  
 **empty.signal();**  
 } **catch** (InterruptedException ie) { ie.printStackTrace(); }  
 **finally** {  
 **lock.unlock();**  
 }  
 }  
  
 **public void** get() {  
 **lock.lock();** **try** {  
 **if** (flag == **false**) {  
 **empty.await();**  
 }  
 sleep(1);  
 System.***out***.println("GOT : " + **this**.value);  
 flag = **false**;  
 **full.signal();** } **catch** (InterruptedException ie) { ie.printStackTrace(); }

**finally** {  
 **lock.unlock();**  
 }  
 }  
}

**public class** TestPCRenetrant {  
  
 **public void** produce(ProducerConsumerReentrant pc) {  
 **for** (**int** i = 0; i < 5; i++)  
 pc.put(i);  
 }  
 **public void** consume(ProducerConsumerReentrant pc) {  
 **for** (**int** i = 0; i < 5; i++)  
 pc.get();  
 }  
  
 **public void** check() {  
 ProducerConsumerReentrant pc = **new** ProducerConsumerReentrant();  
  
 **new Thread(() -> produce(pc)).start();  
 new Thread(() -> consume(pc)).start();**  
 }  
  
 **public static void** main(String[] args) {  
 **new** TestPCRenetrant().check();  
 }  
}

**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();  
 }  
 } 🡸 End of if   
 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();  
 }  
 } 🡸 End of if  
 System.***out***.println("Even : " + val);  
 isOdd = **true**;  
 **notify();** }  
}

**public static void** main(String[] args) {  
 **new** TestEvenOdd().check();  
 }

**public class** TestEvenOdd {  
  
 **public void** printEven(EvenOddPrinter printer) {  
 **for** (**int** i = 0; i < 10; i = i + 2) **printer.printEven(i);** }  
  
 **public void** printOdd(EvenOddPrinter printer) {  
 **for** (**int** i = 1; i < 10; i = i + 2) **printer.printOdd(i);**  
 }  
  
 **public void** check() {  
 EvenOddPrinter printer = **new** EvenOddPrinter();  
 **new** Thread( () -> printEven(printer)).start();  
 **new** Thread( () -> printOdd(printer)).start();  
 }  
  
}

**Print Even and Odd using Semaphore**

**public class** EvenOddPrinter {  
 **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 static void** main(String[] args) {  
 **new** TestEvenOdd().check();  
 }

**public class** TestEvenOdd {  
  
 **public void** sleep(**long** time) {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(time);  
 } **catch** (InterruptedException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public void** printEven(EvenOddPrinter printer) {  
 **for** (**int** i = 0; i < 10; i = i + 2) {  
 printer.printEven(i);  
 sleep(1);  
 }  
 }  
  
 **public void** printOdd(EvenOddPrinter printer) {  
 **for** (**int** i = 1; i < 10; i = i + 2) {  
 printer.printOdd(i);  
 sleep(1);  
 }  
 }  
  
 **public void** check() {  
 EvenOddPrinter printer = **new** EvenOddPrinter();  
 **new** Thread( () -> printEven(printer)).start();  
 **new** Thread( () -> printOdd(printer)).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.

**Blocking Queue Implementation using Condition**

public class BlockingQueue<T> {

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

private int capacity;

private Lock lock = new ReentrantLock();

private Condition notFull = lock.newCondition();

private Condition notEmpty = lock.newCondition();

public BlockingQueue(int capacity) {

this.capacity = capacity;

}

public void put(T element) throws InterruptedException {

lock.lock();

try {

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

notFull.await();

}

queue.add(element);

notEmpty.signal();

} finally {

lock.unlock();

}

}

public T take() throws InterruptedException {

lock.lock();

try {

while(queue.isEmpty()) {

notEmpty.await();

}

T item = queue.remove();

notFull.signal();

return item;

} finally {

lock.unlock();

}

}

}

Of course if you actually need a blocking queue, then you should use an implementation of the BlockingQueue interface. The wait() method causes the current thread to wait (possibly forever) until another thread notifies it of a condition change. You use wait() in conjunction with notify() to coordinate the activities of multiple threads using the same resources.