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…");  
 }  
 }  
 **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…");  
 }  
 }  
 **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. 🡺

**Another Example on Producer and Consumer using Queue**

import java.util.concurrent.TimeUnit;  
class **MyQueue** {  
 private int contents;  
 private **boolean available = false;**  
   
 public synchronized int **consume**() {  
 **while (!available)** {  
 try {  
 TimeUnit.*SECONDS*.sleep(1);  
 **wait();**  
 System.*out*.println("waited inside consume");  
 } catch (InterruptedException e) {}  
 }  
 System.*out*.println("Inside consume ...., going to notify()");  
 available = false;  
 System.*out*.println("Consumer has already consumed "+contents+" , now turn for producer");**notifyAll(); *// notifies the Producer thread***  
 return contents;  
 }  
  
 public synchronized void **produce**(int value) {  
 **while (available)** {  
 try {  
 TimeUnit.*SECONDS*.sleep(1);  
 **wait();** System.*out*.println("waited inside produce");  
 } catch (InterruptedException e) { }   
 }  
 System.*out*.println("Inside produce ...., going to notify()");  
 contents = value;  
 available = true;  
 System.*out*.println("Producer has already produced "+contents+" , now turn for Consumer");**notifyAll(); *//notifies the consumer thread***  
 }  
}

class **Consumer** extends Thread {  
 private MyQueue queue;  
 private int number;  
   
 public Consumer(MyQueue c, int number) {  
 queue = c;  
 this.number = number;  
 }  
 public void run() {  
 int value = 0;  
 for (int i = 0; i < 5; i++) {  
 **value = queue.consume();**  
*//("Consumer #" + this.number + " got: " + value);* }  
 }  
}

import java.util.concurrent.TimeUnit;  
class **Producer** extends Thread {  
 private MyQueue queue;  
 private int number;  
 public Producer(MyQueue c, int number) {  
 queue = c;  
 this.number = number;  
 }   
 public void run() {  
 for (int i = 0; i < 5; i++) {  
 **queue.produce(i);**  
*//("Producer #" + this.number + " put: " + i);* try {  
 TimeUnit.*SECONDS*.sleep(1);  
 } catch (InterruptedException e) { }  
 }   
 }  
}

public class ProducerConsumerTest {  
 public static void main(String[] args) {  
 MyQueue queue = new MyQueue();  
 Producer p1 = new Producer(queue, 1);  
 Consumer c1 = new Consumer(queue, 1);  
 p1.start(); c1.start(); }   
 } }

**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 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.java**

**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.java**  
**import** java.util.concurrent.Semaphore;  
**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();  
 }  
}

**Example Scenario**

Let us see the game of PingPong. In this game Alice and Bob will play alternately.

public class **PingPong** {  
 *// state variable identifying whose turn it is.* private String whoseTurn = null;  
  
 public synchronized boolean hit(String opponent) {  
 String x = Thread.*currentThread*().getName();  
 **if (whoseTurn == null) {  
 whoseTurn = x;  
 return true;  
 }** if (x.compareTo(whoseTurn) == 0) {  
 System.*out*.println("PING! (" + x + ")");  
 whoseTurn = opponent;  
 notifyAll();  
 } else {  
 try {  
 long t1 = System.*currentTimeMillis*();  
 Thread.*sleep*(1500);  
 wait(2500);  
 if ((System.*currentTimeMillis*() - t1) > 2500) {  
 System.*out*.println("\*\*\*\*\*\* TIMEOUT! " + x +  
 " is waiting for " + whoseTurn + " to play.");  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 return true; *// keep playing.* }  
}

public class **Player** implements Runnable {  
 PingPong myTable; *// Table where they play* String myOpponent;  
  
 public Player(String opponent, PingPong table) {  
 myTable = table;  
 myOpponent = opponent;  
 }  
  
 public void run() {  
 while (true) {  
 myTable.hit(myOpponent);  
 }  
 }  
}

public class **Game** {  
  
 public static void main(String args[]) {  
 PingPong table = new PingPong();  
 *//Alice will play with Bob* **Thread alice = new Thread(new Player("bob", table));**  
 *//Bob will play with Alice* **Thread bob = new Thread(new Player("alice", table));**  
 alice.setName("alice");  
 bob.setName("bob");  
 alice.start(); *// alice starts playing* bob.start(); *// bob starts playing* try {  
 *// Wait 5 seconds* Thread.*currentThread*().*sleep*(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 table.hit("DONE"); *// cause the players to quit their threads.* try {  
 Thread.*currentThread*().*sleep*(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**A ping pong game can be written in the following manner**

public class PingPongGame {  
 private boolean hasPlayed = false;  
  
 public synchronized void **ping()** {  
 while (hasPlayed == false) {  
 try {  
 wait();  
 } catch (Exception ex) { ex.printStackTrace();}  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + " Played : Ping");  
 hasPlayed = false;  
 notify();  
 }  
  
 public synchronized void **pong()** {  
 while (hasPlayed) {  
 try {  
 wait();  
 } catch (Exception ex) { ex.printStackTrace(); }  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + " Played : Pong");  
 hasPlayed = true;  
 notify();  
 }  
}

public class Player1 implements Runnable {  
 private PingPongGame game;  
 private String name;  
 public Player1(PingPongGame game, String name) {  
 this.game = game;  
 this.name = name;  
 }  
  
 @Override  
 public void run() {  
 Thread.*currentThread*().setName(name);  
 for (int i = 0; i < 5; i++) {  
 game.ping();  
 }  
 }  
}

public class Player2 implements Runnable {  
 private PingPongGame game;  
 private String name;  
 public Player2(PingPongGame game, String name) {  
 this.game = game;  
 this.name = name;  
 }  
  
 @Override  
 public void run() {  
 Thread.*currentThread*().setName(name);  
 for (int i = 0; i < 5; i++) {  
 game.pong();  
 }  
 }  
}

public class Test {  
 public static void main(String[] args) {  
 PingPongGame game = new PingPongGame();  
 Thread t1 = new Thread(

new Player1(game, "Alice"));  
 Thread t2 = new Thread(

new Player2(game, "Bob"));  
 t1.start();  
 t2.start();  
 }  
}

**Example Scenario**

**Write a program with 2 two threads, one prints odd numbers and the other prints even numbers upto 100?**

class **Printer** {  
 **private volatile boolean isOdd;**  
  
 **synchronized void printEven**(int number) {  
 **while (!isOdd)** {  
 try {  
 **wait();** } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
 **System.*out*.println(Thread.*currentThread*().getName() + ":" + number);  
 isOdd = false;  
 notify();** }  
  
 **synchronized void printOdd**(int number) {  
 **while (isOdd) {**  
 try {  
 **wait();**  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
 **System.*out*.println(Thread.*currentThread*().getName() + ":" + number);  
 isOdd = true;  
 notify();**  
 }  
}

class TaskEvenOdd implements Runnable {  
 private int max;  
 private Printer print;  
 private boolean isEvenNumber;  
  
 public TaskEvenOdd(Printer print, int max, boolean isEvenNumber) {  
 this.max = max;  
 this.print = print;  
 this.isEvenNumber = isEvenNumber;  
 }  
 @Override  
 public void run() {  
 **int number = isEvenNumber ? 2 : 1;**  
 while (number <= max) {  
 if (isEvenNumber) {  
 **print.printEven(number);**  
 } else {  
 **print.printOdd(number);**  
 }  
 **number += 2;**  
 }  
 }  
}

public class **TestEvenOdd** {  
 public static void main(String... args) {  
 Printer print = new Printer();  
 Thread oddThread = new Thread(new TaskEvenOdd(print, 10, false),"Odd");  
 Thread evenThread = new Thread(new TaskEvenOdd(print, 10, true),"Even");  
 oddThread.start();  
 evenThread.start();  
 }  
}

**Print Even Odd in another way**

public class EvenOddPrinter {  
 private boolean isOdd = false;  
  
 public synchronized void printEven(int number) {  
 while (isOdd == false) {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("Even:" + number);  
 isOdd = false;  
 notify();  
 }  
  
 public synchronized void printOdd(int number) {  
 while (isOdd == true) {  
 try {  
 wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("Odd:" + number);  
 isOdd = true;  
 notify();  
 }  
}

public class EvenOddThread implements Runnable {  
 private EvenOddPrinter printer;  
 private String oddEvenType;  
  
 public EvenOddThread(EvenOddPrinter printer, String oddEvenType) {  
 this.printer = printer;  
 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) {  
 printer.printEven(i);  
 }  
 }  
  
 public void generateOddNumbers() {  
 for (int i = 1; i < 10; i = i + 2) {  
 printer.printOdd(i);  
 }  
 }  
}

public class TestOddEven {  
 public static void main(String[] args) {  
 EvenOddPrinter printer = new EvenOddPrinter();  
 Thread oddThread = new Thread(

new EvenOddThread(printer, "Odd"));  
 Thread evenThread = new Thread(

new EvenOddThread(printer, "Even"));  
 oddThread.start();  
 evenThread.start();  
 }  
}

**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();  
 }  
}

**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);  
 }  
}

**Test Program**

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

**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) {

wait();

}

queue.add(element);

notify();

}

public synchronized T take() throws InterruptedException {

while(queue.isEmpty()) {

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.

**Example Scenario**

Write a multi-threaded Java program in which, one thread generates odd numbers and write to a pipe and the second thread generates even numbers and write to another pipe, and a third thread receives the numbers from both the pipes and evaluates if the sum is multiples of 5?

import java.io.IOException;  
import java.io.PipedWriter;  
  
public class NumberWriter extends Thread {  
  
 private PipedWriter writer;  
 private int maxNumber;  
 private boolean isEvenNumber;  
  
 public NumberWriter(PipedWriter writer, int maxNumber, boolean isEvenNumber) {  
 this.writer = writer;  
 this.maxNumber = maxNumber;  
 this.isEvenNumber = isEvenNumber;  
 }  
  
 public void run() {  
 int i = 1;  
 while (i <= maxNumber) {  
 try {  
 if (isEvenNumber && (i % 2) == 0) {  
 writer.write(i);  
 } else if (!isEvenNumber && i % 2 != 0) {  
 writer.write(i);  
 }  
 ++i;  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 public static void main(String[] args) {  
 final int MAX\_NUM = 100;  
  
 PipedWriter oddNumberWriter = new PipedWriter();  
 PipedWriter evenNumberWriter = new PipedWriter();  
  
 NumberWriter oddGen = new NumberWriter(oddNumberWriter, MAX\_NUM, false);  
 NumberWriter evenGen = new NumberWriter(evenNumberWriter, MAX\_NUM, true);  
 NumberReceiver receiver = new NumberReceiver(oddNumberWriter, evenNumberWriter);  
  
 oddGen.start();  
 evenGen.start();  
 receiver.start();  
 }  
}

import java.io.IOException;  
import java.io.PipedReader;  
import java.io.PipedWriter;  
  
public class NumberReceiver extends Thread {  
  
 private PipedReader oddReader;  
 private PipedReader evenReader;  
  
 public NumberReceiver(PipedWriter oddWriter, PipedWriter evenWriter) {  
 try {  
 this.oddReader = new PipedReader(oddWriter);  
 this.evenReader = new PipedReader(evenWriter);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void run() {  
 int odd = 0, even = 0;  
  
 try {  
 while (odd != -1) {  
 odd = oddReader.read();  
 even = evenReader.read();  
  
 if ((odd + even) % 5 == 0) {  
 System.*out*.println("match found " + odd + " + " + even + " = " + (odd + even));  
 }  
 }  
 } catch (IOException e) {  
 System.*exit*(1);  
 }  
 }  
}