# How to work with wait(), notify() and notifyAll() in Java?

<https://howtodoinjava.com/java/multi-threading/wait-notify-and-notifyall-methods/>

[**Java concurrency**](https://howtodoinjava.com/java-concurrency-tutorial/) is pretty complex topic and requires a lot of attention while writing application code dealing with multiple threads accessing one/more shared resources at any given time. Java 5, introduced some classes like [**BlockingQueue**](https://howtodoinjava.com/java/multi-threading/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/) and **Executors** which take away some of the complexity by providing easy to use APIs.

Programmers using concurrency classes will feel a lot more confident than programmers directly handling synchronization stuff using **wait()**, **notify()** and **notifyAll()** method calls. I will also recommend to use these newer APIs over synchronization yourself, BUT many times we are required to do so for various reasons e.g. maintaining legacy code. A good knowledge around these methods will help you in such situation when arrived.

In this tutorial, I am discussing the **purpose of wait() notify() notifyall() in Java**. We will understand the **difference between wait and notify**.

## **1. What are wait(), notify() and notifyAll() methods?**

The Object class in Java has three final methods that allow threads to communicate about the locked status of a resource.

#### wait()

It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method. The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism.

In other words, it is not possible for us to implement the wait() method purely in Java. It is a **native method**.

General syntax for calling wait() method is like this:

|  |
| --- |
| wait() method syntax |
| synchronized( lockObject )  {      while( ! condition )      {          lockObject.wait();      }        //take the action here;  } |

#### notify()

It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier’s synchronized block has completed.

So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

|  |
| --- |
| notify() method syntax |
| synchronized(lockObject)  {      //establish\_the\_condition;        lockObject.notify();        //any additional code if needed  } |

#### notifyAll()

It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify()method above.

General syntax for calling notify() method is like this:

|  |
| --- |
| notifyAll() method syntax |
| synchronized(lockObject)  {      establish\_the\_condition;        lockObject.notifyAll();  } |

In general, a thread that uses the wait() method confirms that a condition does not exist (typically by checking a variable) and then calls the wait() method. When another thread establishes the condition (typically by setting the same variable), it calls the notify() method. The wait-and-notify mechanism does not specify what the specific condition/ variable value is. It is on developer’s hand to specify the condition to be checked before calling wait() or notify().

## 2. How to use with wait(), notify() and notifyAll() methods

In this exercise, we will solve **producer consumer problem** using wait() and notify() methods. To keep program simple and to keep focus on usage of wait() and notify() methods, we will involve only one producer and one consumer thread.

Other features of the program are :

* Producer thread produce a new resource in every 1 second and put it in ‘taskQueue’.
* Consumer thread takes 1 seconds to process consumed resource from ‘taskQueue’.
* Max capacity of taskQueue is 5 i.e. maximum 5 resources can exist inside ‘taskQueue’ at any given time.
* Both threads run infinitely.

#### 2.1. Producer Thread

Below is the code for producer thread based on our requirements :

|  |
| --- |
| Producer.java |
| class Producer implements Runnable  {     private final List<Integer> taskQueue;     private final int           MAX\_CAPACITY;       public Producer(List<Integer> sharedQueue, int size)     {        this.taskQueue = sharedQueue;        this.MAX\_CAPACITY = size;     }       @Override     public void run()     {        int counter = 0;        while (true)        {           try           {              produce(counter++);           }           catch (InterruptedException ex)           {              ex.printStackTrace();           }        }     }       private void produce(int i) throws InterruptedException     {        synchronized (taskQueue)        {           while (taskQueue.size() == MAX\_CAPACITY)           {              System.out.println("Queue is full " + Thread.currentThread().getName() + " is waiting , size: " + taskQueue.size());              taskQueue.wait();           }             Thread.sleep(1000);           taskQueue.add(i);           System.out.println("Produced: " + i);           taskQueue.notifyAll();        }     }  } |

* Here “produce(counter++)” code has been written inside infinite loop so that producer keeps producing elements at regular interval.
* We have written the produce() method code following the general guideline to write wait()method as mentioned in first section.
* Once the wait() is over, producer add an element in taskQueue and called notifyAll() method. Because the last-time wait() method was called by consumer thread (that’s why producer is out of waiting state), consumer gets the notification.
* Consumer thread after getting notification, if ready to consume the element as per written logic.
* Note that both threads use sleep() methods as well for simulating time delays in creating and consuming elements.

#### 2.2. Consumer Thread

Below is the code for consumer thread based on our requirements :

|  |
| --- |
| Consumer.java |
| class Consumer implements Runnable  {     private final List<Integer> taskQueue;       public Consumer(List<Integer> sharedQueue)     {        this.taskQueue = sharedQueue;     }       @Override     public void run()     {        while (true)        {           try           {              consume();           } catch (InterruptedException ex)           {              ex.printStackTrace();           }        }     }       private void consume() throws InterruptedException     {        synchronized (taskQueue)        {           while (taskQueue.isEmpty())           {              System.out.println("Queue is empty " + Thread.currentThread().getName() + " is waiting , size: " + taskQueue.size());              taskQueue.wait();           }           Thread.sleep(1000);           int i = (Integer) taskQueue.remove(0);           System.out.println("Consumed: " + i);           taskQueue.notifyAll();        }     }  } |

* Here “consume()” code has been written inside infinite loop so that consumer keeps consuming elements whenever it finds something in taskQueue.
* Once the wait() is over, consumer removes an element in taskQueue and called notifyAll()method. Because the last-time wait() method was called by producer thread (that’s why producer is in waiting state), producer gets the notification.
* Producer thread after getting notification, if ready to produce the element as per written logic.

#### 2.3. Test producer consumer example

Now lets test producer and consumer threads.

|  |
| --- |
| ProducerConsumerExampleWithWaitAndNotify.java |
| public class ProducerConsumerExampleWithWaitAndNotify  {     public static void main(String[] args)     {        List<Integer> taskQueue = new ArrayList<Integer>();        int MAX\_CAPACITY = 5;        Thread tProducer = new Thread(new Producer(taskQueue, MAX\_CAPACITY), "Producer");        Thread tConsumer = new Thread(new Consumer(taskQueue), "Consumer");        tProducer.start();        tConsumer.start();     }  } |

Program Output.

|  |
| --- |
| Console |
| Produced: 0  Consumed: 0  Queue is empty Consumer is waiting , size: 0  Produced: 1  Produced: 2  Consumed: 1  Consumed: 2  Queue is empty Consumer is waiting , size: 0  Produced: 3  Produced: 4  Consumed: 3  Produced: 5  Consumed: 4  Produced: 6  Consumed: 5  Consumed: 6  Queue is empty Consumer is waiting , size: 0  Produced: 7  Consumed: 7  Queue is empty Consumer is waiting , size: 0 |

I will suggest you to change the time taken by producer and consumer threads to different times, and check the different outputs in different scenario.

## 3. Interview questions on wait(), notify() and notifyAll() methods

#### 3.1. What happens when notify() is called and no thread is waiting?

In general practice, this will not be the case in most scenarios if these methods are used correctly. Though if the notify() method is called when no other thread is waiting, notify() simply returns and the notification is lost.

Since the wait-and-notify mechanism does not know the condition about which it is sending notification, it assumes that a notification goes unheard if no thread is waiting. A thread that later executes the **wait()** method has to wait for another notification to occur.

#### 3.2. Can there be a race condition during the period that the wait() method releases OR reacquires the lock?

The wait() method is tightly integrated with the lock mechanism. The object lock is not actually freed until the waiting thread is already in a state in which it can receive notifications. It means only when thread state is changed such that it is able to receive notifications, lock is held. The system prevents any race conditions from occurring in this mechanism.

Similarly, system ensures that lock should be held by object completely before moving the thread out of waiting state.

#### 3.3. If a thread receives a notification, is it guaranteed that the condition is set correctly?

Simply, no. Prior to calling the wait() method, a thread should always test the condition while holding the synchronization lock. Upon returning from the wait() method, the thread should always retest the condition to determine if it should wait again. This is because another thread can also test the condition and determine that a wait is not necessary — processing the valid data that was set by the notification thread.

This is a common case when multiple threads are involved in the notifications. More particularly, the threads that are processing the data can be thought of as consumers; they consume the data produced by other threads. There is no guarantee that when a consumer receives a notification that it has not been processed by another consumer.

As such, when a consumer wakes up, it cannot assume that the state it was waiting for is still valid. It may have been valid in the past, but the state may have been changed after the notify() method was called and before the consumer thread woke up. Waiting threads must provide the option to check the state and to return back to a waiting state in case the notification has already been handled. This is why we always put calls to the wait() method in a loop.

#### 3.4. What happens when more than one thread is waiting for notification? Which threads actually get the notification when the notify() method is called?

It depends on many factors.Java specification doesn’t define which thread gets notified. In runtime, which thread actually receives the notification varies based on several factors, including the implementation of the Java virtual machine and scheduling and timing issues during the execution of the program.

There is no way to determine, even on a single processor platform, which of multiple threads receives the notification.

Just like the notify() method, the notifyAll() method does not allow us to decide which thread gets the notification: they all get notified. When all the threads receive the notification, it is possible to work out a mechanism for the threads to choose among themselves which thread should continue and which thread(s) should call the wait() method again.

#### 3.5. Does the notifyAll() method really wake up all the threads?

Yes and no. All of the waiting threads wake up, but they still have to reacquire the object lock. So the threads do not run in parallel: they must each wait for the object lock to be freed. Thus, only one thread can run at a time, and only after the thread that called the notifyAll() method releases its lock.

#### 3.6. Why would you want to wake up all of the threads if only one is going to execute at all?

There are a few reasons. For example, there might be more than one condition to wait for. Since we cannot control which thread gets the notification, it is entirely possible that a notification wakes up a thread that is waiting for an entirely different condition.

By waking up all the threads, we can design the program so that the threads decide among themselves which thread should execute next. Another option could be when producers generate data that can satisfy more than one consumer. Since it may be difficult to determine how many consumers can be satisfied with the notification, an option is to notify them all, allowing the consumers to sort it out among themselves.

# The Evolution of the Producer-Consumer Problem in Java

<https://dzone.com/articles/the-evolution-of-producer-consumer-problem-in-java>

## Using a Thread Pool

What else can we improve here? Let’s analyze what we did. We’ve instantiated two threads, one that puts some elements in the blocking queue, the producer, and another that retrieves elements from the queue, the consumer.

But, good software techniques suggest that creating and destroying threads manually is bad practice. Thread creation is an expensive task. Each thread creation implies the following steps:

* It allocates memory for a thread stack
* The OS creates a native thread corresponding to the Java thread
* Descriptors relating to the thread are added to the JVM internal data structures

Don’t get me wrong. There is nothing wrong with having more threads. That’s how parallelism works. The problem here is that we’ve created them “manually." That’s the bad practice. If we create them manually, besides the creation’s cost, another problem is that we don’t have control over how many of them are running at the same time. For example, if millions of requests are reaching a server app, and for each request, a new thread is created, then millions of threads will run in parallel and this could lead to a [thread starvation](https://en.wikipedia.org/wiki/Starvation_(computer_science)).

So, we need a way to strategically manage threads. And here comes the thread pools.

Thread pools handle the threads' life cycle based on a selected strategy. It holds a limited number of idle threads and reuses them when it needs to solve tasks. This way, we don’t have to create a new thread every time for a new request, and therefore, we can avoid reaching a thread starvation,

The Java thread pool implementation consists of:

* A task queue
* A collection of worker threads
* A thread factory
* Metadata for managing thread pool state.

For running some tasks concurrently, you have to put them in the task queue. Then, when a thread is available, it will receive a task and run it. The more available threads, the more tasks that are executed in parallel.

Beside the thread lifecycle management, another advantage when working with a thread pool is that when you plan on how to split the work to be executed concurrently, you can think in a more functional way. The unit of parallelism is not the thread anymore; it’s the task. You design some tasks that are executed concurrently, and not some threads that share a common memory and are running in parallel. Thinking in a functional way can help us avoid some common multithreading problems, like deadlocks or data races. Nothing can stop us from reaching into these problems again, but, because using the functional paradigm, we don’t imperatively synchronize the concurrent computation (with locks). This is far less happening than working directly with threads and shared memory. This is not the case in our example since the tasks share a blocking queue, but I just wanted to highlight this advantage.

[Here](https://allegro.tech/2015/05/thread-pools.html)and [here](https://dzone.com/articles/getting-the-most-out-of-the-java-thread-pool) you can find more details about thread pools.

With all of this being said, let’s see how our example looks using a thread pool.

package ProducerConsumer;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.LinkedBlockingDeque;

public class ProducerConsumerExecutorService {

public static void main(String[] args) {

BlockingQueue<Integer> blockingQueue = new LinkedBlockingDeque<>(2);

ExecutorService executor = Executors.newFixedThreadPool(2);

Runnable producerTask = () -> {

try {

int value = 0;

while (true) {

blockingQueue.put(value);

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

value++;

Thread.sleep(1000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

};

Runnable consumerTask = () -> {

try {

while (true) {

int value = blockingQueue.take();

System.out.println("Consume " + value);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

};

executor.execute(producerTask);

executor.execute(consumerTask);

executor.shutdown();

}

}

The difference here is that, instead of manually creating and running the consumer and producer threads, we build a thread pool, and it will receive two tasks, a producer and a consumer task. The producer and consumer tasks are actually the same runnables that were used in the previous example. Now, the executor (the thread pool implementation) receives the tasks, and its working threads will execute them.

In our simple case, everything will work the same as before. Just like in previous examples, we still have two threads, and they still produce and consume elements in the same way. So, we don’t have a performance improvement here, but the code looks cleaner. We no longer build the threads manually, but, instead, we just specify what we want. And, we want a way to execute some tasks in parallel.

So, when you use a thread pool, you don’t have to think to threads as the unit of parallelism, but instead, you think to some tasks that are executed concurrently. That’s what you need to know, and the executor will handle the rest. It will receive some tasks, and then, it will execute them using the available working threads.