<https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html>

# Guarded Blocks

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose, for example guardedJoy is a method that must not proceed until a shared variable joy has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

public void guardedJoy() {

// Simple loop guard. Wastes

// processor time. Don't do this!

while(!joy) {}

System.out.println("Joy has been achieved!");

}

A more efficient guard invokes [Object.wait](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--) to suspend the current thread. The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

public synchronized void guardedJoy() {

// This guard only loops once for each special event, which may not

// be the event we're waiting for.

while(!joy) {

try {

wait();

} catch (InterruptedException e) {}

}

System.out.println("Joy and efficiency have been achieved!");

}

**Note: Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.**

Like many methods that suspend execution, wait can throw InterruptedException. In this example, we can just ignore that exception — we only care about the value of joy.

Why is this version of guardedJoy synchronized? Suppose d is the object we're using to invoke wait. When a thread invokes d.wait, it must own the intrinsic lock for d — otherwise an error is thrown. Invoking wait inside a synchronized method is a simple way to acquire the intrinsic lock.

When wait is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke [Object.notifyAll](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notifyAll--), informing all threads waiting on that lock that something important has happened:

public synchronized notifyJoy() {

joy = true;

notifyAll();

}

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

**Note:** There is a second notification method, notify, which wakes up a single thread. Because notify doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.

Let's use guarded blocks to create a *Producer-Consumer* application. This kind of application shares data between two threads: the *producer*, that creates the data, and the *consumer*, that does something with it. The two threads communicate using a shared object. Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data.

In this example, the data is a series of text messages, which are shared through an object of type [Drop](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Drop.java):

public class Drop {

// Message sent from producer

// to consumer.

private String message;

// True if consumer should wait

// for producer to send message,

// false if producer should wait for

// consumer to retrieve message.

private boolean empty = true;

public synchronized String take() {

// Wait until message is

// available.

while (empty) {

try {

wait();

} catch (InterruptedException e) {}

}

// Toggle status.

empty = true;

// Notify producer that

// status has changed.

notifyAll();

return message;

}

public synchronized void put(String message) {

// Wait until message has

// been retrieved.

while (!empty) {

try {

wait();

} catch (InterruptedException e) {}

}

// Toggle status.

empty = false;

// Store message.

this.message = message;

// Notify consumer that status

// has changed.

notifyAll();

}

}

The producer thread, defined in [Producer](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Producer.java), sends a series of familiar messages. The string "DONE" indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

import java.util.Random;

public class Producer implements Runnable {

private Drop drop;

public Producer(Drop drop) {

this.drop = drop;

}

public void run() {

String importantInfo[] = {

"Mares eat oats",

"Does eat oats",

"Little lambs eat ivy",

"A kid will eat ivy too"

};

Random random = new Random();

for (int i = 0;

i < importantInfo.length;

i++) {

drop.put(importantInfo[i]);

try {

Thread.sleep(random.nextInt(5000));

} catch (InterruptedException e) {}

}

drop.put("DONE");

}

}

The consumer thread, defined in [Consumer](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/Consumer.java), simply retrieves the messages and prints them out, until it retrieves the "DONE" string. This thread also pauses for random intervals.

import java.util.Random;

public class Consumer implements Runnable {

private Drop drop;

public Consumer(Drop drop) {

this.drop = drop;

}

public void run() {

Random random = new Random();

for (String message = drop.take();

! message.equals("DONE");

message = drop.take()) {

System.out.format("MESSAGE RECEIVED: %s%n", message);

try {

Thread.sleep(random.nextInt(5000));

} catch (InterruptedException e) {}

}

}

}

Finally, here is the main thread, defined in [ProducerConsumerExample](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/ProducerConsumerExample.java), that launches the producer and consumer threads.

public class ProducerConsumerExample {

public static void main(String[] args) {

Drop drop = new Drop();

(new Thread(new Producer(drop))).start();

(new Thread(new Consumer(drop))).start();

}

}

**Note:** The Drop class was written in order to demonstrate guarded blocks. To avoid re-inventing the wheel, examine the existing data structures in the [Java Collections Framework](https://docs.oracle.com/javase/tutorial/collections/index.html) before trying to code your own data-sharing objects. For more information, refer to the [Questions and Exercises](https://docs.oracle.com/javase/tutorial/essential/concurrency/QandE/questions.html) section.

# 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**.

Read more : [Difference between wait() and sleep() in Java](https://howtodoinjava.com/java/multi-threading/sleep-vs-wait/)

## 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().

Let’s write a small program to understand **how wait(), notify(), notifyall() methods should be used** to get desired results.

## 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.