How To Do In Java

How to Work With wait(), notify() and notifyAll() in Java?

Multithreading in java 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** and **Executors** which take away some of the complexity by providing easy to use APIs. Programmers using these classes will feel a lot more confident than programmers directly handling synchronization stuff using **wait() and notify()** 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 some **concepts around methods wait(), notify() and notifyAll()**.

What are wait(), notify() and notifyAll() methods?

Before moving into concepts, lets note down few very basic definitions involved for these methods.

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

1. **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:

```
synchronized( lockObject )
{
    while( ! condition )
    {
        lockObject.wait();
    }

    //take the action here;
}
```

2. **notify()** : It wakes up the first 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:

```
synchronized(lockObject)
{
    //establish_the_condition;

    lockObject.notify();

    //any additional code if needed
```

```
}
```

3. **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:

```
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().

So far, we learned few basic things which you probably already knew. Let's write a small program to understand how these methods should be used to get desired results.

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 :

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

Designing Producer Thread

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

```
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.wait();
            }

            Thread.sleep(1000);
            taskQueue.add(i);
            System.out.println("Produced: " + i);
            taskQueue.notifyAll();
        }
    }
}
```

1) Here "produce(counter++)" code has been written inside infinite loop so that producer keeps producing elements at regular interval.
2) We have written the produce() method code following the general guideline to write wait() method as mentioned in first section.
3) 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.
4) Consumer thread after getting notification, if ready to consume the element as per written logic.
5) Note that both threads use sleep() methods as well for simulating time delays in creating and consuming elements.

Designing Consumer Thread

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

```
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 , siz
                taskQueue.wait();
            }
            Thread.sleep(1000);
            int i = (Integer) taskQueue.remove(0);
            System.out.println("Consumed: " + i);
            taskQueue.notifyAll();
        }
    }
}
```

1) Here "consume()" code has been written inside infinite loop so that consumer keeps consuming elements whenever it finds something in taskQueue..
2) 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.
3) Producer thread after getting notification, if ready to produce the element as per written logic.

Test the Application

Now lets test producer and consumer threads.

```
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();
   }
}

Output:

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.

Interview Questions on wait(), notify() and notifyAll() Methods

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.

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.

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.

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.

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.

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.

Happy Learning !!

Related Posts:

1. **Difference between sleep() and wait()?**
2. **Difference between yield() and join() in threads in java?**
3. **How to Use Locks in Java | java.util.concurrent.locks.Lock Tutorial and Example**
4. **Thread synchronization, object level locking and class level locking**
5. **How to use BlockingQueue and ThreadPoolExecutor in java**
6. **When to use CountDownLatch : Java concurrency example tutorial**
7. **Core java interview questions series : Part 3**
8. **Real java interview questions asked for Oracle Enterprise Manager Project**