ReentrantLock class provides implementation of Lock’s newCondition() method - description and solving producer consumer program using this method in java

You are here : Home / Core Java Tutorials / Thread Concurrency Tutorial in java

In previous thread concurrency tutorial we learned what is Locks and ReEntrantLocks in java and in this thread concurrency tutorial we will learn
*about **Condition** interface, Condition interface **important methods**, Program to demonstrate usage of **newCondition()** method for solving Producer consumer problem,* difference between ***traditional synchronization*** *and* ***Condition*** *interface.*

Contents of page :

*1) About **Condition** interface in thread concurrency in java.*

Condition interface is found in `java.util.concurrent.locks` package.
Condition instance are similar to using Wait(), notify() and notifyAll() methods on object.

In case of **traditional synchronization**, there is **only one object monitor** so we can have only **single wait-set per object**. But,
Condition instance are used with Lock instance, **Condition factors out the Object monitor methods** (wait, notify and notifyAll) into distinct objects to give the **multiple wait-sets per**

Lock replaces the use of synchronized methods and blocks, & a **Condition** replaces the use of the Object monitor methods.

*1.1) Condition interface **important methods** in java >*

- *void await()*

method is **similar to** wait() method of object class.

method causes the current thread to wait until one of the following thing happen >

- signal()/signalAll() method is called, or
- current thread is interrupted.

- *boolean await(long time, TimeUnit unit)*

method is **similar to** **wait(long timeout)** method of object class.

method causes the current thread to wait until one of the following thing happen >

- signal()/signalAll() method is called, or
- current thread is interrupted, or
- specified *time* elapses.

- *void signal()*

method is **similar to notify()** method of object class.

Wakes up one waiting thread. Thread waits by calling await() method.

- *void signalAll()*

method is **similar to** **notifyAll()** method of object class.

Wakes up all waiting thread. Thread waits by calling await() method.

*2) **ReentrantLock** is a class which implements **Lock** interface. So, ReentrantL class provides implementation of Lock’s **newCondition()** method in java >*

- *Condition newCondition()*

  Method returns a Condition instance to be used with this Lock instance.

  Condition instance are similar to using Wait(), notify() and notifyAll() methods.

  - IllegalMonitorStateException is thrown **if this lock is not held when** any of the **Condition waiting** or **signalling methods** are called.
  - **Lock is released** when the **condition waiting methods are called** and before they return, the lock is reacquired and the **lock hold count** restored to what it wa the method was called.
  - If a **thread is interrupted while waiting** then InterruptedException will be thrown and following things will happen -
    - the **wait will be over**, and
    - **thread's interrupted status will be cleared**.
  - Waiting threads are signalled in FIFO (first in first out order) order.
  - When lock is ***fair***, first lock is obtained by longest-waiting thread.

    If lock is not ***fair***, any waiting thread could get lock, at discretion of implementation.

*3) Program/ Example to demonstrate usage of **newCondition()** method - solving Producer consumer problem in java >*

```java
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class ReentrantLockConditionTest {

    public static void main(String[] args) {

           List<Integer> sharedQueue = new LinkedList<Integer>(); //Creating shared object

     Lock lock = new ReentrantLock();
      //producerCondition
      Condition producerCondition = lock.newCondition();
      //consumerCondition
      Condition consumerCondition = lock.newCondition();

       Producer producer=new Producer(sharedQueue,lock,producerCondition,consumerCondition);
       Consumer consumer=new Consumer(sharedQueue,lock,producerCondition,consumerCondition);

        Thread producerThread = new Thread(producer, "ProducerThread");
        Thread consumerThread = new Thread(consumer, "ConsumerThread");
        producerThread.start();
        consumerThread.start();

    }


}


/**
 * Producer Class.
 */
class Producer implements Runnable {

    private List<Integer> sharedQueue;
    private int maxSize=2; //maximum number of products which sharedQueue can hold at a time.

    Lock lock;
     Condition producerCondition;
     Condition consumerCondition;


    public Producer(List<Integer> sharedQueue, Lock lock,
             Condition producerCondition, Condition consumerCondition) {
        this.sharedQueue = sharedQueue;
        this.lock=lock;
        this.producerCondition=producerCondition;
        this.consumerCondition=consumerCondition;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {  //produce 10 products.
         try {
           produce(i);
         } catch (InterruptedException e) {  e.printStackTrace();   }
        }
    }

    public void produce(int i) throws InterruptedException {
           lock.lock();

           // if sharedQuey is full producer await until consumer consumes.
           if (sharedQueue.size() == maxSize) {
                  producerCondition.await();
           }

           System.out.println("Produced : " + i);
           // as soon as producer produces (by adding in sharedQueue) it signals consumer.
           sharedQueue.add(i);
           consumerCondition.signal();

           lock.unlock();

    }

}

/**
 * Consumer Class.
 */
class Consumer implements Runnable {
    private List<Integer> sharedQueue;
    Lock lock;
     Condition producerCondition;
     Condition consumerCondition;

    public Consumer(List<Integer> sharedQueue, Lock lock,
             Condition producerCondition, Condition consumerCondition) {
        this.sharedQueue = sharedQueue;
        this.lock=lock;
        this.producerCondition=producerCondition;
        this.consumerCondition=consumerCondition;
    }

    @Override
    public void run() {
           for (int i = 1; i <= 10; i++) {   //produce 10 products.
         try {
           consume();
         } catch (InterruptedException e) {  e.printStackTrace();   }
        }

    }


    public void consume() throws InterruptedException {
           lock.lock();

           // if sharedQuey is empty consumer await until producer produces.
           if (sharedQueue.size() == 0) {
                  consumerCondition.await();
           }


        /*If sharedQueue not empty consumer will consume
      * (by removing from sharedQueue) and signal the producer.
      */
           System.out.println("CONSUMED: " + sharedQueue.remove(0));
           producerCondition.signal();

             lock.unlock();

    }


}

/*OUTPUT

Produced : 1
Produced : 2
CONSUMED: 1
CONSUMED: 2
Produced : 3
Produced : 4
CONSUMED: 3
CONSUMED: 4
Produced : 5
Produced : 6
CONSUMED: 5
CONSUMED: 6
```

```
Produced : 7
Produced : 8
CONSUMED: 7
CONSUMED: 8
Produced : 9
Produced : 10
CONSUMED: 9
CONSUMED: 10

*/
```

3.1) Program and output analyzation >

We created following and shared them with both producer and consumer

- `sharedQueue`
- `Lock lock = new ReentrantLock();`
  `//producerCondition`
- `Condition producerCondition = lock.newCondition();`
  `//consumerCondition`
- `Condition consumerCondition = lock.newCondition();`

Below operations are performed by acquiring a lock [by calling `lock.lock()`].

`if sharedQuey is full producer await` [by calling `producerCondition.await()`] `until consumer consumes.`

`As soon as producer produces` [by calling `sharedQueue.add(i)`] , `it signals consumer`[by calling `consumerCondition.signal()`]. Once signalling has been release lock [b `lock.unLock()`]

Below operations are performed by acquiring a lock [by calling `lock.lock()`].

`if sharedQuey is empty consumer await` [by calling `consumerCondition.await()`] `until producer produces.`

`If sharedQueue not empty consumer will consume` [by calling `sharedQueue.remove(0)`] `and signal the producer` [by calling `producerCondition.signal()`]. Once signa been release lock [by calling `lock.unLock()`]

*4) Difference between **traditional synchronization** and **Condition** interface in thre concurrency in java.*

- In case of **traditional synchronization**, there is **only one object monitor** so we can have only **single wait-set per object**. But,
  Condition instance are used with Lock instance, **Condition factors out the Object monitor methods** (wait, notify and notifyAll) into distinct objects to give the **multiple wait-s object**.

- **Example >** In the above program we we created **`producerCondition`** and **`consumerCondition`** from Lock**.** And thread rather than acquiring and releasing lock on produc consumer object as done in previous tutorials, acquired and released lock on **`producerCondition`** and **`consumerCondition`.** Hence, enabling **multiple wait-sets per ob**

Summary >

In previous thread concurrency tutorial we learned what is **Locks and ReEntrantLocks in java** and in this thread concurrency tutorial we learne *about **Condition** interface, Condition interface **important methods**, Program to demonstrate usage of **newCondition()** method for solving Pro consumer problem,* difference between ***traditional synchronization** and **Condition** interface.*

Having any doubt? or you you liked the tutorial! Please comment in below section.

*Please express your love by liking **JavaMadeSoEasy.com** (**JMSE**) on **facebook**, following on **google+** or Twitter.*