Producer Consumer pattern using Custom implementation of BlockingQueue interface in java

Contents of page :

Hi ! In this tutorial we will learn how to solve Producer consumer problem using custom LinkedBlockingQueue class which implements BlockingQueue interface. Earlier, we solved Consumer Producer problem by using BlockingQueue provided in Java API.

1) Key Features of custom/own ***BlockingQueue*** *in java >*

- This BlockingQueue implementation follows FIFO (first-in-first-out).
- New elements are inserted at the tail of the queue and,
- Removal elements is done at the head of the queue.
- Blocking queue internally uses Linked List for implementing Queue in java.

2) Methods used in custom BlockingQueue in java >

`void put(E item) throws InterruptedException ;`	>Inserts the specified element into this queue only if space is available else waits for space to become available. `>used by producer to put/produce in sharedQueue.`
`E take() throws`	>Retrieves and removes the head of this queue Retrieves

`InterruptedException;`	and removes the head of this queue waits for element to become available. **>**`used by consumer to take/consume from sharedQueue.`

3) Example/ Program to solve Consumer Producer problem using custom implementation of >

- **BlockingQueue** interface in java and,
- **LinkedBlockingQueue** class which implements BlockingQueue interface in java.

```
import java.util.LinkedList;
import java.util.List;




/**
 * Implementing custom BlockingQueue interface .
 * This BlockingQueue implementation follows FIFO (first-in-first-out).
 * New elements are inserted at the tail of the queue,
 * and removal elements is done at the head of the queue.
 *
 * @author AnkitMittal
 * Copyright (c), AnkitMittal .
 * All Contents are copyrighted and must not be reproduced in any form.
 */
interface BlockingQueueCustom<E> {

      /**
       * Inserts the specified element into this queue
       * only if space is available else
       * waits for space to become available.
       */
      void put(E item)  throws InterruptedException ;


      /**
       * Retrieves and removes the head of this queue
       * only if elements are available else
       * waits for element to become available.
       */
      E take()  throws InterruptedException;
}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
/**
 * Implementing custom LinkedBlockingQueue class.
 * This BlockingQueue implementation follows FIFO (first-in-first-out).
 * New elements are inserted at the tail of the queue,
 * and removal elements is done at the head of the queue.
 *
 * @author AnkitMittal
 * Copyright (c), AnkitMittal .
```

```
 * All Contents are copyrighted and must not be reproduced in any form.
 */
class LinkedBlockingQueueCustom<E> implements BlockingQueueCustom<E>{

      private List<E> queue;
      private int  maxSize ; //maximum number of elements queue can hold at a time.

      public LinkedBlockingQueueCustom(int maxSize){
     this.maxSize = maxSize;
     queue = new LinkedList<E>();
      }


      /**
       * Inserts the specified element into this queue
       * only if space is available else
       * waits for space to become available.
       */
      public synchronized void put(E item)  throws InterruptedException  {

               //check space is available or not.
               if (queue.size() == maxSize) {
             this.wait();
               }

               //space is available, insert.
          queue.add(item);
          this.notify();
      }


      /**
       * Retrieves and removes the head of this queue
       * only if elements are available else
       * waits for element to become available.
       */
      public synchronized E take()  throws InterruptedException{

            //waits element is available or not.
        if (queue.size() == 0) {
            this.wait();
        }

        //element is available, remove.
        this.notify();
         return queue.remove(0);
      }
}


/**
 * Producer Class in java
 */
class Producer implements Runnable {

    private final BlockingQueueCustom<Integer> sharedQueue;

    public Producer(BlockingQueueCustom<Integer> sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
```

```
    @Override
    public void run() {
        for(int i=1; i<=10; i++){
         try {
             System.out.println("Produced : " + i);
             //put/produce into sharedQueue.
             sharedQueue.put(i);
         } catch (InterruptedException ex) {

         }
        }
    }

}

/**
 * Consumer Class in Java
 */
class Consumer implements Runnable{

    private BlockingQueueCustom<Integer> sharedQueue;

    public Consumer (BlockingQueueCustom<Integer> sharedQueue) {
        this.sharedQueue = sharedQueue;
    }

    @Override
    public void run() {
        while(true){
         try {
           //take/consume from sharedQueue.
             System.out.println("CONSUMED : "+ sharedQueue.take());
         } catch (InterruptedException ex) {

         }
        }
    }


}


/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
/**
 * Main class in java
 */
public class ProducerConsumerBlockingQueueCustom {

    public static void main(String args[]){

     BlockingQueueCustom<Integer> sharedQueue = new LinkedBlockingQueueCustom<Integer>(10);
//Creating shared object

     Producer producer=new Producer(sharedQueue);
     Consumer consumer=new Consumer(sharedQueue);

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

```
    }

}

/*

Produced : 1
Produced : 2
Produced : 3
CONSUMED : 1
Produced : 4
CONSUMED : 2
Produced : 5
CONSUMED : 3
Produced : 6
CONSUMED : 4
Produced : 7
CONSUMED : 5
Produced : 8
CONSUMED : 6
Produced : 9
CONSUMED : 7
Produced : 10
CONSUMED : 8
CONSUMED : 9
CONSUMED : 10

*/
```