

# 13. Notes - BlockingQueue and revised producer and consumer problem

## java.util.concurrent package

Has several utilities which help developing multithreaded application.

- ExecutorService
- Callable interface
- Future object
- Locks
- BlockingQueue implementations
- Few other utilities.

## BlockingQueue

BlockingQueue is an interface that extends Queue interface. And the implementations include

- ArrayBlockingQueue
  - It is bounded i.e. you need to specify the size.
  - operate on FIFO logic i.e. first in first out, which means that the first inserted element will be the first to be removed.
- LinkedBlockingQueue
  - Optionally Bounded, based on linked nodes i.e. nothing but the linked list.
  - It too operates on FIFO logic.
- PriorityBlockingQueue
  - Unbounded

- Objects should be Comparable or you should provide a Comparator.
- And there are few other implementations as well.

## BlockingQueue operations

Operations that throw Exception if the operation fails.

- add(o)
  - It tries to add an element and if there is no sufficient capacity available this method will throw an exception.
- remove(o)
  - Removes the element that matches with the given object it compares the elements using equals method.
- element()
  - Returns the head element with out removing it. But element() method throws an exception if queue is empty

Operations that return a boolean value with out exception

- offer(o)
  - Returns true if the element is added otherwise false.
- poll()
  - Removes the head element of the queue and returns it, if queue is empty it returns null.
- peek()
  - Returns the head element with out removing it, it returns null if queue is empty.

Operations that block.

- put(o)
  - It will add the element to the queue, but if the queue is full, then it will block the thread till the space is available.
- take()

- Returns the head element of the queue, if queue is empty this method will block the thread till an element is available.

And the methods with timeout

- offer(o, timeout, timeunit)
- poll(timeout, timeunit)

## Revised Producer and Consumer Example -

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

// Changed from MessageQueue to BlockingQueue.

class ProducerThread extends Thread {
    BlockingQueue<String> queue;

    public ProducerThread(BlockingQueue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for(int i=1; i <= 10; i++) {
            String msg = "Hello-" + i;

            // Blocks the thread until the space is available.
            try {
                queue.put(msg);
                System.out.println("Produced - " + msg);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class ConsumerThread extends Thread {
    BlockingQueue<String> queue;

    public ConsumerThread(BlockingQueue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for(int i=1; i<=10; i++) {
```

```

        String message = null;

        // Blocks the thread until the element is available.
        try {
            message = queue.take();
            System.out.println("Consumed - " + message);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1);
        new ProducerThread(queue).start();
        new ConsumerThread(queue).start();
    }
}

```