12. Notes - Producer and Consumer Problem

Thread signalling using wait() and notify() -

Threads can signal each other using the wait and notify methods. wait(), notify() and notifyAll() are the methods of the class Object and hence they are part of all the objects. Just think about a scenario where one thread waits for a signal from some other thread in order to proceed with the execution.

wait() method Releases the lock over the object and takes the thread to WAITING state. And the thread remains in that state until some other thread calls the notify() method over the same object. Once notify() is invoked it ends the wait for one single thread and takes the thread to BLOCKED state where the thread remains in that state till the lock is obtained. wait() only returns after obtaining the lock.

wait(long millis) slightly differs, as it takes thread to TIMED_WAITING and waits only for the specified duration.

notify() - notifies one single thread where as **notifyAll()** notifies all the threads waiting for the signal.

Note - In order to call the wait and notify methods the corresponding thread should hold the lock on the object using synchronized method or block.

Producer and Consumer Problem -

Here the problem that we are dealing with is the slow consumer, problem occurs when the producer produces the messages at a faster rate than the consumer can consume.

How to solve this problem?

There are different ways of solving it, one approach is to ask the producer to wait till the message is consumed.

Example -

```
// Producer and Consumer problem
//    producer --> messageQueue --> consumer.

import java.util.ArrayList;
import java.util.List;

class MessageQueue {

   List<String> messages;
   int limit;

   public MessageQueue(int limit) {
       messages = new ArrayList<String>();
       this.limit = limit;
   }

   public boolean isFull() {
       return messages.size() == limit;
   }

   public boolean isEmpty() {
       return messages.size() == 0;
   }

   public synchronized void enqueue(String msg) {

       while (isFull()) {
           try {
              this.wait();
           } catch (InterruptedException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
           }
       }

       messages.add(msg);
       this.notify();
   }

   public synchronized String dequeue() {

       while (isEmpty()) {
           try {
              this.wait();
           } catch (InterruptedException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
           }
```

```
        }

        String message = messages.remove(0);
        this.notify();
        return message;
    }

}

class ProducerThread extends Thread {
    MessageQueue queue;

    public ProducerThread(MessageQueue queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for(int i=1; i <= 10; i++) {
            String msg = "Hello-" + i;
            queue.enqueue(msg);
            System.out.println("Produced - " + msg);
        }
    }
}

class ConsumerThread extends Thread {
    MessageQueue queue;

    public ConsumerThread(MessageQueue queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        for(int i=1; i<=10; i++) {
            String message = queue.dequeue();
            System.out.println("Consumed - " + message);
        }
    }
}

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

Explanation -

Here we have a class MessageQueue which acts as a buffer between producer thread and the consumer thread. And the buffer is bounded, so we can set the limit during object creation.

isFull() returns true if messages size reaches the limit.

isEmpty() return true if messages size is equal to zero.

enqueue() - Here it is invoked only by ProducerThread; it adds (at the end) the message to the messages array if it is not full, if full it calls the wait() over the queue object (this) till the consumer consumes the message and notifies it. Also once it adds the message to the queue, it calls the notify() in order to end the consumer's wait() if any.

dequeue() - Here it is invoked only by ConsumerThread; it removes the first element and returns it. If the queue is empty it calls the wait() over the queue object till the producer produces the message. Once it consumes the message it calls the notify() to end the producer's wait() if any.

main() initiates these operations, by creating the MessageQueue object and passing it to both the producer and consumer threads.