Producer consumer problem

Aim

Write a program to implement producer consuming

Program description

Multi threading replaces event loop programming by dividing your tasks into discrete and logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called 0only from within a synchronized context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- notify( ) wakes up the first thread that called wait( ) on the same object.
- notifyAll( ) wakes up all the threads that called wait( ) on the same object.

The highest priority thread will run first.

These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

Additional forms of wait( ) exist that allow you to specify a period of time to wait.

The following sample program incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

```
// An incorrect implementation of a producer and consumer.
class Q {
        int n;
        synchronized int get() {
                System.out.println("Got: " + n);
                return n;
        }
        synchronized void put(int n) {
                this.n = n;
                System.out.println("Put: " + n);
                }
}
class Producer implements Runnable {
        Q q;
```

```
        Producer(Q q) {
                this.q = q;
                new Thread(this, "Producer").start();
                }
        public void run() {
                int i = 0;
                while(true) {q.put(i++);}
                }
}
class Consumer implements Runnable {
        Q q;
        Consumer(Q q) {
                this.q = q;
                new Thread(this, "Consumer").start();
                }
        public void run() {
                while(true) {q.get();}
                }
}
class PC {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
                System.out.println("Press Control-C to stop.");
        }
}
```

Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
        int n;
        boolean valueSet = false;
        synchronized int get() {
                if(!valueSet)
                try {wait();
                } catch(InterruptedException e) {
                        System.out.println("InterruptedException caught");
                }
                System.out.println("Got: " + n);
                valueSet = false;
                notify();
                return n;
        }
        synchronized void put(int n) {
                if(valueSet)
                try {
                        wait();
                } catch(InterruptedException e) {
                        System.out.println("InterruptedException caught");
                }
                this.n = n;
                valueSet = true;
                System.out.println("Put: " + n);
                notify();
        }
}
class Producer implements Runnable {
        Q q;
        Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
        }
        public void run() {
                int i = 0;
                while(true) {
                        q.put(i++);
                        }
        }
}
class Consumer implements Runnable {
        Q q;
        Consumer(Q q) {
                this.q = q;
                new Thread(this, "Consumer").start();
        }
        public void run() {
```

```
                while(true) {q.get();}
        }
}
class PCFixed {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
                System.out.println("Press Control-C to stop.");
        }
}
```

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells the **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```