Computer Programming 2 with JAVA

Practical 10 : Threads - Producers and Consumers

Written and composed by Robin Knox-Grant : "Beyond the Buttons"

In Object Oriented software design, objects communicate with each other by exchanging "messages". If object X needs some data from object Y, X will send a message to Y asking for what it needs. Y will then respond by supplying the requested data – if it can.

Y may not have the data X needs when it is asked for, in which case X will have to keep asking Y at regular intervals until Y is able to deliver. Implementing this technique in code would represent a technique called *"polling"*. It would be incorporated in a loop in which Y's status is checked with each iteration of the loop. This would continue until Y is able to deliver, but in the process, valuable clock cycles are wasted.

If we applied the polling technique to this Producer/Consumer scenario, the Producer would run a loop which would check to see if the Consumer was ready to receive new data. This would be necessary because the new data would be lost if the Consumer was not ready to use it. But it is equally important that the Consumer knows that the Producer has produced a new number. If the Producer had not produced a new number when it is needed, the Consumer would be forced to use an old number – an equally unsatisfactory situation. So using polling, both the Producer and the Consumer threads spend most of their time checking on each other in an attempt to avoid losing valuable data.

The problem is not that the Consumer sometimes has to wait for the Producer (and vice versa). That's what we want. The problem is that the CPU is kept very busy with a totally unproductive activity (all this mutual checking). The question to be answered when using this polling technique is this. How often should the Consumer check the Producer and how often should the Producer check the Consumer? If the polling loops run too quickly, there will be unnecessary checking (unnecessary CPU activity) and if they run too slowly then time will be wasted as the one thread waits unnecessarily for the other. Given the fact that both the Producer and the Consumer take varying periods of time to process their data, there is no ideal frequency for either of the polling loops. Although the technique will work, it really is a no win situation with regard to efficiency.

The ideal would be to somehow enable the Producer and the Consumer to communicate with each other so that the Producer could tell the Consumer when it was ready to deliver, and the Consumer could tell the Producer when it was ready to receive. In that way the CPU would not have to waste valuable clock cycles (it could be doing other things) checking on the two threads to make sure that they were both ready to exchange data.

This Consumer/Producer relationship could be quite a legitimate fact of life; i.e. there could be many situations in which a method (the Consumer) needs to be supplied regularly with new data from some other part of the program (the Producer).

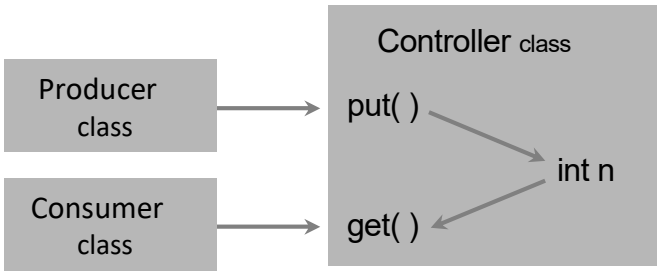
If the Consumer tries to access the Producer for code before it has been produced, there could be consequences. Likewise the Producer, as the supplier of the data, would want to be sure that the Consumer is going to be ready to use all the data that it produces. If it produces data that the Consumer is not ready to use, the data may be lost - again with possible consequences. Here's the first of two applications that will demonstrate what is being talked about.

FIT2034 : Threads – Producers and Consumers

First, this application sets up a scenario in which a Producer thread churns out a series of numbers which are to be used by a Consumer thread.

1. There is a *Producer thread* that produces a series of numbers from 0 to 19. It simply churns them out in a loop. With each iteration of the loop, it calls the method put(), and passes each new number to this method. All put() does is to print the numbers to the console output screen and then initialises a class variable '**n**' with the value passed to **put()**. The printing to the screen of the number by **put()** shows that the number has been produced.

2. Then there is a *Consumer thread*. This thread also runs a loop which calls the method **get().** All **get()** does is to access the class variable '**n**' which holds the latest number delivered by the producer, and prints it to the screen. The printing to the screen of the number by **get()** shows that number has been used.



Producer calls put() and passes new number. put() updates `n` and prints number to screen.
Consumer calls get(). get() reads `n` and prints number to screen.

The Producer's call to **put()** (each time the loop goes round) is delayed by a call to method randomDelay(). What this method does is to put the Producer thread to sleep for a period of time that *will be different each time it is called*. The sleep period is a randomly generated value. The range of the sleep periods will be from 30 to 2000 milliseconds.

The same thing happens in the Consumer thread. It calls get() with every iteration of its while() loop, but it's calls to get() are delayed by calls to its randomDelay() method (refer to code below). So now we've got a situation where the Producer thread is churning out a series of numbers and passing them to its **put()** method with every iteration of its loop. But the calls to **put()** are delayed by between 30 and 2000 milliseconds.

After number 19 has been produced, the Producer thread terminates the program.

The Consumer thread also runs a loop in which it calls its **get()** method with each loop iteration. It's calls to get() are also delayed by between 30 and 2000 milliseconds. The Controller class defines the get() and put() methods. The variable `n` is a class variable so it is accessible to both the put() and the get() methods. The `put()` method will *initialise* it and print its latest value to the screen and the get() method will *read* it and print its value to the screen. Still in the Controller class, the put() and get() methods are **synchronized** which of course means that the Producer and Consumer threads will have to get the Controller's lock before they will be allowed to run their respective methods.

If the producer thread is running its **`put()`** method, the consumer thread will not be allowed to run its get() method at the same time.

In this first program there is no attempt to get the two threads to talk to each other. The idea is to compare the performance of this first example with that of the next example in which we set up a communication channel between the two threads. Here's the first example.

The Producer thread runs in a loop that produces a sequence of numbers from 0 to 19. When the loop counter becomes bigger than 19, the program terminates. The important thing that Producer's run() does is to call method **`put()`** in the Controller class and pass n (number) to it. Method put() will print value **n** to the screen. It will be called nineteen times during the life of the program. There is an imposed time delay before each call to put(). A call to randomDelay() will make the thread sleep for a period of time between 30 and 2000 milliseconds. Each time put() is called, a new number is passed to it.

The Consumer thread also runs in a loop but doesn't produce numbers. All it does is to call get() in the Controller class. But it is also subject to a random time delay before it calls get(). There is no synchronized code in either of the thread classes, so the thread scheduler is free to run and interrupt the two threads whenever it likes - *except* when they are sleeping and when they are executing the synchronized methods put() and get() in the Controller class.

The Controller class defines the methods get() and put(). These are the methods that are called by the threads. Thread Producer calls put() and thread Consumer calls get(). All these methods do is to print the latest value of n to the screen when called by their respective threads.

Here's an example of the output I got on my machine. You will almost certainly get a different result on your computer. But whatever your output, there will be some numbers that are "put" by the Producer that are not "GOT" by the Consumer. There will also be some numbers that the Consumer is forced to use more than once because a new number has not been delivered by the Producer.

//----------The output I got on my machine ----------

```
        GOT:  0
 put:  0
 put:  1
        GOT:  1
 put:  2
        GOT:  2
 put:  3
 put:  4
        GOT:  4
 put:  5
        GOT:  5
        GOT:  5
 put:  6
        GOT:  6
 put:  7
        GOT:  7
 put:  8
```

```
        GOT:  8
put:  9
        GOT:  9
put: 10
        GOT: 10
        GOT: 10
        GOT: 10
put: 11
put: 12
put: 13
        GOT: 13
put: 14
        GOT: 14
put: 15
        GOT: 15
put: 16
        GOT: 16
put: 17
put: 18
        GOT: 18
put: 19
        GOT: 19
```

//---------- The JAVA source code which produced this output ----------

```
public class TestDriver
{
        public static void main(String args[ ])
        {
            Controller outputObj = new Controller();
            new Producer(outputObj);
            new Consumer(outputObj);
        }
}


public class Controller
{
        int n;                                          // n must be available to get()
        synchronized void get()
        {
            if(n < 10)   System.out.println("           GOT:  " + n);
            else         System.out.println("           GOT: " + n);
        }

        synchronized void put( int n )
        {
            this.n = n;
            if(n < 10)   System.out.println("  put:  " + n);
            else         System.out.println("  put: " + n);
        }
    }
```

```
public class Producer implements Runnable
{
        Controller  outputObj;
        Producer(Controller outputObj)              // Constructor method
        {
           this.outputObj = outputObj;              // Initialise the class variable outputObj
           new Thread(this, "Producer").start();    // "this" is datahandle of Producer class
        }
        public void run()                           // Producer's run() method defines
        {                                           // the Producer thread
           int i = 0;
           while(true)
           {
              randomDelay();                        // Thread will sleep for time
              outputObj.put( i++ );                 // between 30 and 1970 milliSecs
              if(i > 19)
                    System.exit(0);
           }
        }
        void randomDelay()
        {
           long time = (int) ( ( Math.random() * 1970 )  + 30 );
           try
           {
              Thread.sleep( time );
           }
           catch(InterruptedException e)
           {            System.out.println("Producer Interrupted");     }
        }
     }

public class Consumer implements Runnable
{
     Controller outputObj;
     Consumer(Controller outputObj)                 // Constructor method
     {
        this.outputObj = outputObj;
        new Thread(this, "Consumer").start();       // "this" is datahandle of Consumer class
     }
     public void run()                              // Consumer's run() method defines
     {                                              // the Consumer thread
        while(true)
        {
           randomDelay();                           // Thread will sleep for time between
                                                    // between 30 and 1970 milliSecs
           outputObj.get();
        }
     }
     void randomDelay()
     {
           int time = (int) ( ( Math.random() * 1970 ) + 30 );
           try
              {
                  Thread.sleep(time);
              }
           catch(InterruptedException e)
                { System.out.println("Consumer Interrupted"); }
           }
}
```

With reference to the output shown above, the situation is that the Producer is not waiting until the Consumer has used the new number, and the Consumer is not waiting until the Producer has produced one. The two threads are doing their own thing independently of each other, and in the process there's a lot of inefficiency. The Producer is producing some numbers that are not used by the Consumer, and are therefore lost to the Consumer, and the Consumer sometimes uses an old previously used number because the Producer hasn't delivered a new number. The basic problem is one that is not confined to computers. The parties concerned are not talking to each other. Sounds familiar?

So the question is how do we get the Producer to wait until the Consumer is ready to use a new number, and how do get the Consumer to wait until the Producer has produced a new number? The answer is the method **wait()**.

But the use of **wait()** does not solve all our problems. It's all very well telling the Producer to wait until the Consumer is ready to use a new number, but it will wait forever unless, sooner or later, the Consumer **notifies** the Producer that it (the Producer) can go ahead and produce a new number.

Likewise, it's all very well telling the Consumer to wait until the Producer has produced a new number, but the Consumer will wait indefinitely unless the Producer **notifies** the Consumer that it has a new number ready to be delivered. That's what we want and that's what the methods **wait()**, **notify()** and **notifyAll()** can do.

Methods wait() and notify()

Refer to the code below and the insertion of the methods wait() and notify(). According to the output (shown below), the code shows an improvement of the previous application with methods wait() and notify() included. Notice the following:

1. The get() and put() methods are synchronized which of course means that only one or the other can be accessed by a thread at any one time. While one of the synchronized methods is running, it cannot be interrupted by the thread scheduler.
2. wait() and notify() must be called from inside a try block because they can throw checked exceptions.
3. The boolean variable "flag" is a class variable which of course means it can be accessed from anywhere in the class for the purposes of either setting it or reading it.
4. Variable "flag" is set to false when it is declared. So therefore flag is false when either put() or get() is called for the first time. This has a bearing on the if() conditions of **lines 16** and **31** in the code that follows. Towards the end of the put() method, flag is set to true and towards the end of the get() method it is set to false.
5. Both threads are running in loops. The Producer thread calls put() from inside its loop and also increments **n**. It will terminate the program when n > 19. The Producer has to run method **put()** for n to be incremented. The Consumer thread calls get() from inside an infinite loop.

When a thread calls wait() it must own the lock of the object on which wait() is called. If it does not own the lock, an IllegalMonitorStateException will be thrown.

```
public class TestDriver
{
        public static void main(String args[ ])
        {
           Controller outputObj = new Controller();
           new Producer(outputObj);
           new Consumer(outputObj);
        }
}

public class Controller
{
        int n;                                          // n is needed in get() and put()
        boolean flag = false;                           // Variable flag has class scope

        synchronized void get()
        {
           if( ! flag )
           {
           try    { wait(); }                                      // Call wait()
              catch(InterruptedException e)
              {
                 System.out.println("InterruptedException e caught");
              }
           }

           if(n < 10)   System.out.println("          GOT:  " + n);
           else         System.out.println("          GOT: " + n);
           flag = false;
        notify();                                       // Call notify( )
        }

        synchronized void put( int n )
        {
           if(flag)
           {
           try    { wait(); }                                      // Call wait()
              catch(InterruptedException e)
              { System.out.println("InterruptedException caught"); }
           }

           this.n = n;
           if(n < 10)   System.out.println("  put:  " + n);
           else         System.out.println("  put: " + n);
           flag = true;
        notify();                                                  // Call notify( )
        }
   }
```

```
public class Producer implements Runnable
{
        Controller  outputObj;
        Producer(Controller outputObj)              // Constructor method
        {
           this.outputObj = outputObj;
           new Thread(this, "Producer").start();
        }
        public void run()
        {
           int i = 0;
           while(true)
           {
              randomDelay();
              outputObj.put( i++ );
              if(i > 19) System.exit(0);
           }
        }

        void randomDelay()
        {
           int time = (int) ( ( Math.random() * 1730 )  + 30 );
           try
           {
                 Thread.sleep(time);
           }
           catch(InterruptedException e)
                 { System.out.println("Producer Interrupted"); }
           }
     }                                              // End of Producer class definition

public class Consumer implements Runnable
{
        Controller outputObj;
        Consumer(Controller outputObj)                      // Constructor method
        {
           this.outputObj = outputObj;
           new Thread(this, "Consumer").start();
        }

        public void run()    {
             while(true)     {
                   randomDelay();
                  outputObj.get();
             }
        }
        void randomDelay()          {
              int time = (int) ( ( Math.random() * 1730 ) + 30 );
              try               {
                    Thread.sleep(time);
              }
              catch(InterruptedException e )
              {          System.out.println("Consumer Interrupted");        }
        }
   }                                    // End of Consumer class definition
```

As this program runs, notice how that the Consumer will sometimes wait for the Producer and vice versa. You'll see this in the time it takes for the threads to respond to each other.

This is the output I got:

```
put:  0
                GOT:  0
put:  1
                GOT:  1
put:  2
                GOT:  2
put:  3
                GOT:  3
put:  4
                GOT:  4
```

```
put: 10
                GOT: 10
put: 11
                GOT: 11
put: 12
                GOT: 12
put: 13
                GOT: 13
put: 14
                GOT: 14
```

Note (by the output displayed) that the problem is solved. Every number that is produced by Producer is used by Consumer. Not one is wasted. Here's a modified figure of the various states. The Monitor states on the left have been added to what we had before.