COL106: Threads Tutorial

1 Processes and Threads

A process is an active program i.e. a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.

The operating system maintains management information about a process in a process control block (PCB). Modern operating systems allow a process to be divided into multiple threads of execution, which share all process management information except for information directly related to execution. This information is held in a thread control block (TCB).

There are several advantages to breaking down a process which include making the resultant thread lightweight, as the process in it's entirety would be very heavy and improving the overall throughput by increasing parallelism among other things.

1.1 Threads

Threads are mini-processes. They are created to execute multiple parts of a program concurrently. This paradigm of executing multiple parts of a program is called **Multithreading**.
A thread is a lightweight process that can be managed independently by a scheduler. It improves the application performance using parallelism. A thread shares information like data segment, code segment, files etc. with its peer threads while it contains its own registers, stack, counter etc.

Threads in a process can execute different parts of the program code at the same time. They can also execute the same parts of the code at the same time, but with different execution state:

- They have independent current instructions; that is, they have (or appear to have) independent program counters.
- They are working with different data; that is, they are (or appear to be) working with independent registers

Multithreading is an execution model that allows a single process to have multiple code segments (i.e., threads) run concurrently within the context of that process. You can think of threads as child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system.

1.2 Why Multithreading?

The ability of a program to concurrently execute multiple regions of code provides capabilities that are difficult or impossible to achieve with strictly sequential languages. Sequential object-oriented languages send messages (make method calls) and then block or wait for the operation to complete.
For example, Consider a simple scenario where a User is interacting with a GUI application. Let's say

the user issued a command which takes a long time to execute. Now if the application does not use multithreading i.e. the application is strictly sequential, then the GUI would be stuck and the User would not be able to interact with the application until the command execution is complete.
Such scenarios are not only common, but a norm in today's Computing Environment where Systems have multiple processors, and programs are written to use multiple Threads to maximize utilization of computing resources as well as improving User Experience.

2 Threads in Java

Java provides built-in support for multithreaded programming, i.e. it provides language level and library support for creating and managing threads out of the box.

2.1 Java Thread Model

In Java, a thread can exist in one of a set of states defined by the Java Thread Model. The states are-

- **New-** When an instance of the Thread class is created, it is in the **New** state.
- **Running-** As clear from the name, when a Java thread is executing it is in the **Running** state.
- **Suspended-** A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked-** A Java thread can be blocked when waiting for a resource. A suspended state first goes to the blocked state, and then returns to the runnable state once it acquires the lock on resources.
- **Terminated-** A thread can be **terminated** midway it's execution or once it completes execution. Once a thread is terminated it cannot resume execution.

Threads move from one state to another via a variety of means. The common methods for controlling a thread's state are shown in Figure 1. Below, we summarize these methods:

- **start():** A newborn thread with this method enter into Runnable state and Java run time create a system thread context and starts it running. This method for a thread object can be called only once.
- **stop():** This method causes a thread to stop immediately. This is often an abrupt way to end a thread.
- **suspend():** This method is different from stop() method. It takes the thread and causes it to stop running and later on can be restored by calling it again.
- **resume():** This method is used to revive a suspended thread. There is no guarantee that the thread will start running right way, since there might be a higher priority thread running already, but, resume() causes the thread to become eligible for running.
- **sleep(int n):** This method causes the run time to put the current thread to sleep for n milliseconds. After n milliseconds have expired, this thread will become eligible to run again.

Other complex functions such as yield, join, wait and notify will be explained later. First we'll take a look at the above mentioned basic functions and their usage.

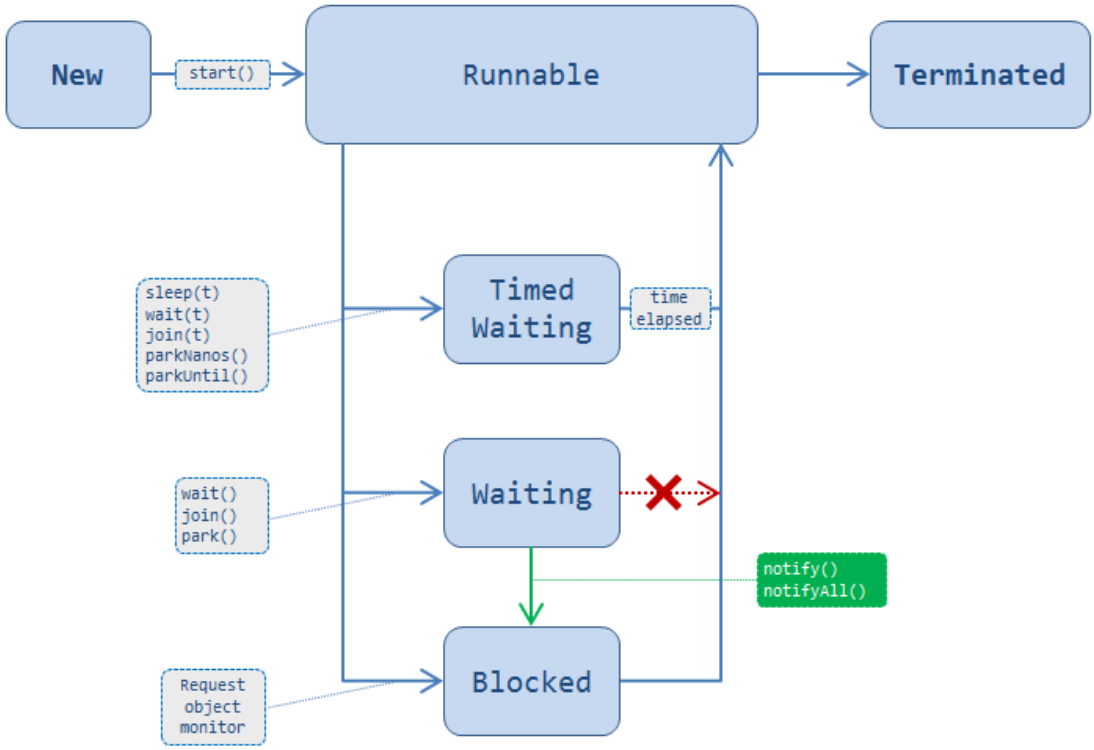

Figure 1: State Transition diagram of a Java Thread

2.2 Creating and Running a Thread

As with everything in Java, all functionality of a Thread is encapsulated in a class, called **Thread**. There are two ways in which you can create and use a Thread in Java, one is extending the Thread class and the other is implementing the Runnable Interface. We'll take a look at both.

2.2.1 Thread Class

First let's look at a simple example which extends the Thread class to use Multithreading.

Demonstration_1.java

```
1  class ThreadA extends Thread{
2      public void run( ) {
3          for(int i = 1; i <= 5; i++) {
4              System.out.println("From Thread A with i = "+ -1*i);
5          }
6          System.out.println("Exiting from Thread A ...");
7      }
8  }
9  class ThreadB extends Thread {
10     public void run( ) {
11         for(int j = 1; j <= 5; j++) {
12             System.out.println("From Thread B with j= "+2* j);
```

```
13          }
14          System.out.println("Exiting from Thread B ...");
15      }
16  }
17  class ThreadC extends Thread{
18       public void run( ) {
19            for(int k = 1; k <= 5; k++) {
20                 System.out.println("From Thread C with k = "+ (2*k-1));
21            }
22            System.out.println("Exiting from Thread C ...");
23       }
24  }
25
26  public class Demonstration_1{
27      public static void main(String args[]) {
28            ThreadA a = new ThreadA();
29            ThreadB b = new ThreadB();
30            ThreadC c = new ThreadC();
31            a.start();b.start();c.start();
32            System.out.println("... Multithreading is over ");
33      }
34  }
```

Output:

```
From Thread A with i = -1
From Thread A with i = -2
From Thread A with i = -3
From Thread B with j= 2
From Thread A with i = -4
From Thread A with i = -5
Exiting from Thread A ...
... Multithreading is over
From Thread C with k = 1
From Thread B with j= 4
From Thread B with j= 6
From Thread B with j= 8
From Thread B with j= 10
Exiting from Thread B ...
From Thread C with k = 3
From Thread C with k = 5
From Thread C with k = 7
From Thread C with k = 9
Exiting from Thread C ...
```

In the above simple example, three threads (all of them are of some type) will be executed concurrently. To use a thread, the code to run using the thread has to be put inside the **run()** function. To start running a thread, the **start()** function is called. This setups the Thread and then invokes the run() function.

The output shown is only one the many outputs that the above program could have produced. Since there is no synchronization or restriction on the order of execution of the threads, they run as and when they are scheduled.

2.2.2 Runnable Interface

Another method to create Threads is to use the Runnable interface. Objects of the class implementing the Runnable interface are passed as parameter to a Thread constructor to create a Thread.

Demonstration_2.java

```
1  class ThreadX implements Runnable{
2       public void run( ) {
3          for(int i = 1; i <= 5; i++) {
4             System.out.println("Thread X with i = "+ -1*i);
5          }
6         System.out.println("Exiting Thread X ...");
7       }
8  }
9
10 class ThreadY implements Runnable {
11      public void run( ) {
12          for(int j = 1; j <= 5; j++) {
13              System.out.println("Thread Y with j = "+ 2*j);
14          }
15         System.out.println("Exiting Thread Y ...");
16      }
17 }
18 class ThreadZ implements Runnable{
19      public void run( ) {
20          for(int k = 1; k <= 5; k++) {
21              System.out.println("Thread Z with k = "+ (2*k-1));
22          }
23              System.out.println("Exiting Thread Z ...");
24      }
25 }
26
27 public class Demonstration_2 {
28     public static void main(String args[]) {
29         ThreadX x = new ThreadX();
30         Thread t1 = new Thread(x);
31         ThreadY y = new ThreadY();
32         Thread t2 = new Thread(y);
33         Thread t3 = new Thread(new ThreadZ());
34
35         t1.start();t2.start();t3.start();
36         System.out.println("... Multithreading is over ");
37     }
38 }
```

Output:

```
Thread X with i = -1
Thread X with i = -2
Thread Z with k = 1
Thread Z with k = 3
Thread Z with k = 5
Thread Z with k = 7
Thread Z with k = 9
Exiting Thread Z ...
... Multithreading is over
Thread Y with j = 2
Thread Y with j = 4
Thread Y with j = 6
Thread Y with j = 8
Thread Y with j = 10
Exiting Thread Y ...
Thread X with i = -3
Thread X with i = -4
Thread X with i = -5
Exiting Thread X ...
```

As before, the output shown is only one of the many outputs possible. The key difference here to notice between the two ways of using Threads is that, while using the Runnable interface, an object of the class implementing the Runnable Interface must be passed as an argument to a Thread object to run it. Then the start function is called on the Thread. The JVM function knows that in this case, it needs to call the run function of the Runnable class and not the Thread object.

Of the two ways a Thread can be made, the Runnable interface method is preferred since it allows the class to extend to any other class that may be of use, which would not be possible if the Thread class is used.

2.3 Using Thread functions

Now that we understand the basics of creating a thread, we move on to understand the various functions to control the execution and life-cycle of a Thread.
We now explain what the yield function is used for-

- **yield():** Whenever a thread calls yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint. If any thread executes yield method, thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not current thread will keep executing.

We will try out the following example to understand the use of stop, yield and sleep methods.

Demonstration_3.java

```
1  class ClassA extends Thread{
2      public void run() {
3          System.out.println("Start Thread A ....");
4          for(int i = 1; i <= 5; i++) {
5              if (i==1) yield();
6              System.out.println("From Thread A: i = "+ i);
7          }
8          System.out.println("... Exit Thread A");
9      }
10 }
11
12 class ClassB extends Thread{
13     public void run() {
14         System.out.println("Start Thread B ....");
15         for(int j = 1; j <= 5; j++) {
16             System.out.println("From Thread B: j = "+ j);
17             if (j==2) stop();
18         }
19         System.out.println("... Exit Thread B");
20     }
21 }
22
23 class ClassC extends Thread{
24     public void run() {
25         System.out.println("Start Thread C ....");
26         for(int k = 1; k <= 5; k++) {
27             System.out.println("From Thread C: j = "+ k);
28             if (k==3){
29                 try{
30                     sleep(1000);
31                 }catch(Exception e){}
32             }
33         }
34         System.out.println("... Exit Thread C");
35     }
36 }
37
38 public class Demonstration_3{
39     public static void main (String args[]) {
40         ClassA t1 = new ClassA();
41         ClassB t2 = new ClassB();
42         ClassC t3 = new ClassC();
43         t1.start(); t2.start(); t3.start();
44         System.out.println("... End of execution ");
45     }
46 }
```

Output:

```
Start Thread A ....
Start Thread C ....
Start Thread B ....
... End of executuion
From Thread A: i = 1
From Thread B: j = 1
From Thread B: j = 2
From Thread C: j = 1
From Thread A: i = 2
From Thread A: i = 3
From Thread A: i = 4
From Thread A: i = 5
... Exit Thread A
From Thread C: j = 2
From Thread C: j = 3
From Thread C: j = 4
From Thread C: j = 5
... Exit Thread C
```

From the above output we can analyse the way the functions work.

- First t1 starts executing and as soon as the value of the loop variable (i) is equal to 1, the thread t1 yields and gives other threads a chance to execute. The Thread scheduler then selects t2 for execution.
- Execution of thread t2 goes upto 2 iterations until it is stopped. The Thread scheduler now has two options, t1 and t3. It chooses t3 for execution.
- t3 runs for 1 iteration before the control is transferred back to t1, which completes it's execution.
- t3 is now the only thread remaining which completes its execution after waiting for 1s(1000ms) after it's third iteration.

Again, this is only one of the many possible outputs the above code could have produced.
Now we move on to understand the join method.

- **join():** This method allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then t.join() will make sure that t is terminated before the next instruction is executed by the program.

We'll use the previous demonstration and add one extra line after starting the t1 thread.

Demonstration_4.java

```
1  class ClassA extends Thread {
2      public void run() {
3          System.out.println("Start Thread A ....");
4          for (int i = 1; i <= 5; i++) {
5              if (i == 1) yield();
6              System.out.println("From Thread A: i = " + i);
7          }
8          System.out.println("... Exit Thread A");
9      }
10 }
11
12 class ClassB extends Thread {
13     public void run() {
14         System.out.println("Start Thread B ....");
15         for (int j = 1; j <= 5; j++) {
16             System.out.println("From Thread B: j = " + j);
17             if (j == 2) stop();
18         }
19         System.out.println("... Exit Thread B");
20     }
21 }
22
23 class ClassC extends Thread {
24     public void run() {
25         System.out.println("Start Thread C ....");
26         for (int k = 1; k <= 5; k++) {
27             System.out.println("From Thread C: j = " + k);
28             if (k == 3) {
29                 try {
30                     sleep(1000);
31                 } catch (Exception e) {}
32             }
33         }
34         System.out.println("... Exit Thread C");
35     }
36 }
37
38 public class Demonstration_4 {
39     public static void main(String args[]) throws InterruptedException {
40         ClassA t1 = new ClassA();
41         ClassB t2 = new ClassB();
42         ClassC t3 = new ClassC();
43         t1.start();t1.join();t2.start();t3.start();
44         System.out.println("... End of execution ");
45     }
46 }
```

Output:

```
Start Thread A ....
From Thread A: i = 1
From Thread A: i = 2
From Thread A: i = 3
From Thread A: i = 4
From Thread A: i = 5
... Exit Thread A
... End of executuion
Start Thread C ....
Start Thread B ....
From Thread C: j = 1
From Thread B: j = 1
From Thread C: j = 2
From Thread B: j = 2
From Thread C: j = 3
From Thread C: j = 4
From Thread C: j = 5
... Exit Thread C
```

As always this is one of the many outputs the above code produces. But one thing that remains common in all of them is that, Thread t1 always finishes executing before threads t2 and t3 start. This is ensured by calling the join method called on t1.
Before moving on to the wait and notify methods, we will now try to understand the **synchronized** keyword in Java.

2.4 Synchronization

So far the code we have seen had multiple Threads running independent of each other. But what if there were some shared resources being accessed and modified between several threads? If the shared resource is accessed and modified at the same time by different threads, it could lead to unwanted random results. For example, if two threads are accessing the same Linked List, where one is trying to access the last node and the second is trying to delete it. In such a case we must prevent the second thread from modifying the list while the first is in the middle of reading it, to ensure there are no conflicts between the threads.

Since Multithreading introduces an asynchronous behaviour to programs, there must be a way to enforce synchronicity when needed. For this purpose, Java implements an way for synchronization called **monitor**. The monitor can be thought of as a box capable of holding only one thread at a time. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

There is no explicit class Monitor in Java to acquire and release objects. Instead every object in Java has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

When two or more threads need to access a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**.

The key to synchronization is the concept of a monitor, which as explained above is a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread aquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

There are two ways in which synchronization can be achieved, both use the **synchronized** keyword.

2.4.1 Synchronized Methods

As mentioned previously, all objects in Java have their own implicit monitor associated with them. To enter this monitor, we can call the any of its method which has been modified with the **synchronized** keyword. When a thread is in a synchronized method, all other threads trying to call any synchronized method of the same object, have to wait until the first thread completes it's execution of the synchronized method.

Let's try to understand this using an example.

Demonstration_5

```
1  class Receiver{
2      void call(String msg) {
3          System.out.print("[" + msg);
4          try {
5              Thread.sleep(1000);
6          }
7          catch(InterruptedException e) {
8              System.out.println("interrupted");
9          }
10         System.out.print("]");
11     }
12
13 }
14
15 class Caller implements Runnable{
16     Thread t;
17     String msg;
18     Receiver receiver;
19
20     Caller(String msg, Receiver receiver){
21         this.msg = msg;
22         this.receiver = receiver;
23         t = new Thread(this);
24         t.start();
25     }
26
27     public void run() {
28         receiver.call(msg);
29     }
30 }
31
32
33
```

```
34 public class Demonstration_5 {
35     public static void main(String args[]) throws InterruptedException {
36         Receiver r = new Receiver();
37         Caller c1 = new Caller("Hello", r);
38         Caller c2 = new Caller("Synchronized", r);
39         Caller c3 = new Caller("Method", r);
40         try {
41             c1.t.join();
42             c2.t.join();
43             c3.t.join();
44         }
45         catch(InterruptedException e) {
46             System.out.print("interrupted");
47         }
48     }
49 }
```

Output:

```
[Method[Synchronized[Hello]]]
```

Again this is one of the many outputs this program can have. This code does not do any kind of synchronization and as a result the three caller objects accessing the shared receiver object call the method call in an unsynchronized way, since a new msg starts printing before the ending bracket of the last message is printed.
If we now add the keyword **synchronized** before the method call-
synchronized void call(String msg).
We get the following result.
Output:

```
[Hello][Synchronized][Method]
```

The order of the words might be different, but notice that the brackets now end first and then a new message is printed. Since the receiver function is shared and its call method is synchronized, only one thread is able to access that method at a time.

2.4.2 Synchronized Statement

The above method works in the cases where you create the classes you want to multithread. But it won't work with classes that were not designed to work with multithreaded access i.e. the class does not use synchronized methods. Since the class is not created by you, it's source code is also hidden from you. So modifying the class by adding the synchronized modifier is off the table.

In such cases, there exists another rather simple method to achieve synchronization. Instead of modifying the methods to be synchronized you put the calls to them in a synchronized block. The general form of a synchronized block is-

```
1 synchronized(object){
2     // statements to be synchronized
3 }
```

Modifying the previous example using a synchronized block-

Demonstration_6.java

```
1  class Receiver{
2      void call(String msg) {
3          System.out.print("[" + msg);
4          try {
5              Thread.sleep(1000);
6          }
7          catch(InterruptedException e) {
8              System.out.println("interrupted");
9          }
10         System.out.print("]");
11     }
12 }
13
14 class Caller implements Runnable{
15     Thread t;
16     String msg;
17     Receiver receiver;
18
19     Caller(String msg, Receiver receiver){
20         this.msg = msg;
21         this.receiver = receiver;
22         t = new Thread(this);
23         t.start();
24     }
25
26     public void run() {
27         synchronized(receiver) { receiver.call(msg); }
28     }
29 }
30
31 public class Demonstration_6 {
32     public static void main(String args[]) throws InterruptedException {
33         Receiver r = new Receiver();
34         Caller c1 = new Caller("Hello", r);
35         Caller c2 = new Caller("Synchronized", r);
36         Caller c3 = new Caller("Method", r);
37         try {
38             c1.t.join();c2.t.join();c3.t.join();
39         }
40         catch(InterruptedException e) {
41             System.out.print("interrupted");
42         }
43     }
44 }
```

The above code has the same output as the synchronized method approach.

2.5 Inter Process Communication

The preceding examples of synchronization unconditionally blocked other threads from asynchronous access to certain methods. This use of implicit monitors in Java is powerful, but better results and finer control can be achieved using inter process communication.

Consider the classic scenario of the Producer-Consumer problem, where one thread or process is creating data and another thread or process is consuming it. The consumer has to wait until the producer has produced some data. For this, the consumer will run a loop checking the condition of the existence of produced data, wasting many CPU cycles in the process. If the producer also had to wait for the consumer to consume the data, then it would also run such a loop and waste many CPU cycles. This process of Polling is undesirable and can be avoided using Interprocess communication.

This IPC is achieved using the wait, notify and notifyall methods. These methods are final methods declared in the Object class, so every class has them. These methods can only be called from a synchronized context. The methods are explained below-

- **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 a thread that called **wait()** on the same object.
- **notifyAll():** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

First we'll consider an incorrect implementation of the Producer Consumer Solution.
Demonstration_7.java

```
1  class Q{
2      int n;
3
4      synchronized int get() {
5          System.out.println("Got:" + n);
6          return n;
7      }
8
9      synchronized void put(int n) {
10         this.n = n;
11         System.out.println("Put: " + n);
12     }
13 }
14 class Producer implements Runnable{
15     Q q;
16     Thread t;
17     Producer(Q q){
18         this.q = q;
19         t = new Thread(this);
20         t.start();
21     }
22
23
24
```

```
25      @Override
26      public void run() {
27          int i=0;
28
29          while(true) {
30              q.put(i++);
31          }
32      }
33  }
34
35  class Consumer implements Runnable{
36      Q q;
37      Thread t;
38
39      Consumer(Q q){
40          this.q = q;
41          t = new Thread(this);
42          t.start();
43      }
44
45      @Override
46      public void run() {
47          while(true) {
48              q.get();
49          }
50      }
51  }
52
53  public class Demonstration_7 {
54      public static void main(String args[]) throws InterruptedException {
55          Q q = new Q();
56          new Producer(q);
57          new Consumer(q);
58      }
59  }
```

Output:

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

As before this is only one of the possible outputs. As you can see that the Producer produced 1, and the consumer consumed 1, five times. Nothing is stopping the consumer from consuming the same value twice. And similarly nothing is stopping the Producer from overrunning the consumer.
In such cases, the wait and notify methods should be used. The correct implementation is-

Demonstration_8.java

```
1  class Q{
2      int n;
3      boolean valSet = false;
4      synchronized int get() {
5          while(!valSet) {
6              try {
7                  wait();
8              }
9              catch(InterruptedException e) {
10                 System.out.println("Interrupted");
11             }
12         }
13         System.out.println("Got:" + n);
14         valSet = false;
15         notify();
16         return n;
17     }
18
19     synchronized void put(int n) {
20         while(valSet) {
21             try{
22                 wait();
23             }
24             catch (InterruptedException e) {
25                 System.out.println("Interrupted");
26             }
27         }
28         this.n = n;
29         valSet = true;
30         System.out.println("Put: " + n);
31         notify();
32     }
33 }
34
35 class Producer implements Runnable{
36     Q q;
37     Thread t;
38     Producer(Q q){
39         this.q = q;
40         t = new Thread(this);
41         t.start();
42     }
```

```
43
44      @Override
45      public void run() {
46          int i=0;
47          while(true) {
48              q.put(i++);
49          }
50      }
51  }
52
53  class Consumer implements Runnable{
54      Q q;
55      Thread t;
56
57      Consumer(Q q){
58          this.q = q;
59          t = new Thread(this);
60          t.start();
61      }
62
63      @Override
64      public void run() {
65          while(true) {
66              q.get();
67          }
68      }
69  }
70
71  public class Demonstration_8 {
72      public static void main(String args[]) throws InterruptedException {
73          Q q = new Q();
74          new Producer(q);
75          new Consumer(q);
76      }
77  }
```

Output:

```
Put: 1
Get: 1
Put: 2
Get: 2
Put: 3
Get: 3
Put: 4
Get: 4
```

As you can see, now the producer only produces a new value if the consumer has consumed the previous value and the consumer only consumes one value once. Inside the get method, wait is called which makes

the consumer wait for the valSet boolean to be unset by put method i.e. consumer waits for value to be produced. Similarly inside the put method, wait is called which makes the producer wait for the valSet boolean to be set by the get method i.e. the producer waits for the previous value to be consumed by the consumer.

This producer consumer solution handles the production and consumption of only 1 item at a time. If the shared item is made to be a list or queue, then more than one item can be produced or consumed.

2.6 The Volatile keyword

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. If a field is declared volatile, in that case the Java memory model ensures that all threads see a consistent value for the variable. Consider the following class-

```
1 class SharedObj
2 {
3     // Changes made to sharedVar in one thread
4     // may not immediately reflect in other thread
5     static int sharedVar = 6;
6 }
```

If more than one thread is accessing the SharedObj, then it is possible that both of them have different values for the sharedVar variable. This is due to the fact that process or threads sometimes store the updated value of the variable in the cache and not in the main memory. Due to this, different threads might have different values for the same object or variable.

To resolve this issue we can make use of the **volatile** keyword. If the last code is modified to the following-

```
1 class SharedObj
2 {
3     // Changes made to sharedVar in one thread
4     // may not immediately reflect in other thread
5     static volatile int sharedVar = 6;
6 }
```

The volatile keyword can only be used with variables. Using it ensures that there is only one copy in the main memory of the variable which is accessible to all of the threads, i.e. threads do not have any local cached copy of the variable.