Chapter 6

Threads and Multithreading in Java

by Debasis Samanta

CONTENTS

Introduction

Multi-threading means multiple flow of control. Multi-threading programming is a conceptual paradigm for programming where one can divide a program into two or more processes which can be run in parallel. There are two main advantages of multi-threading : Fist, program with multiple threads will, in general, result in better utilization of system resources, including the CPU, because another line of execution can grab the CPU when one line of execution is blocked. Second, there are several problems better solved by multiple threads. For example, we can easily write a multi-threaded program to show animation, play music, display documents, and down load files from the network at the same time.

Java is a multi-threaded language. Java allows to write a program where more than one processes can be executed concurrently within the single program. Java's threads are often referred to as light weight threads, which means that they run in the same memory space. Because Java threads run in the same memory space, they can easily communicate among themselves because an object in one thread can call a method in another thread without any overhead from the operating system. In this Tutorial we will learn how to do multi-threaded programming in Java.

Basics of a thread

As with the Java concepts, everything about thread are defined in a class Thread. The Thread class encapsulates all of the control one will need over threads. The Thread class is our only link to manage how threads behave. In this Section, we will learn about : how to create and run a thread, the life cycle of a thread, and the thread controlling methods.

Creating and Running a Thread

There are two ways to define a thread: one is sub classing Thread and other is using the Runnable interface.

Using the sub classing thread : With this method, we have to define a class as a sub class of the Thread class. This sub class should contain a body which will be defined by a method run(). This run() method contains the actual task that the thread should perform. An instance of this sub class is then to be created by a new statement, followed by a call to the thread's start() method to have the run() method executed. Let us consider the Illustration 6.1 which includes a program to create there individual threads that each print out their own " Hello World !" string.

Illustration 6.1 // Creating and running threads using sub classing Thread /

```
/* Creating three threads using the class Thread and then running them concurrently. */
class ThreadA extends Thread{
     public void run( ) {
        for(int i = 1; i <= 5; i++) {
           System.out.println("From Thread A with i = "+ -1*i);
        }
        System.out.println("Exiting from Thread A ...");
     }
}

class ThreadB extends Thread {
    public void run( ) {
       for(int j = 1; j <= 5; j++) {
         System.out.println("From Thread B with j= "+2* j);
       }
       System.out.println("Exiting from Thread B ...");
        }
}
class ThreadC extends Thread{
     public void run( ) {
         for(int k = 1; k <= 5; k++) {
             System.out.println("From Thread C with k = "+ (2*k-1));
         }
         System.out.println("Exiting from Thread C ...");
     }
}

public class Demonstration_111 {
    public static void main(String args[]) {
         ThreadA a = new ThreadA();
         ThreadB b = new ThreadB();
         ThreadC c = new ThreadC();
         a.start();
         b.start();
         c.start();
         System.out.println("... Multithreading is over ");
    }
}
```

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. Note that a thread can be directed to start its body by start() method.

Using the Runnable interface : A second way to create threads is to make use of the Runnable interface. With this approach, first we have to implement the Runnable interface.[Runnable interface is already defined in the system package **java.lang** with a single method **run()** as below :

```
public interface Runnable {
                        public abstract void run( );
                }  ]
```

When we will create a new thread, actually a new object will be instantiated from this Runnable interface as the target of our thread, meaning that the thread will look for the code for the run( ) method within our object's class instead of inside the Thread's class. This is illustrated with an example where two processes Brother and Sister will be executed simultaneously.

Illustration 6.2

```
/* Creating three threads using the Runnable interface and then running them concurrently. */
class ThreadX implements Runnable{
     public void run( ) {
        for(int i = 1; i <= 5; i++) {
           System.out.println("Thread X with i = "+ -1*i);
        }
       System.out.println("Exiting Thread X ...");
     }
}

class ThreadY implements Runnable {
     public void run( ) {
         for(int j = 1; j <= 5; j++) {
             System.out.println("Thread Y with j = "+ 2*j);
         }
        System.out.println("Exiting Thread Y ...");
     }
}
class ThreadZ implements Runnable{
     public void run( ) {
         for(int k = 1; k <= 5; k++) {
             System.out.println("Thread Z with k = "+ (2*k-1));
         }
             System.out.println("Exiting Thread Z ...");
     }
}

public class Demonstration_112 {
    public static void main(String args[]) {
                ThreadX x = new ThreadX();
                Thread t1 = new Thread(x);

                ThreadY y = new ThreadY();
                Thread t2 = new Thread(y);

                //ThreadZ z = new ThreadZ();
                //Thread t3 = new Thread(z);
                Thread t3 = new Thread(new ThreadZ());

                t1.start();
                t2.start();
                t3.start();

        System.out.println("... Multithreading is over ");
    }
}
```

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 ...
```

Note : Note in the above example, how after implementing objects, their thread is created and their threads start execution. Also note that, a class instance with the run( ) method defined within must be passed in as an argument in creating the thread instance so that when the start() method of this Thread instance is called, Java run time knows which run() method to execute. This alternative method of creating a thread is useful when the class defining run() method needs to be a sub class of other classes; the class can inherit all the data and methods of the super class.

Life cycle of threads

Each thread is always in one of five states, which is depicted in **Figure 6.1**

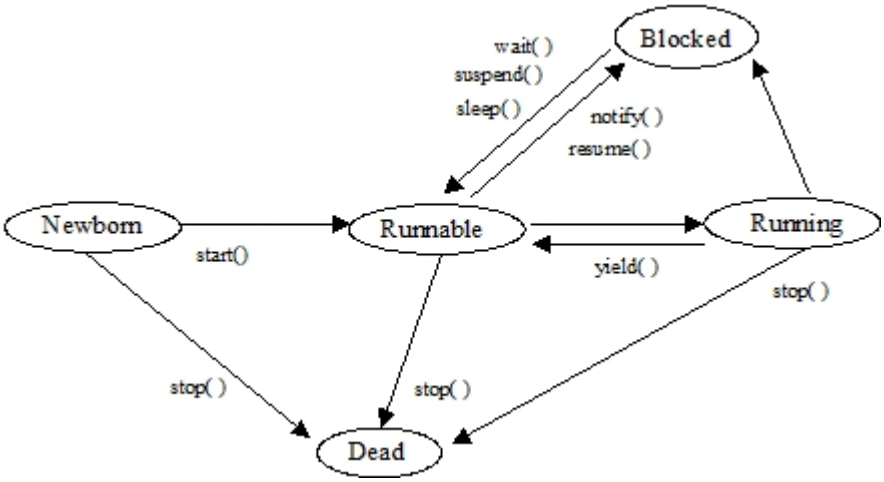


Figure 6.1 : Five states of a thread

Newborn : When a thread is created (by new statement) but not yet to run, it is called in Newborn state. In this state, the local data members are allocated and initialized.

Runnable : The Runnable state means that a thread is ready to run and is awaiting for the control of the processor, or in other words, threads are in this state in a queue and wait their turns to be executed.

Running : Running means that the thread has control of the processor, its code is currently being executed and thread will continue in this state until it get preempted by a higher priority thread, or until it relinquishes control.

Blocked : A thread is Blocked means that it is being prevented from the Runnable (or Running) state and is waiting for some event in order for it to reenter the scheduling queue.

Dead : A thread is Dead when it finishes its execution or is stopped (killed) by another thread.

Threads move from one state to another via a variety of means. The common methods for controlling a thread's state is shown in **Figure 6.1.** Below, we are to 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 once only

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 gurantee 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 elligible to run again.

yield( ) : The yield() method causes the run time to switch the context from the current thread to the next available runnable thread. This is one way to ensure that the threads at lower priority do not get started.

Other methods like **wait(), notify(), join()** etc. will be discussed in subsequent discussion. Let us illustrate the use of these method in a simple Application.

Illustration 6.3

```
/* Demonstration of thread class methods : getID() */
/* Java code for thread creation by extending the Thread class */

class ThreadId extends Thread {
    public void run()  {
        try {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() + " is running");
        }
        catch (Exception e)  {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

public class Demonstration_113{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            ThreadId object = new ThreadId();
            object.start();
        }
    }
}
```

OUTPUT:

```
Thread 21 is running
Thread 22 is running
Thread 23 is running
Thread 25 is running
Thread 26 is running
Thread 27 is running
Thread 24 is running
Thread 28 is running
```

Illustration 6.4

```
/* Demonstration of thread class methods : getID() */
/* Java code for thread creation by implementing the Runnable Interface */

class ThreadId implements Runnable
{
    public void run()     {
        try  {
            // Displaying the thread that is running
            System.out.println ("Thread " + Thread.currentThread().getId() + " is running");
          }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
class Demonstration_114  {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++){
                        Thread object = new Thread(new ThreadId());
                        object.start();
        }
    }
}
```

OUTPUT:

```
Thread 21 is running
Thread 23 is running
Thread 24 is running
Thread 25 is running
Thread 22 is running
Thread 27 is running
Thread 26 is running
Thread 28 is running
```

Illustration 6.5

```
/* Use of yield(), stop() and sleep() methods */

class ClassA extends Thread{
        public void run() {
                System.out.println("Start Thread A ....");
                for(int i = 1; i <= 5; i++) {
                        if (i==1) yield();
                        System.out.println("From Thread A: i = "+ i);
                }
                System.out.println("... Exit Thread A");
        }
}

class ClassB extends Thread{
        public void run() {
                System.out.println("Start Thread B ....");
                for(int j = 1; j <= 5; j++) {
                        System.out.println("From Thread B: j = "+ j);
                        if (j==2) stop();
                }
                System.out.println("... Exit Thread B");
        }
}

class ClassC extends Thread{
        public void run() {
                System.out.println("Start Thread C ....");
                for(int k = 1; k <= 5; k++) {
                        System.out.println("From Thread B: j = "+ k);
                        if (k==3){
                                try{
                                        sleep(1000);
                                }catch(Exception e){}
                        }
                }
                System.out.println("... Exit Thread C");
        }
}


public class Demonstration_115{
        public static void main (String args[]) {
            ClassA t1 = new ClassA();
                ClassB t2 = new ClassB();
                ClassC t3 = new ClassC();
                t1.start(); t2.start(); t3.start();
                System.out.println("... End of executuion ");
        }
}
```

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 B: 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 B: j = 2
From Thread B: j = 3
From Thread B: j = 4
From Thread B: j = 5
... Exit Thread C
```

Illustration 6.6

```
/* Use of suspend() and resume() methods */

class Thread1 extends Thread {
        public void run( ) {
                try{
                        System.out.println (" First thread starts running" );
                        sleep(10000);
                        System.out.println (" First thread finishes running" );
                }
                catch(Exception e){     }
        }
}

class Thread2 extends Thread {
        public void run( ) {
                try{
                        System.out.println ( "Second thread starts running");
                        System.out.println ( "Second thread is suspended itself ");
                        suspend( );
                        System.out.println (" Second  thread runs again" );
                }
                catch(Exception e){     }
        }
}

class Demonstration_116{
        public static void main (String args[ ] ){
                try{
                        Thread1 first = new Thread1( );  // It is a newborn thread i.e. in Newborn st
                        Thread2 second= new Thread2( );  // another new born thread

                        first.start( );    // first is scheduled  for running
                        second.start( );   // second is scheduled for running

                        System.out.println("Revive the second thread" );  // If it is suspended
                        second.resume( );

                        System.out.println ("Second thread went for 10 seconds sleep " );
                        second.sleep (10000);

                        System.out.println ("Wake up second thread and finishes running" );
                        System.out.println ( " Demonstration is finished ");
                }
                catch(Exception e){     }
        }
}
```

OUTPUT:

```
Revive the second thread
First thread starts running
Second thread starts running
Second thread is suspended itself
Second thread went for 10 seconds sleep
```

Status of a Thread

It is some time essential to know some information about threads. There are number of methods defined in Thread which can be called for getting information about threads. Some of the most commonly used methods for thread's status are listed here :

currentThread( ) : The CurrentThread() is a static method returns the Thread object which is the currently running thread.
setName(String s) : The SetName() method is to assign a name s for a thread prior its execution. This, therefore, identifies the thread with a string name. This is helpful for debugging multi-threaded programs.
getName( ) : This method returns the current string value of the thread's name as set by SetName().
setPriority (int p) : This method sets the thread's priotity to an integer value p passed in. There are several predefined priotiry constants defined in class Thread : **MIN-PRIORITY, NORM-PRIORTY and MAX-PRIORITY,** which are 1, 5, and 10 respectively.
getPriority ( ) : This method returns the thread's current priority, a value between 1 and 10.
isAlive ( ) : This method returns true if the thread is started but not dead yet.
isDaemon ( ) : This method returns true if the thread is a daemon thread.

Following is the **Illustration 6.7** to give an idea how the above mentioned method may be utilized.

Illustration 6.7 // Status information of threads //

```
/* Setting priority to threads */

class ClassA extends Thread{
        public void run() {
                System.out.println("Start Thread A ....");
                for(int i = 1; i <= 5; i++) {
                        System.out.println("From Thread A: i = "+ i);
                }
                System.out.println("... Exit Thread A");
        }
}

class ClassB extends Thread{
public void run() {
        System.out.println("Start Thread B ....");
        for(int j = 1; j <= 5; j++) {
                System.out.println("From Thread B: j = "+ j);
        }
        System.out.println("... Exit Thread B");
        }
}

class ClassC extends Thread{
        public void run() {
        System.out.println("Start Thread C ....");
        for(int k = 1; k <= 5; k++) {
                System.out.println("From Thread B: j = "+ k);
        }
        System.out.println("... Exit Thread C");
        }
}


class Demonstration_117{
        public static void main (String args[]) {
                ThreadA t1 = new ThreadA();
                ThreadB t2 = new ThreadB();
                ThreadC t3 = new ThreadC();

                t3.setPriority(Thread.MAX_PRIORITY);
                t2.setPriority(t2.getPriority() + 1);
                t1.setPriority(Thread.MIN_PRIORITY);

                t1.start(); t2.start(); t3.start();
                System.out.println("... End of executuion ");
        }
}
```

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 B ....
Start Thread C ....
From Thread B: j = 1
From Thread B: j = 2
From Thread B: j = 3
From Thread B: j = 4
From Thread B: j = 5
... Exit Thread B
From Thread B: j = 1
From Thread B: j = 2
From Thread B: j = 3
From Thread B: j = 4
From Thread B: j = 5
... Exit Thread C
```

Illustration 6.8

```
/* Data race example. */

public class Demonstration_118 extends Thread {
    public static int x;
    public void run() {
        for (int i = 0; i < 100; i++) {
            x = x + 1;
            x = x - 1;
        }
    }
    public static void main(String[] args) {
         x = 0;
         for (int i = 0; i < 1000; i++){
            new Demonstration_118().start();
                        System.out.println(x);  // x not always 0!
                 }
    }
}
```

Synchronization and Inter-Thread Communication

It is already mentioned that threads in Java are running in the same memory space, and hence it is easy to communicate between two threads. Inter-thread communications allow threads to talk to or wait on each other. Again, because all the threads in a program share the same memory space, it is possible for two threads to access the same variables and methods in object. Problems may occur when two or more threads are accessing the same data concurrently, for example, one thread stores data into the shared object and the other threads reads data, there can be synchronization problem if the first thread has not finished

storing the data before the second one goes to read it. So we need to take care to access the data by only one thread process at a time. Java provides unique language level support for such synchronization. in this Section we will learn how synchronization mechanism and inter-thread communications are possible in Java.

Synchronization

To solve the critical section problem, one usual concept is known what is called monitor. A monitor is an object which is used as a mutually exclusive lock (called mutex). Only one thread may own a monitor at a given time. When a thread acquires a lock it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the owner thread exits the monitor. But in Java, there is no such monitor. In fact, all Java object have their own implicit monitor associated with them. Here, the key word synchronized is used by which method (s) or block of statements can be made protected from the simultaneous access. With this, when a class is designed with threads in mind, the class designer decides which methods should not be allowed to execute concurrently. when a class with synchronized method is instantiated, the new object is given its own implicit monitor. The entire time that a thread is inside of a synchronized method, all other threads that try to call any other synchronized method on the same instance have to wait. In order to exit the monitor and relinquish control of the object to the next waiting thread the monitor owner simply needs to return from the method.

Let us illustrate this mechanism with a simple example.

Suppose, we want to maintain a bank account of customers. Several transactions, such as deposite some amount to an account and withdraw some amount from an account etc. are possible. Now, for a given account, if two or more transactions come simultaneously then only one transaction should be allowed at a time instead of simulataneous transaction processing so that data inconsistency will never occur. So, what we need is to synchronize the transaction. **Illustration 6.9** is to implement such a task.

Illustration 6.9

```
/* The following Java application shows how the transactions in a bank can be carried out concurrentl
class Account {
        public int balance;
        public int accountNo;
        void displayBalance() {
                System.out.println("Account No:" + accountNo + "Balance: " + balance);
        }

           synchronized void deposit(int amount){
                        balance = balance + amount;
                        System.out.println( amount + " is deposited");
                        displayBalance();
           }

           synchronized void withdraw(int amount){
                         balance = balance - amount;
                         System.out.println( amount + " is withdrawn");
                         displayBalance();
           }
}

class TransactionDeposit implements Runnable{
        int amount;
        Account accountX;
        TransactionDeposit(Account x, int amount){
                accountX = x;
                this.amount = amount;
                new Thread(this).start();
        }

        public void run(){
                accountX.deposit(amount);
        }
}

class TransactionWithdraw implements Runnable{
        int amount;
        Account accountY;

        TransactionWithdraw(Account y, int amount) {
                accountY = y;
                this.amount = amount;
                new Thread(this).start();
        }

        public void run(){
                accountY.withdraw(amount);
        }
}

class Demonstration_119{
        public static void main(String args[]) {
                Account ABC = new Account();
                ABC.balance = 1000;
                ABC.accountNo = 111;
                TransactionDeposit t1;
                TransactionWithdraw t2;
                t1 = new TransactionDeposit(ABC, 500);
```

```
                t2 = new TransactionWithdraw(ABC,900);
        }
}
```

OUTPUT:

```
500 is deposited
Account No:111Balance: 1500
900 is withdrawn
Account No:111Balance: 600
```

In the above example, the keyword synchronized is used for the methods **void deposite(..) and void withdraw(�)** so that these two methods will never run for the same object instance simultaneously.

Alternatively, if one wants to design a class that was not designed for multi-thread access and thus has non-synchronized methods, then it can be wrapped the call to the methods in a synchronized block. Here is the general form of the synchronized statement :

```
synchronized (Object ) { block of statement(s) }
```

where Object is any object reference. For example, make all the methods in Account class as non-synchronized (remove the synchronized key word). Then modify the code for run( ) in class TransactionDeposite and class TransactionWithdraw are as under :

```
    public void run( ) {                // in TransactionDeposite
           synchronized (accountX ) {
                  accountX.deposite(amount );
                }
        }

   public void run( )   {               // in TransactionWithdraw
          synchronized (accountY )       {
                    accountY.withdraw(amount );
                }
        }
```

You will get the same output.

Interested reader may try to run the program without using synchronization and observe the result.

Note : In a class, non-synchronized methods are concurrently executable.

Inter-thread Communication

There are three ways for threads to communicate with each other. The first is through commonly shared data. All the threads in the same program share the same memory space. If an object is accessible to various threads then these threads share access to that object's data member and thus communicate each other.

The second way for threads to communicate is by using thread control methods. There are such three methods by which threads communicate for each other :

suspend ( ): A thread can suspend itself and wait till other thread resume it.
resume ( ): A thread can wake up other waiting thread (which is waiting using suspend()) through its resume() method and then can run concurrently.
join ( ) :This method can be used for the caller thread to wait for the completion of called thread.

The third way for threads to communicate is the use of three methods; **wait(), notify(), and notifyAll();** these are defined in class Object of package **java.lang.** Actually these three methods provide an elegant inter-process communication mechanism to take care the deadlock situation in Java. As there is multi-threading in program, deadlock may occur when a thread holding the key to monitor is suspended or waiting for another thread's completion. If the other thread is waiting for needs to get into the same monitor, both threads will be waiting for ever. The uses of these three methods are briefed as below :

wait ( ):This method tells the calling thread to give up the monitor and make the calling thread wait until either a time out occurs or another thread call the same thread's notify() or notifyAll() method.
Notify ( ): This method wakes up the only one (first) waiting thread called wait() on the same object.
notifyAll( ): This method will wake up all the threads that have been called wait( ) on the same object.

Now, let us demonstrate the classical use of these methods. **Illustration 6.6** is for this purpose.

Illustration 6.10 : // Inter thread communication : Producer & Consumer problem //

```
    class Q {                       // Q is a class containing two parallel processes
            int n;
            boolean flag = false;
//PRODUCER
            synchronized void put( int n) { // Produce a value
                    if(flag) {                                                  // Entry
                       try  wait( );  catch(InterruptedException e);            // to the
                    }                                                   // critical section

                        this.n = n;
                        System.out.println( "Produce :" + n);                   // Critical S

                        flag = true;                                            // Exit from
                        notify( );                                              // critical
                }
//CONSUMER
           synchronized int get( ) {                // Consume a value
                    if(! flag) {                                        // Entry
                      try  wait( );  catch(InterruptedException e);     // to the
                    }                                                   // critical section

                    System.out.println( "Consume :" + n);               // Critical Section

                    flag = false;                                       // Exit from the
                    notify( );                                                  // critical
                 return( n );
            }


    class Producer implement Runnable  {    // Thread for Producer process
                     Q  q;
             Producer ( Q q )       {       // constructor
             this.q =q;
             new thread (this).start ( ) ;          // Producer process is started
            }

       public void run( ) {                 // infinite running  thread for Producer
            int i = 0;
            while (true )
                    q.put ( i++ );
               }
            }

    class Consumer implement Runnable {     // Thread for consumer process
            Q q;
       Consumer (Q q )      {                 // Constructor
       this.q  = q;
       new Thread (this).start ( );
            }

     public void run( ) {           // infinite running thread for Consumer
     while (true)
            q.get ( );
            }
    }

   class PandC  {
```

```
public static void main( String args[ ] ) {
Q q = new Q( );         // an instance of parallel processes  is created
new Producer(q) ;                       // Run the thread for producer
new Consumer (q);                       // Run consumer thread
   }
}
```

To understand this critical section problem in operating system design, user may be referred to : *Operating system concept by Peterson and Sylberchotze , Addiction Wesley Inc.*

Note : All three methods i.e. wait(), notify(), and notifyAll() must only be called from the inside of synchronized methods.

Thread Groups and Daemon

There are two other variation in thread class known : ThreadGroup and Daemon. ThreadGroup, as its name implies, is a group of threads. A thread group can have both threads and other thread groups as its member elements. In Java, there is a default thread group called SystemThreadGroup, which is nothing but the Java run time itself. When a Java application is started, the Java run time creates the main thread group as a member of the system thread group. A main thread is created in the main thread group to run the main( ) method of the Application. In fact, every thread instance is member of exactly one thread group. By default, all new user created threads and thread groups) will become the members of the main thread group. All the threads and thread in an application form a tree with the system thread group as the root.

Daemon threads are service threads. They exist to provide service threads. They exist to provide services to other threads. They normally run in an infinite loop and attending the client threads requesting services. When no other threads exist daemon thread is automatically ceased to exist.

A new thread group can be created by instantiating the thread group class. For example,

```
Threadgroup TG = new ThreadGroup ( ) ;
Thread T = new Thread ( TG) ;
```

These two statements creates a new thread group TG which contains a thread T as the only member.

To create a daemon thread, there is a method setDaemon() can be called just after the creation of a thread and before the execution is started. For example, following two statement is to make a thread as demon thread.

```
Thread T = new Thread ( ) ;
T setDaemon (true);
```

The constructor of the thread is a good candidate for making this method call, Also, by default, all the threads created by a daemon thread are also daemon thread.
Some commonly used methods for handling thread groups and daemon are listed below :

getName ( ) :Returns the name of the thread group .
setName ( ) :Sets the name of the thread group .
geParent ( ) :Returns the parent thread group of the thread group .
getMaxPriority :Returns the current maximum priority of the thread group.
activeCount ( ) :Returns the number of active threads in the thread group.
activeGroupCount :Returns the number of active threads groups in the thread group.
isDaemon ( ) :Returns true if the thread is a daemon thread.
setDaemon ( ) : Set the thread as a daemon thread prior its starting execution.

Practice Questions

Practice 6.1

```
        /* Practice of a multithreaded program using subclassing Thread */
        class ThreadA extends Thread{
                public void run( ) {
                for(int i = 1; i <= 5; i++) {
                System.out.println("From Thread A with i = "+ -1*i);
                }
                System.out.println("Exiting from Thread A ...");
                }
                }

                class ThreadB extends Thread{
                public void run( ) {
                for(int j = 1; j <= 5; j++) {
                System.out.println("From Thread B with j = "+ 2*j);
                }
                System.out.println("Exiting from Thread B ...");
                }
                }

                class ThreadC extends Thread{
                public void run( ) {
                for(int k = 1; k <= 5; k++) {
                System.out.println("From Thread C with k = "+ (2*k-1));
                }
                System.out.println("Exiting from Thread C ...");
                }
                }

                class MultiThreadClass{
                public static void main(String args[]) {
                ThreadA a = new ThreadA();
                ThreadB b = new ThreadB();
                ThreadC c = new ThreadC();

                a.start();
                b.start();
                c.start();

                System.out.println("... Multithreading is over ");
                }
                }
```

Practice 6.2

```
        /* Practice of a multithreaded program using Runnable interface */
                class ThreadX implements Runnable{
                public void run( ) {
                for(int i = 1; i <= 5; i++) {
                System.out.println("Thread X with i = "+ i);
                }
                System.out.println("Exiting Thread X ...");
                }
                }

                class ThreadY implements Runnable{
                public void run( ) {
                for(int j = 1; j <= 5; j++) {
                System.out.println("Thread Y with j = "+ j);
                }
                System.out.println("Exiting Thread Y ...");
                }
                }


                class ThreadZ implements Runnable{
                public void run( ) {
                for(int k = 1; k <= 5; k++) {
                System.out.println("Thread Z with k = "+ k);
                }
                System.out.println("Exiting Thread Z ...");
                }
                }

                class MultiThreadRunnable{
                public static void main(String args[]) {
                ThreadX x = new ThreadZ(); Thread t1 = new Thread(x);
                ThreadY y = new ThreadY(); Thread t2 = new Thread(y);
                ThreadZ z = new ThreadZ(); Thread t3 = new Thread(z);
                t1.start();
                t2.start();
                t3.start();
                System.out.println("... Multithreading is over ");
                }
        }
```

Practice 6.3

```
	/* Use of yield(), stop() and sleep() methods */

	class ClassA extends Thread{

		public void run() {

			System.out.println("Start Thread A ....");

			for(int i = 1; i <= 5; i++) {

				if (i==1) yield();

				System.out.println("From Thread A: i = "+ i);

			}

			System.out.println("... Exit Thread A");

		}

	}

	class ClassB extends Thread{

		public void run() {

			System.out.println("Start Thread B ....");

			for(int j = 1; j <= 5; j++) {

				System.out.println("From Thread B: j = "+ j);

				if (j==2) stop();

			}

			System.out.println("... Exit Thread B");

		}

	}


	class ClassC extends Thread{

		public void run() {

			System.out.println("Start Thread C ....");

			for(int k = 1; k <= 5; k++) {

				System.out.println("From Thread B: j = "+ k);

				if (k==3){

					try{
```

```
                                        sleep(1000);

                                }catch(Exception e){}

                        }

                }

                System.out.println("... Exit Thread C");

        }

}


class ThreadControl{

        public static void main (String args[]) {

                ThreadA t1 = new ThreadA();

                ThreadB t2 = new ThreadB();

                ThreadC t3 = new ThreadC();

                t1.start(); t2.start(); t3.start();

                System.out.println("... End of executuion ");

        }

}
```

Practice 6.4

```
        /* Use of suspend() and resume() methods */
                class Thread1 extends Thread {
                public void run( ) {
                System.out.println (" First thread starts running" );
                sleep(10000);
                System.out.println (" First thread finishes running" );
                }
                }

                class Thread2 extends Thread {
                public void run( ) {
                System.out.println ( "Second thread starts running");
                System.out.println ( "Second thread is suspended itself ");
                suspend( );
                System.out.println (" Second  thread runs again" ));
                }
                }

                class AnotherThreadControl {
                public static void main (String, args[ ] ) {
                Thread1 fist = new Thread1( );  // It is a newborn thread i.e. in Newborn state
                Thread2 second= new Thread2( );  // another new born thread
                first.start( );    // first is scheduled  for running
                second.start( );   // second is scheduled for running

                System.out.println("Revive the second thread" );  // If it is suspended
                second.resume( );
                System.out.println ("Second thread went for 10 seconds sleep " );
                Second.sleep (10000);
                System.out.println ("Wake up second thread and finishes running" );
                System.out.println ( " Demonstration is finished ");
                }
                }
```

Practice 6.5

```
/* Setting priority to threads */
                class ClassA extends Thread{
                public void run() {
                System.out.println("Start Thread A ....");
                for(int i = 1; i <= 5; i++) {
                System.out.println("From Thread A: i = "+ i);
                }
                System.out.println("... Exit Thread A");
                }
                }

                class ClassB extends Thread{
                public void run() {
                System.out.println("Start Thread B ....");
                for(int j = 1; j <= 5; j++) {
                System.out.println("From Thread B: j = "+ j);
                }
                System.out.println("... Exit Thread B");
                }
                }

                class ClassC extends Thread{
                public void run() {
                System.out.println("Start Thread C ....");
                for(int k = 1; k <= 5; k++) {
                System.out.println("From Thread B: j = "+ j);
                }
                System.out.println("... Exit Thread C");
                }
                }


                class ThreadPriorityTest{
                public static void main (String args[]) {
                TheadA t1 = new ThreadA();
                TheadB t2 = new ThreadB();
                TheadC t3 = new Thread3();

                t3.setPriority(Thread.MAX_PRIORITY);
                t2.setPriority(Thread.getPriority() + 1);
                t1.setPriority(Thread.MIN_PRIORITY);
                t1.start(); t2.start(); t3.start();
                System.out.println("... End of executuion ");
        }
   }
```

Practice 6.6

```
/* Following Java application create a list of numbers and then sort in ascending order as wel
    import java.util.*;

        class Numbers {
        public int result[] = new int[10];
        void displayListOfNos()
        {
        System.out.println("Numbers stored in the array:");
        for( int idx=0; idx<10; ++idx) {
        System.out.println(result[idx]);
        }
        }

        void fillTheArray(int aUpperLimit, int aArraySize)
        {
          if (aUpperLimit <=0) {
        throw new IllegalArgumentException("UpperLimit must be positive: " + aUpperLimit);
        }

        if (aArraySize <=0) {
        throw new IllegalArgumentException("Size of returned List must be greater than 0.");
        }

        Random generator = new Random();
        for( int idx=0; idx result[j]) {
        int temp = result[i];
        result[i] = result[j];
        result[j] = temp;
        }
        }
        }

        displayListOfNos();
        }
        }
        class ArrangementAscending implements Runnable {
        Numbers n1 ;
        ArrangementAscending(Numbers n) {
        n1 = n;
        new Thread(this).start();
        }
        public void run() {
        n1.sortAscending();
        }
        }

        class ArrangementDescending implements Runnable {
        Numbers n2;
        ArrangementDescending(Numbers n) {
        n2 = n;
        new Thread(this).start();
        }


        public void run() {
        n2.sortDescending();
        }
        }
```

```
class ArrangingNos {

public static void main(String args[]) {
Numbers n = new Numbers();
n.fillTheArray(20,10);
ArrangementAscending  a1 = new ArrangementAscending(n);
ArrangementDescending d1 = new ArrangementDescending(n);
}
}
```

Practice 6.7

```
/* The following Java application shows how the transactions in a bank can be carried out concu
        class Account {
        public int balance;
        public int accountNo;
        void displayBalance()
        {
        System.out.println("Account No:" + accountNo + "Balance: " + balance);
        }


        synchronized void deposit(int amount)
        {
        balance = balance + amount;
        System.out.println( amount + " is deposited");
        displayBalance();
        }

        synchronized void withdraw(int amount)
        {
        balance = balance - amount;
        System.out.println( amount + " is withdrawn");
        displayBalance();
        }
        }

        class TransactionDeposit implements Runnable
        {
        int amount;
        Account accountX;
        TransactionDeposit(Account x, int amount)
        {
        accountX = x;
        this.amount = amount;
        new Thread(this).start();
        }


        public void run()
        {
        accountX.deposit(amount);
        }
        }

        class TransactionWithdraw implements Runnable
        {
        int amount;
        Account accountY;
        TransactionWithdraw(Account y, int amount) {
        accountY = y;
        this.amount = amount;
        new Thread(this).start();
        }
        public void run()
        {
        accountY.withdraw(amount);
        }
        }
```

```
class Transaction {
public static void main(String args[]) {
Account ABC = new Account();
ABC.balance = 1000;
ABC.accountNo = 111;
TransactionDeposit t1;
TransactionWithdraw t2;
t1 = new TransactionDeposit(ABC, 500);
t2 = new TransactionWithdraw(ABC,900);
}
}
```

Assignment

- **Q: Write a Java program which handles Push operation and Pop operation on stack concurrently.**
- **Q: Write a Java program which first generates a set of random numbers and then determines negative, positive even, positive odd numbers concurrently.**

Q&A

Q: What is the thread?

A: A thread is a lightweight subprocess. It is a separate path of execution because each thread runs in a different stack frame. A process may contain multiple threads. Threads share the process resources, but still, they execute independently..

Q: How to implement Threads in java?

A: Threads can be created in two ways i.e. by implementing **java.lang.Runnable interface or extending java.lang.Thread class and then extending run method.**
Thread has its own variables and methods, it lives and dies on the heap. But a thread of execution is an individual process that has its own call stack. Thread are lightweight process in java.
1)Thread creation by implementingjava.lang.Runnableinterface. We will create object of class which implements Runnable interface :

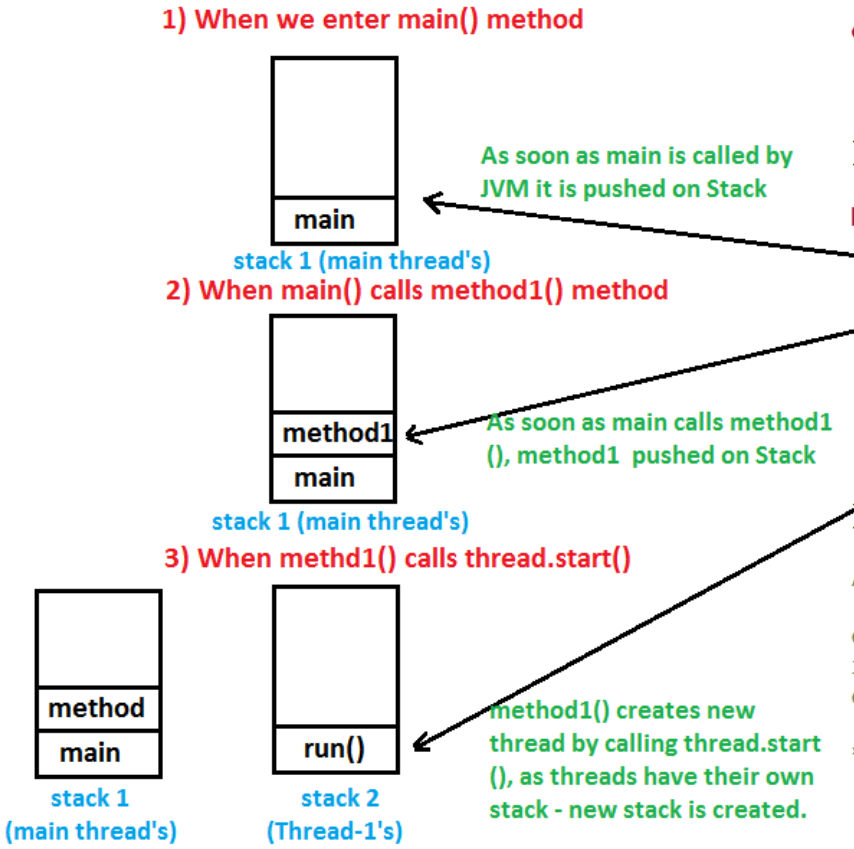
```
MyRunnable runnable=new MyRunnable();

Thread thread=new Thread(runnable);
```

2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e. runnable object :

```
Thread thread=new Thread(runnable);
```

Q: Does Thread implements their own Stack, if yes how?

A: Yes, Threads have their own stack. This is very interesting question, where interviewer tends to check your basic knowledge about how threads internally maintains their own stacks.Shown in figure below.




```
class MyThread extends Thread{
    public void run(){
        System.out.println("in run() method");
    }
}

public class MyClass {
    public static void main(String...args){
        System.out.println("In main() method");
        method1();
    }

    static void method1(){
        MyThread obj=new MyThread();
        obj.start();
    }
}

/*OUTPUT

currentThreadName= main
in run() method
currentThreadName= Thread-1

*/
```

Q: What is multithreading?

A: Multithreading is a process of executing multiple threads simultaneously. Multithreading is used to obtain the multitasking. It consumes less memory and gives the fast and efficient performance. Its main advantages are:

- Threads share the same address space.
- The thread is lightweight.
- The cost of communication between the processes is low.

Q: What do you understand by inter-thread communication?

A:
- The process of communication between synchronized threads is termed as inter-thread communication.
- Inter-thread communication is used to avoid thread polling in Java.
- The thread is paused running in its critical section, and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It can be obtained by wait(), notify(), and notifyAll() methods.

Q: How can you say Thread behaviour is unpredictable?

A: Thread behaviour is unpredictable because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform. To achieve we are going to create 2 threads on same Runnable Object, create for loop in run() method and start both threads. There is no surety that which threads will complete first, both threads will enter anonymously in for loop.

Q: How can you ensure all threads that started from main must end in order in which they started and also main should end in last?

A: We can use join() methodto ensure all threads that started from main must end in order in which they started and also main should end in last.In other words waits for this thread to die. Calling join() method internally calls join(0).

Q: What is difference between starting thread with run() and start() method?

A: When you call start() method, main thread internally calls run() method to start newly created Thread, so run() method is ultimately called by newly created thread.
When you call run() method main thread rather than starting run() method with newly thread it start run() method by itself.

Q: What is significance of using Volatile keyword?

A: Java allows threads to access shared variables. As a rule, to ensure that shared variables are consistently updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that enforces mutual exclusion for those shared variables.
If a field is declared volatile, in that case the Java memory model ensures that all threads see a consistent value for the variable.
Note:volatile is only a keyword, can be used only with variables.

Q: What is race condition in multithreading and how can we solve it?

A: When more than one thread try to access same resource without synchronization causes race condition. So we can solve race condition by using either synchronized block or synchronized method. When no two threads can access same resource at a time phenomenon is also called as mutual exclusion.

Q: When should we interrupt a thread?

A: We should interrupt a thread when we want to break out the sleep or wait state of a thread. We can interrupt a thread by calling the interrupt() throwing the InterruptedException.

Q: What is the purpose of the Synchronized block?

A: The Synchronized block can be used to perform synchronization on any specific resource of the method. Only one thread at a time can execute on a particular resource, and all other threads which attempt to enter the synchronized block are blocked.

- Synchronized block is used to lock an object for any shared resource.
- The scope of the synchronized block is limited to the block on which, it is applied. Its scope is smaller than a method.

Q: What is the difference between notify() and notifyAll()?

A: The notify() is used to unblock one waiting thread whereas notifyAll() method is used to unblock all the threads in waiting state.

Q: How to detect a deadlock condition? How can it be avoided?

A: We can detect the deadlock condition by running the code on cmd and collecting the Thread Dump, and if any deadlock is present in the code, then a message will appear on cmd. Ways to avoid the deadlock condition in Java:

- **Avoid Nested lock:** Nested lock is the common reason for deadlock as deadlock occurs when we provide locks to various threads so we should give one lock to only one thread at some particular time.
- **Avoid unnecessary locks:** we must avoid the locks which are not required.
- **Using thread join:** Thread join helps to wait for a thread until another thread doesn't finish its execution so we can avoid deadlock by maximum use of join method.

Q: Difference between wait() and sleep().

A: **called from synchronized block** :wait() method is always called from synchronized block i.e. wait() method needs to lock object monitor before object on which it is called. But sleep() method can be called from outside synchronized block i.e. sleep() method doesn't need any object monitor.
IllegalMonitorStateException : if wait() method is called without acquiring object lock than IllegalMonitorStateException is thrown at runtime, but sleep() methodnever throws such exception.
Belongs to which class: wait() method belongs to java.lang.Object class but sleep() method belongs to java.lang.Thread class.
Called on object or thread: wait() method is called on objects but sleep() method is called on Threads not objects.
Thread state: when wait() method is called on object, thread that holded object's monitor goes from running to waiting state and can return to runnable state only when notify() or notifyAll()method is called on that object. And later thread scheduler schedules that thread to go from from runnable to running state. when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.
When called from synchronized block:when wait() method is called thread leaves the object lock. But sleep()method when called from synchronized block or method thread doesn't leaves object lock.

Q: Similarity between yield() and sleep().

A:

- yield() and sleep() method belongs to java.lang.Thread class.
- yield() and sleep() method can be called from outside synchronized block.
- yield() and sleep() method are called on Threads not objects.

Q: What are daemon threads?

A: Daemon threads are low priority threads which runs intermittently in background for doing garbage collection.
Few salient features of daemon() threads>

- Thread scheduler schedules these threads only when CPU is idle.
- Daemon threads are service oriented threads, they serves all other threads.
- These threads are created before user threads are created and die after all other user threads dies.
- Priority of daemon threads is always 1 (i.e. MIN_PRIORITY).
- User created threads are non daemon threads.
- JVM can exit when only daemon threads exist in system.
- we can use isDaemon() method to check whether thread is daemon thread or not.
- we can use setDaemon(boolean on) method to make any user method a daemon thread.
- If setDaemon(boolean on) is called on thread after calling start() method than IllegalThreadStateException is thrown.

Q: Can a constructor be synchronized?

A: No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.

- Enclosing constructor in synchronized block will generate compilation error.
- Using synchronized in constructor definition will also show compilation error.

COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted
we can use synchronized block inside constructor.