Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Multiprocessing in java is a executing multiple process simultaneously

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

Multithreading enables to write very efficient program that make maximum use of CPU.

Why Thread/Advantage of Multithreading

- To enhance parallel processing
- To increase response to the user
- To utilize the idle time of the CPU
- To increase the speed of program because of program divided into thread which executed concurrently.
- Prioritize your work depending on priority
- It doesn't block the user because threads are independent and you can perform multiple operations at same time.
- You can perform many operations together so it saves time.
- Threads are independent so it doesn't affect other threads if exception occurs in a single thread.
- Thread is lightweight.
- Threads share the same address space.
- Cost of communication between the thread is low.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU.

Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

A process is a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

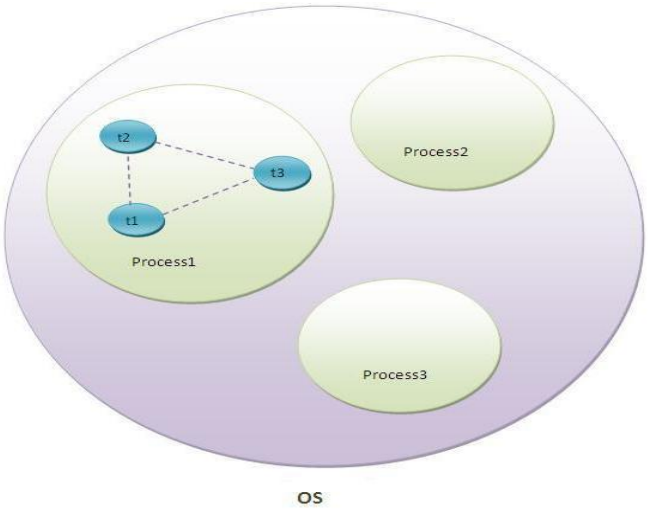
In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads

- Thread is lightweight.
- Threads share the same address space.
- Cost of communication between the thread is low.

What is Thread in java?

A single application/program is divided into small part, each part is called as thread. Thread is single sequential flow of control within a program.

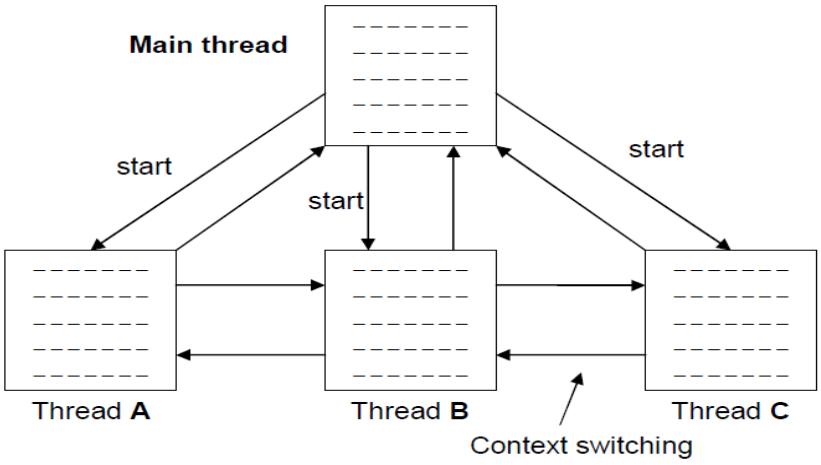
A thread is a lightweight process, a smallest unit of processing. It is a separate path of execution. Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Multithreaded program in Java

When a program contains multiple flows of control it is known as multithreaded program. **Figure illustrates** a Java program containing four different threads, one main and three others



Multithreaded program in Java

In a Java program a main thread is actually a main thread module which is the creating and starting point for all other threads A, B and C. Once initiated by main thread they can run concurrently and share the resources jointly.

Threads running in parallel do not actually mean that they are running at the same time. Since, all threads are running on the single processor, the flow of execution is shared among the threads. The Java interpreter handles the context switching of control among threads insuch a way that they runs concurrently.

There are two type of System:

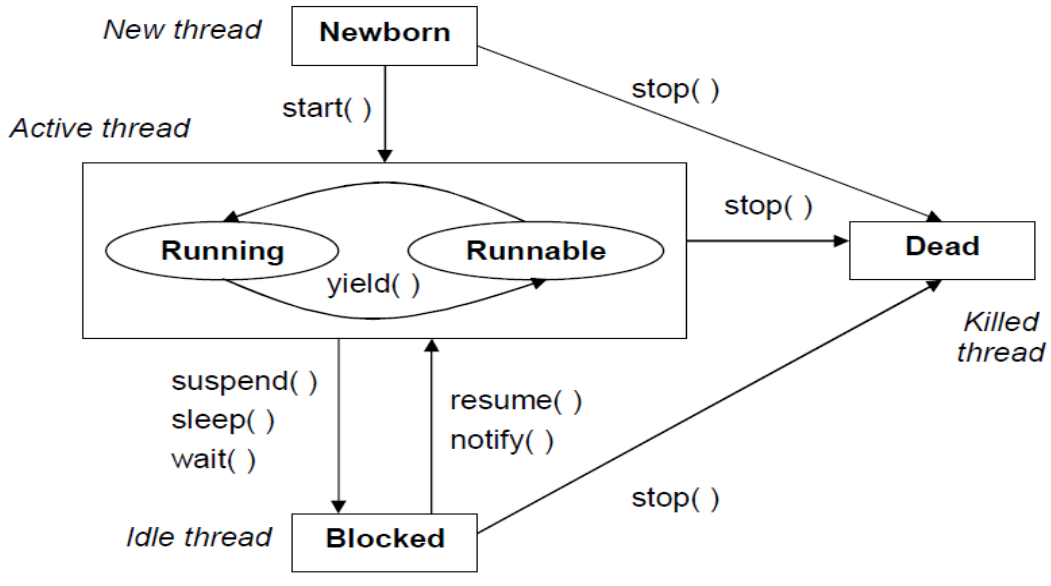
1)Single-threaded systems use an approach called an event loop with polling. In this system, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

2)Multithreded System- Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Life cycle of a Thread

A thread can be in one of the five states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New born state 2. Runnable state 3. Running state 4. Blocked state
5. Dead state



Thread life-cycle

Newborn state

After creation of the thread object, the thread is born which is called as newborn thread. This thread is not scheduled for running. We can either start this state to make it runnable using start( ) method or kill the thread using stop( ) method. Only these two operations can be performed on this state of the thread. If we attempt to perform any other operation, the exception will be thrown.

Runnable state

When the thread is ready for execution and is waiting for availability of the processor, it is said to be in runnable state. The thread has joined the queue of threads that are waiting for execution

If all the threads have equal priority, then they are given time slots for execution in round robin fashion i.e. on first come first serve basis. The thread that relinquishes control joins the queue at the end and again waits for its execution. This process of assigning time to threads is known as time slicing. If we want a thread to relinquish control to another thread of equal priority, a yield ( ) method can be used.

Running state

When the processor has given time to the thread for its execution then the thread is said to be in running state. The thread runs until it relinquishes control to its own or it is pre-empted by a higher priority thread. A running thread may relinquish its control in any one of the following situations:

1. The thread has been suspended using suspend( ) method. This can be revived by using resume( ) method. This is useful when we want to suspend a thread for some time rather than killing it.
2. We can put a thread to sleep using sleep (time) method where 'time' is the time value given in milliseconds. Means, the thread is out of queue during this time period.

3. A thread can wait until some event occurs using wait( ) method. This thread can be scheduled to run again using notify( ) method.

Blocked state

When a thread is prevented from entering into the runnable state and subsequently in running state it is said to be blocked. This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Dead state

A running thread ends its life when it has completed its execution of run( ) method. It is a natural death. However we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon as it is born or while it is running or even when it is in blocked state.

The main thread

All the Java programs are at least single threaded programs. When a Java program starts up, one thread begins running immediately. This is usually called the main thread of the program, because it is the one that is executed when our program begins. **The main thread is important for two reasons:**

a. It is the thread from which other "child" threads can be initiated
b. Generally it must be the last thread to finish execution because it performs various shutdown actions.

Though the main thread is created automatically when the program is started, it can be controlled through a Thread object. For this, we must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread class. Its general form is as shown below:

static Thread currentThread( )

This method returns a reference to the thread in which it is called. Once we obtained a reference to the main thread, we can control it just like any other thread.

```
// Using the main Thread.
class CurrentThread
{
    public static void main(String args[]) throws InterruptedException
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // changing the name of the thread
        t.setName("Prime Thread");
        System.out.println("After name change: " + t);
        Thread.sleep(3000);
        System.out.println("End of main thread & Program");
}
}
```

Using and controlling the main thread

Observe the program, a reference to the current thread (the main thread, here) is obtained by calling currentThread( ), and this reference is stored in the local variable 't'. Next, the program displays information about the thread directly by printing the value of object. The program then calls setName( ) to change the internal name of the thread. Information about the thread is then redisplayed. Next, the current thread is made to sleep for 3000 milliseconds. The sleep( ) method in Thread might throw an InterruptedException. As it is an unchecked

Exception, it can be thrown using the throws clause. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[Prime Thread,5,main]
End of main thread & Program
```

Observe the output of program when 't' is used as an argument to println( ). This display, in order: the name of the thread, its priority, and the name of the thread group to which it belongs. By default, the name of the main thread is main. Its priority is 5, which is also the default value, and main is also the name of the group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment. After the name of the thread is changed, t is again displayed. This time, the new name of the thread i.e. "Prime Thread" is displayed.

The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is:

```
static void sleep(long milliseconds)
throws InterruptedException
```

The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException. The sleep( ) method has another form, which allows us to specify the period in terms of milliseconds as well as nanoseconds:

we can set the name of a thread by using setName( ). We can obtain the name of a thread by calling getName( ). These methods are members of the Thread class and are declared as:

final void setName(String threadName)
final String getName( )

Here, threadName specifies the name of the thread.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class: Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).

set the name of a thread by using **setName( )**.

we can obtain the name of a thread by calling **getName( )**
These methods are members of the **Thread** class and are declared like this:
final void setName(String *threadName*)

final String getName( )

Here, *threadName* specifies the name of the thread.

Creating thread By extending Thread class

For creating a thread a class have to extend the ***Thread*** Class. For creating a thread by this procedure you have to follow these steps:

1. Extend the ***java.lang.Thread*** Class.
   By extend the ***Thread*** class of ***java.lang*** package. **Syntax.....**
   class MyThread extends Thread
   {
   - - - - - - - - - -
   }
2. Override the ***run( )*** method in the subclass to define the code execute by the thread.
   public void run()
   {
   }
3. Create an instance/object of this subclass. This subclass may call a Thread class constructor by subclass constructor.
   Eg. MyThread t=new MyThread();;
4. Invoke the ***start( )*** method on the instance of the class to make the thread eligible for running.
   Eg. t.start();

Example :

```
class MyThread extends Thread
{
        public void run()
        {
          for(int i=1;i<=10;i++)
           {
             System.out.println(i);
           }
        }
}
class MyThreadDemo
{
        public static void main( String args[] )
        {
        MyThread mt = new  MyThread();
        mt.start();
        }
}
```

```
C:\Program Files\Java\jdk1.7.0_71\bin>javac MyThreadDemo.java
C:\Program Files\Java\jdk1.7.0_71\bin>java MyThreadDemo
1
2
3
4
5
6
```

```
7
8
9
10
```

Can we Start a thread twice ?

No, a thread cannot be started twice. If you try to do so, ***IllegalThreadStateException*** will be thrown.

```
public static void main( String args[] )
        {
          MyThread mt = new  MyThread();
          mt.start();
            mt.start();     //Throw the Exception
        }
```

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using ***start()*** method, exception is thrown.

Creating a thread using Runnable interface - step by step approach

1. Implement the "Runnable" interface in your own class.
2.Implement the run() method defined in the "Runnable" interface in your own class. Which means you must override the run() method with your code.
3. Instantiate a runnable object of your own class.
4. Instantiate a thread object of the "Thread" class with the runnable object specified by passing as an argument.
5. Call the start() method on the new object.

```
class MyThread implements Runnable
{
        public void run()
        {
          System.out.println("Thread Running");
        }
}
class MyThreadDemo1
{
        public static void main( String args[] )
        {
        MyThread mt = new  MyThread();
        Thread t=new Thread(mt);
        t.start();
        }
}
```

Output :

```
C:\Program Files\Java\jdk1.7.0_71\bin>javac MyThreadDemo1.java
C:\Program Files\Java\jdk1.7.0_71\bin>java MyThreadDemo1
        Thread Running
```

If you are not extending the Thread class,your class object would not be treated as a thread

object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

The isAlive( ) and join( ) methods

The isAlive( ) method of Thread class is used to determine whether the thread has finished its execution or not? The general form of this method is:

final boolean isAlive()

This method returns true if the thread upon which it is called is still running else returns false. The join( ) method is used to wait for the thread to finish. Its general form is:

final void join() throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow us to specify a maximum amount of time that we want to wait for the specified thread to terminate.

Thread Priority

Each thread have a priority, Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. Or at what time the thread is to be executed. when thread assigned a priority which affect the order in which threads are scheduled for execution or running. **Thread priority is used to decide when to switch from one running thread to another.**

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Higher-priority threads get more CPU time than lower-priority threads. the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also pre-empt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will pre-empt the lower-priority thread

Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority. There are 3 constant values as

```
public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY
```

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

To set and get a thread's priority

To set a thread's priority, use the setPriority ( ) method, which is a member of Thread. This is its general form:

final void setPriority(int level)

Here, level specifies the new priority setting for the calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as final variables within Thread.

We can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

final int getPriority( )

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

Test it Now

```
Output:running thread name is:Thread-0
      running thread priority is:10
      running thread name is:Thread-1
      running thread priority is:1
```

```
/* thread priority example in java */

class TestA extends Thread {
   public void run() {
      System.out.println("TestA Thread started");
      for (int i = 1; i <= 4; i++) {
         System.out.println("\t From thread TestA :i=" + i);
      }
      System.out.println("exit from TestA");
   }
}
```

```
class TestB extends Thread {
   public void run() {
      System.out.println("TestB Thread started");
      for (int j = 1; j <= 4; j++) {
         System.out.println("\t From thread TestB :j=" + j);
      }
      System.out.println("exit from TestB");
   }
}
class TestC extends Thread {
   public void run() {
      System.out.println("testC Thread started");
      for (int k = 1; k <= 4; k++) {
         System.out.println("\t From thread TestC :K=" + k);
      }
      System.out.println("exit from TestC");
   }
}
public class TestThread {
   public static void main(String args[]) {
      TestA A = new TestA();
      TestB B = new TestB();
      TestC C = new TestC();
      C.setPriority(Thread.MAX_PRIORITY);
      B.setPriority(Thread.NORM_PRIORITY+1);
      A.setPriority(Thread.MIN_PRIORITY);
      System.out.println(" Begin thread A");
      A.start();
      System.out.println(" Begin thread B");
      B.start();
      System.out.println(" Begin thread C");
      C.start();
      System.out.println(" End of main thread");
   }
}
```

Output:

```
Begin thread A
Begin thread B
TestA Thread started
Begin thread C
From thread TestA :i=1
testC Thread started
From thread TestC :K=1
```

```
End of main thread
From thread TestC :K=2
From thread TestC :K=3
From thread TestC :K=4
exit from TestC
From thread TestA :i=2
From thread TestA :i=3
TestB Thread started
From thread TestB :j=1
From thread TestB :j=2
From thread TestB :j=3
From thread TestB :j=4
exit from TestB
From thread TestA :i=4
exit from TestA
```

Thread synchronization

Java support multithreading, in multithreaded application thread might want to access the data and call methods simultaneously and this may create problem such as violation of data and unpredictable result. These problems are referred as concurrency problem. Synchronization is the technique that can be used to overcome the concurrency problem that may occur when when multiple threads try to access the same resource.

What is Synchronization?

"When more than one thread uses same resource one should ensure that the resource may be used by only one thread at a time this process is called Synchronization".

Java uses the concept of semaphore for synchronization. This is similar to lock whenever a thread is making an attempt to use shared resources it will lock the resource and then after using it , it will release the resource.

- Key to synchronization is the concept of Monitor
- Monitor is an object that is used as a mutually exclusive lock.
- When thread acquires a lock, it is said to have entered in the monitor and thread has to wait.

Syntax to use synchronized block

```
synchronized (object reference expression)
 {
        //code block
  }
```

Example of synchronization in java

```
class Message
```

```
{
   void display( )
{
       System.out.println("HI");
     System.out.println("I");
     System.out.println("AM");
     System.out.println("LEARNING");
     System.out.println("JAVA");
     try{
           Thread.sleep(400);
         }
         catch(Exception e){
         System.out.println(e);
       }

 }//end of the method
}  //end of the class

class MyThread1 implements Runnable
{
 Message m;
Thread t;
        public MyThread1(Message a)
        {
        m=a;
        t=new Thread(this);
        t.start();
        }

        public void run()
        {
        m.display();
        }
  }
class syncdemo
 {
        public static void main(String args[])
        {
                Message obj = new Message();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread1 t2=new MyThread1(obj);
        }
}
```

Output

```
C:\Program Files\Java\jdk1.7.0_71\bin>javac syncdemo.java

C:\Program Files\Java\jdk1.7.0_71\bin>java syncdemo
HI
 HI
I
I
AM
AM
LEARNING
LEARNING
JAVA
JAVA
```

See the output of above example the output is not proper it means one thread which ob1 take the control of CPU & before it complete all operation in the display method the thread is forced to sleep by method sleep() the second thread which is created object ob2 take the control and start printing the message so output will be mixed and unexpected as shown.

So to solve the above problem & to get correct output you must restrict the access to the method display() to only one thread at a time. To do this you simply need to precede display() with the keyword 'synchronized ' like this

```
synchronized void display( )
   {  .....................
      .....................
   }
```

After changes in the program output will be:

```
C:\Program Files\Java\jdk1.7.0_71\bin>javac syncdemo.java
C:\Program Files\Java\jdk1.7.0_71\bin>java syncdemo
HI
I
AM
LEARNING
JAVA
HI
I
AM
LEARNING
JAVA
```

Q1. Write a program to create two threads; one to print numbers in original order and other to reverse order from 1 to 50.(4-marks for Logic, 4-marks for correct Syntax)

```
class original extends Thread
{
        public void run()
        {
        try
        {
          for(int i=1; i<=50;i++)
          {
            System.out.println("\t First Thread="+i);
           Thread.sleep(300);
          }
        }
        catch(Exception e)      {}
        }
}
class reverse extends Thread
{
        public void run()
        {
        try
        {
          for(int i=50; i>0;i--)
          {
            System.out.println("\t Second Thread="+i);
           Thread.sleep(300);
          }
        }
        catch(Exception e){}
        }
}
class orgrev
{
public static void main(String args[])
{
new original().start();
new reverse().start();
 }
}
```

Or

```
class FirstThread extends Thread
```

```
{
public void run()
{
for (int i=1; i<=50; i++)
{
System.out.println("Message from First Thread : " +i);
}
}
}
class SecondThread extends Thread
{
public void run()
{
for (int i=50; i>0; i--)
{
System.out.println( "Message from Second Thread : " +i);
}
}
}
public class ThreadDemo
{
public static void main(String args[])
{
FirstThread firstThread = new FirstThread();
SecondThread secondThread = new SecondThread();
firstThread.start();
secondThread.start();
}
}
```

Q2. Write a program to create two threads; one to print prime numbers and other to non prime number from 1 to 20.(4-marks for Logic, 4-marks for correct Syntax)

```
class prime extends Thread
{
        public void run()
       {
       int i=0,j=0;
       for(i=2;i<11;i++)
       {
          for(j=2;j<i;j++)
          {
            if(i%j==0)
                  break;
          }
       if(i==j)
       System.out.println(i+" is a prime no.");
       try{
       sleep(500);
       }
       catch(Exception e){}
       }
```

```
        }
    }
class nonprime extends Thread
{
        public void run()
        {
        int i=0,j=0;
        for(i=2;i<11;i++)
        {
          for(j=2;j<i;j++)
          {
            if(i%j==0)
                break;
          }
        if(i!=j)
        System.out.println(i+" is not a prime no.");
        try{
        sleep(500);
        }
        catch(Exception e){}
        }
       }
}
class primeThread
{
   public static void main(String args[])
   {
        prime p=new prime();
        nonprime n=new nonprime();
        System.out.println("Start main");
        System.out.println("1 is universal constant");
        p.start();
        n.start();
    }
}
```

---------- OUTPUT ----------

```
C:\Program Files\Java\jdk1.7.0_71\bin>javac primeThread.java

C:\Program Files\Java\jdk1.7.0_71\bin>java primeThread
Start main
1 is universal constant
2 is a prime no.
3 is a prime no.
4 is not a prime no.
5 is a prime no.
6 is not a prime no.
7 is a prime no.
8 is not a prime no.
9 is not a prime no.
10 is not a prime no.
11 is a prime no.
```

12 is not a prime no.
13 is a prime no.
14 is not a prime no.
15 is not a prime no.
16 is not a prime no.
17 is a prime no.
18 is not a prime no.
19 is a prime no.

Q3. Write a program to create two threads; one to print Odd numbers and other to Even number from 1 to 20.(4-marks for Logic, 4-marks for correct Syntax)

```
class even extends Thread
{
        public void run()
        {
          for(int i=1; i<=20;i++)
          {
             if(i%2==0)
                  System.out.println("\t Even number="+i);
          }

        }
}
class odd extends Thread
{
        public void run()
        {
          for(int i=1; i<=20;i++)
          {
             if(i%2!=0)
                  System.out.println("\t odd number="+i);
          }
        }
}
class oddeven
{
      public static void main(String args[])
        {
         even a=new even();
        odd b=new odd();
        a.start();
        b.start();

        }
}
```

---------- OUTPUT ----------
C:\Program Files\Java\jdk1.7.0_71\bin>javac oddeven.java

```
C:\Program Files\Java\jdk1.7.0_71\bin>java oddeven
        odd number=1
        odd number=3
        Even number=2
        Even number=4
        odd number=5
        Even number=6
        odd number=7
        odd number=9
        odd number=11
        odd number=13
        Even number=8
        Even number=10
        odd number=15
        Even number=12
        Even number=14
        Even number=16
        odd number=17
        Even number=18
        odd number=19
        Even number=20
```

Q4. Write a program to create two threads; one to print given array in ascending order and other to print descending order .(4-marks for Logic, 4-marks for correct Syntax)

```
class Ass extends Thread
{
        public void run()
        {
                int a[]={20,30,55,88,99};

          for(int i=0; i<a.length;i++)
          {
          for(int j=0; j<a.length;j++)
          {
                if(a[i]<a[j])
                {
                  int temp=a[i];
                      a[i]=a[j];
                      a[j]=temp;
                }
        }
        }
        System.out.println("Ascending order:");
        for(int i=0; i<a.length;i++)
          {
              System.out.print("\t"+a[i]);
          }
        System.out.println("");
        }
}
```

```
class Des extends Thread
{
        public void run()
        {
                int a[]={20,30,55,88,99};

          for(int i=0; i<a.length;i++)
          {
          for(int j=0; j<a.length;j++)
          {
                if(a[i]>a[j])
                {
                  int temp=a[i];
                     a[i]=a[j];
                     a[j]=temp;
                }
        }
        }
          System.out.println("Descending order:");
        for(int i=0; i<a.length;i++)
          {
             System.out.print("\t"+a[i]);
           }
          }
}
class Assdes
{
public static void main(String args[])
{
        Ass a1=new Ass();
         a1.start();

        Des a2=new Des();
         a2.start();

 }
}
```

```
C:\Program Files\Java\jdk1.7.0_71\bin>javac Assdes.java

C:\Program Files\Java\jdk1.7.0_71\bin>java Assdes
Ascending order:
        20      30      55      88      99
Descending order:
        99      88      55      30      20
```

Thread Exceptions

- **InterruptedException** : Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted.

```
if (Thread.interrupted())  // Clears interrupted status!
      throw new InterruptedException();
```

- **ThreadDeath** :
  An instance of ThreadDeath is thrown when the Thread.stop() method is invoked.

- **IllegalArgumentException :** Thrown to indicate that a method has been passed an illegal or inappropriate argument.