Java means DURGA SOFT..

CORE JAVA

Material

Java™

India's No.1 Software Training Institute

DURGASOFT

www.durgasoft.com Ph: 9246212143 ,8096969696

Multi Threading

Information about multithreading:-

1) The earlier days the computer's memory is occupied only one program after completion of one program it is possible to execute another program is called uni programming.
2) Whenever one program execution is completed then only second program execution will be started such type of execution is called co operative execution, this execution we are having lot of disadvantages.
    a. Most of the times memory will be wasted.
    b. CPU utilization will be reduced because only program allow executing at a time.
    c. The program queue is developed on the basis co operative execution

To overcome above problem a new programming style will be introduced is called multiprogramming.

1) Multiprogramming means executing the more than one program at a time.
2) All these programs are controlled by the CPU scheduler.
3) CPU scheduler will allocate a particular time period for each and every program.
4) Executing several programs simultaneously is called multiprogramming.
5) In multiprogramming a program can be entered in different states.
    a. Ready state.
    b. Running state.
    c. Waiting state.
6) Multiprogramming mainly focuses on the number of programs.

Advantages of multiprogramming:-

1. The main advantage of multithreading is to provide simultaneous execution of two or more parts of a application to improve the CPU utilization.
2. CPU utilization will be increased.
3. Execution speed will be increased and response time will be decreased.
4. CPU resources are not wasted.

Thread:-

1) Thread is nothing but separate path of sequential execution.
2) The independent execution technical name is called thread.
3) Whenever different parts of the program executed simultaneously that each and every part is called thread.
4) The thread is light weight process because whenever we are creating thread it is not occupying the separate memory it uses the same memory. Whenever the memory is shared means it is not consuming more memory.
5) Executing more than one thread a time is called multithreading.

Information about main Thread;-

When a java program started one Thread is running immediately that thread is called main thread of your program.

1. It is used to create a new Thread(child Thread).
2. It must be the last thread to finish the execution because it perform various actions.
   It is possible to get the current thread reference by using currentThread() method it is a static public method present in Thread class.

```
class CurrentThreadDemo
{   public static void main(String[] arhgs)
    {       Thread t=Thread.currentThread();
            System.out.println("current Thread--->"+t);
            //change the name of the thread
            t.setName("ratan");
            System.out.println("after name changed---> "+t);
    }
};
```

Single threaded model:-

```
class Test
{                                                   begins
        public static void main(String[] args)
        {
                System.out.println("Hello World!");
                System.out.println("hi rattaiah");       body
                System.out.println("hello Sravyasoft");
        }
}                                                   end
```

In the above program only one thread is available is called main thread to know the name of the thread we have to execute the fallowing code.

The main important application areas of the multithreading are

1. Developing video games
2. Implementing multimedia graphics.
3. Developing animations

A thread can be created in two ways:-

1) By extending Thread class.
2) By implementing **java.lang.Runnable** interface

First approach to create thread extending Thread class:-

Step 1:- **Our normal java class will become Thread class whenever we are extending predefined Thread class.**

```
class MyThread extends Thread
{
};
```

Step 2:- override the run() method to write the business logic of the Thread(run() method present in Thread class).

```
class MyThread extends Thread
{       public void run()
        {       System.out.println("business logic of the thread");
                System.out.println("body of the thread");
        }
}
```

Step 2:- Create userdefined Thread class object.

```
MyThread t=new MyThread();
```

Step 3:- ***Start the Thread by using start() method of Thread class.***

```
t.start();
```

Example :-

```
class MyThread extends Thread//defining a Thread
{       //business logic of user defined  Thread
        public void run()
        {       for (int i=0;i<10;i++)
                {       System.out.println("userdefined Thread");
                }
        }
};
class ThreadDemo
{       public static void main(String[] args)      //main thread started
        {       MyThread t=new MyThread();   //MyThread is created
                t.start();          //MyThread execution started
                //business logic of main Thread
                for (int i=0;i<10;i++)
                {       System.out.println("Main Thread");
                }
        }
};
```

Flow of execution:-

1) Whenever we are calling t.start() method then JVM will search start() method in the MyThread class since not available so JVM will execute parent class**(*Thread)*** start() method.

   Thread class start() method responsibilities

   a. User defined thread is registered into Thread Scheduler then only decide new Thread is created.
   b. The Thread class start() automatically calls run() to execute logics of userdefined Thread.

Thread Scheduler:-

- ✓ Thread scheduler is a part of the JVM. It decides thread execution.
- ✓ Thread scheduler is a mental patient we are unable to predict exact behavior of Thread Scheduler it is JVM vendor dependent.
- ✓ Thread Scheduler mainly uses two algorithms to decide Thread execution.
  1) Preemptive algorithm.
  2) Time slicing algorithm.
- ✓ We can't expect exact behavior of the thread scheduler it is JVM vendor dependent. So we can'tsay expect output of the multithreaded examples we can say the possible outputs.

Preemptive scheduling:-

In this highest priority task is executed first after this task enters into waiting state or dead state then only another higher priority task come to existence.

Time Slicing Scheduling:-

A task is executed predefined slice of time and then return pool of ready tasks. The scheduler determines which task is executed based on the priority and other factors.

Example :-is it possible to start a thread twice : no

```
class MyThread extends Thread
```

```
{       public static void main(String[] args)//main thread started
        {       MyThread t=new MyThread();    //MyThread is created
                t.start();
                t.start();
        }
};
D:\DP>java MyThread
Exception in thread "main" java.lang.IllegalThreadStateException
```

Life cycle stages are:-

1) **New**
2) **Ready**
3) **Running state**
4) **Blocked / waiting / non-running mode**
5) **Dead state**

New :- MyThread t=new MyThread();

Ready :- t.start()

Running state:- If thread scheduler allocates CPU for particular thread. Thread goes to running state The Thread is running state means the run() is executed.

Blocked State:-

If the running thread got interrupted of goes to sleeping state at that moment it goes to the blocked state.

Dead State:-If the business logic of the project is completed means run() over thread goes dead state.

Second approach to create thread implementing Runnable interface:-

Step 1:-our normal java class will become Thread class whenever we are implementing Runnable interface.

```
class MyClass extends Runnable
{
};
```

Step2: override run method to write logic of Thread.

```
class MyClass extends Runnable
{       public void run()
        {       System.out.println("Rattaiah from SravyaInfotech");
                System.out.println("body of the thread");
        }
}
```

Step 3:- **Creating a object.**

MyClass obj=new MyClass();

Step 4:- **Creates a Thread class object.**

After new Thread is created it is not started running until we are calling start() method. So whenever we are calling start method that start() method call run() method then the new Thread execution started.

```
Thread  t=new Thread(obj);
t.start();
```

creation of Thread implementing Runnable interface :-

```
class MyThread implements Runnable
{       public void run()
        {       //business logic of user defined Thread
                for (int i=0;i<10;i++)
                {       System.out.println("userdefined Thread");
                }
        }
};
class ThreadDemo
{       public static void main(String[] args)      //main thread started
        {       MyThread r=new MyThread();   //MyThread is created
                Thread t=new Thread(r);
                t.start();          //MyThread execution started
                //business logic of main Thread
                for (int i=0;i<10;i++)
                {       System.out.println("Main Thread");
                }
        }
};
```


First approach:-

important point is that when extending the Thread class, the sub class cannot extend any other base classes because Java allowsonlysingle inheritance.

Second approach:-

1) Implementing the Runnable interface does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread.
2) By implementing the Runnable interface, the class can still extend other base classes if necessary.

Creating two threads by extending Thread class using anonymous inner classes:-

```
class ThreadDemo
{       public static void main(String[] args)
        {       Thread t1 = new Thread()        //anonymous inner class
```

```
                {       public void run()
                        {System.out.println("user Thread-1");
                        }
                };
                Thread t2 = new Thread()        //anonymous inner class
                {       public void run()
                        {System.out.println("user thread-2");
                        }
                };
                t1.start();
                t2.start();
        }
};
```

Creating two threads by implementing Runnable interface using anonymous inner classes:-

```
class ThreadDemo
{       public static void main(String[] args)
        {
                Runnable r1 = new Runnable()
                {       public void run()
                        {System.out.println("user Thread-1");
                        }
                };
                Runnable r2 = new Runnable()
                {       public void run()
                        {System.out.println("user thread-2");
                        }
                };

                Thread t1 = new Thread(r1);
                Thread t2 = new Thread(r2);
                t1.start();
                t2.start();
        }
};
```

Different ways to start the Thread:-

```
class MyThread extends Thread
{       public void run()
        {System.out.println("user thread is running extends Thread");
        }
};
class MyRunnable implements Runnable
{       public void run()
        {System.out.println("user thread is Running implements Runnable");
        }
};
class ThreadDemo
```

```
{       public static void main(String[] args)
        {       //creating Thread class object by passing anonymous classes
                new Thread(new MyThread()).start();
                new Thread(new MyRunnable()).start();
        }
};
```

Internal Implementation of multiThreading:-

```
interface Runnable
{       public abstract void run();
}
class Thread implements Runnable
{       public void run()
        {       //empty implementation
        }
};
class MyThread extends Thread
{       public void run()       //overriding run() to write business logic
        {       for (int i=0;i<5 ;i++ )
                {       System.out.println("user implementation");
                }
        }
};
```

Difference between t.start() and t.run():-

- In the case of t.start(), Thread class start() is executed a new thread will be created that is responsible for the execution ofrun() method.
- But in the case of t.run() method, no new thread will be created and the run() is executed like anormal method call by the main thread.

Note :- Here we are not overriding the run() method so thread class run method is executed which is having empty implementation so we are not getting any output.

```
class MyThread extends Thread
{       }
```

```
class  ThreadDemo
{       public static void main(String[] args)
        {       MyThread t=new MyThread();
                t.start();
                for (int i=0;i<5;i++ )
                {       System.out.println("main thread");
                }
        }
}
```

Note :- If we are overriding start() method then JVM is executes override start() method at this situation we are not giving chance to the thread class start() hence n new thread will be created only one thread is available the name of that thread is main thread.

```
class MyThread extends Thread
{       Public void start()
        {       System.out.println("override start method");
        }
}
class  ThreadDemo
{       public static void main(String[] args)
        {       MyThread t=new MyThread();
                t.start();
                for (int i=0;i<5 ;i++ )
                {       System.out.println("main thread");
                }
        }
}
```

Different Threads are performing different tasks:--

1) Particular task is performed by the number of threads here number of threads(t1,t2,t3) are executing same method (functionality).
2) In the above scenario for each and every thread one stack is created. Each and every method called by particular Thread the every entry stored in the particular thread stack.

```
class MyThread1 extends Thread
{       public void run()
        {       System.out.println("ratan task");
        }
};
class MyThread2 extends Thread
{       public void run()
        {       System.out.println("Sravya task");
        }
};
class MyThread3 extends Thread
{       public void run()
        {       System.out.println("anu task");
        }
};
```

```
class ThreadDemo
{       public static void main(String[] args) //1- main Thread
        {       MyThread1 t1 = new MyThread1();
                MyThread2 t2 = new MyThread2();
                MyThread3 t3 = new MyThread3();
                t1.start(); //2
                t2.start(); //3
                t3.start();         //4
        }
};
```

Here Four Stacks are created

Main ----------stack1
t1----------stack2
t2----------stack3
t3----------stack4

Multiple threads are performing single task:-

```
class MyThread extends Thread
{       public void run()
        {       System.out.println("Sravyasoft task");
        }
}
class  ThreadDemo
{       public static void main(String[] args)//main Thread is started
        {       MyThread t1=new MyThread();         //new Thread created
                MyThread t2=new MyThread();         //new Thread created
                MyThread t3=new MyThread();         //new Thread created
                t1.start();         //Thread started
                t2.start();         //Thread started
                t3.start();         //Thread started
        }
}
```

Getting and setting names of Thread:-

1) Every Thread in java having name
    a. default name of the main thread is main
    b. default name of user created threads starts from ***Thread-0***.
        t1 --→Thread-0
        t2 --→Thread-1
        t3 --→Thread-2
2) To set the name use setName() & to get the name use getName(),
    Public final String getName()
    Public final void setName(String name)

Example:-

```
class MyThread extends Thread
{       public void run()
        {       System.out.println("thread is running");
        }
```

```
}
class ThreadDemo
{       public static void main(String args[])
        {       MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                System.out.println("t1 Thread name="+t1.getName());
                System.out.println("t2 Thread name="+t2.getName());
                System.out.println(Thread.currentThread().getName());
                t1.setName("ratan");
                System.out.println("after changeing t1 Thread name="+t1.getName());
        }
}
```


Thread Priorities:-

1. Every Thread in java has some property. It may be default priority provided be the JVM orcustomized priority provided by the programmer.
2. The valid range of thread priorities is 1 – 10. Where one is lowest priority and 10 is highest priority.
3. The default priority of main thread is 5. The priority of child thread is inherited from the parent.
4. Thread defines the following constants to represent some standard priorities.
5. Thread Scheduler will use priorities while allocating processor the thread which is havinghighest priority will get chance first and the thread which is having low priority.
6. If two threads having the same priority then we can't expect exact execution order it dependsupon Thread Scheduler.
7. The thread which is having low priority has to wait until completion of high priority threads.
8. Three constant values for the thread priority.
    a. **MIN_PRIORITY = 1**
    b. **NORM_PRIORITY = 5**
    c. **MAX_PRIORITY = 10**

Thread class defines the following methods to get and set priority of a Thread.

Public final int getPriority()
Public final void setPriority(int priority)

Here 'priority'indicates a number which is in the allowed range of 1 – 10. Otherwise we will get

Runtime exception saying “IllegalArgumentException”.

Thread priority decide when to switch from one running thread to another this process is called context switching.

```
class MyThread extends Thread
{       public void run()
        {       System.out.println("current Thread name = "+Thread.currentThread().getName());
                System.out.println("current Thread priority = "+Thread.currentThread().getPriority());
        }
};
class ThreadDemo
{       public static void main(String[] args)//main thread started
        {       MyThread t1 = new MyThread();
                MyThread t2 = new MyThread();
                t1.setPriority(Thread.MIN_PRIORITY);
                t2.setPriority(Thread.MAX_PRIORITY);
                t1.start();
                t2.start();
        }
};
```


Java.lang.Thread.yield():-

- Yield() method causes to pause current executing Thread for giving the chance for waitingthreads of same priority.
- If there are no waiting threads or all threads are having low priority then the same thread willcontinue its execution once again.

Syntax:-

Public static native void yield();

Ex:

```
class MyThread extends Thread
{       public void run()
        {
                for(int i=0;i<10;i++)
                {       Thread.yield();
                        System.out.println("child thread");
                }
        }
}
class ThreadYieldDemo
{       public static void main(String[] args)
        {       MyThread t1=new MyThread();
                t1.start();
                for(int i=0;i<10;i++)
                {       System.out.println("main thread");
                }
        }
}
```

Java.lang.Thread.join(-,-)method:-

- Join method allows one thread to wait for the completion of another thread.
  - t.join(); ---> here t is a Thread Object whose thread is currently running.
- Join() is used to stop the execution of the thread until completion of some other Thread.
-

if a t1 thread is executed t2.join() at that situation t1 must wait until completion of the t2 thread.

public final void join()throws InterruptedExcetion
Public final void join(long ms)throws InterruptedException
Public final void join(long ms, int ns)throws InterruptedException

Methods of Thread class:-

```
class MyThread extends Thread
{       public void run()
        {       for (int i=0;i<5;i++)
                {       try{    Thread.sleep(2000);   }
                        catch(InterruptedException e)
                        {e.printStackTrace();
                        }
                        System.out.println(i);
                }
        }
};
class ThreadDemo
{       public static void main(String[] args)
        {       MyThread t1=new MyThread();
                MyThread t2=new MyThread();
```

```
            MyThread t3=new MyThread();
            t1.start();
            try
            {t1.join();       }
            catch (InterruptedException ie)
            {ie.printStackTrace();
            }
            t2.start();
            t3.start();
        }
};
```

Java.lang.Thread.Interrupted():-

- A thread can interrupt another sleeping or waiting thread. But one thread is able to interrupted only another sleeping or waiting thread.
- To interrupt a thread use Thread class interrupt() method.

Public void interrupt()

Effect of interrupt() method call:-

```
class MyThread  extends Thread
{       public void run()
        {       try
                {       for (int i=0;i<10;i++ )
                        {       System.out.println("i am sleeping ");
                                Thread.sleep(5000);
                        }
                }
                catch (InterruptedException ie)
                {       System.out.println("i got interupted by interrupt() call");
                }
        }
};
class ThreadDemo
{       public static void main(String[] args)
        {       MyThread t=new MyThread();
```

```
                t.start();
                t.interrupt();
            }
        };
```

No effect of interrupt() call:-

```
class MyThread  extends Thread
{       public void run()
        {       for (int i=0;i<10;i++ )
                {       System.out.println("i am sleeping ");
                }
        }
};
class ThreadDemo
{       public static void main(String[] args)
        {       MyThread t=new MyThread();
                t.start();
                t.interrupt();
        }
};
```

NOTE:- The interrupt() is effected whenever our thread enters into waiting state or sleeping state and if the our thread doesn't enters into the waiting/sleeping state interrupted call will be wasted.

Shutdown Hook:-

- Shutdown hook used to perform cleanup activities when JVM shutdown normally or abnormally.
- Clean-up activities like
  - Resource release
  - Database closing
  - Sending alert message
- So if you want to execute some code before JVM shutdown use shutdown hook

The JVM will be shutdown in fallowing cases.

a. When you typed ctrl+C
b. When we used System.exit(int)
c. When the system is shutdown ……etc

To add the shutdown hook to JVM use addShutdownHook(obj) method of Runtime Class.

public void addShutdownHook(java.lang.Thread);

To remove the shutdown hook from JVM use removeShutdownHook(obj) method of Runtime Class.

public boolean removeShutdownHook(java.lang.Thread);

To get the Runtime class object use static factory method getRuntime() & this method present in Runtime class

*Runtime r = Runtime.**getRuntime();***

Factory method:- one java class method is able to return same class object or different class object is called factory method.

Example :-

```
class MyThread extends Thread
{       public void run()
        {System.out.println("shoutdown hook");
        }
};
class ThreadDemo
{       public static void main(String[] args)throws InterruptedException
        {       MyThread t = new MyThread();
                //creating Runtime class Object by using factory method
                Runtime r = Runtime.getRuntime();
                r.addShutdownHook(t);//adding Thread to JVM hook
                for (int i=0;i<10 ;i++)
                {System.out.println("main thread is running");
                        Thread.sleep(3000);
                }
        }
};
```

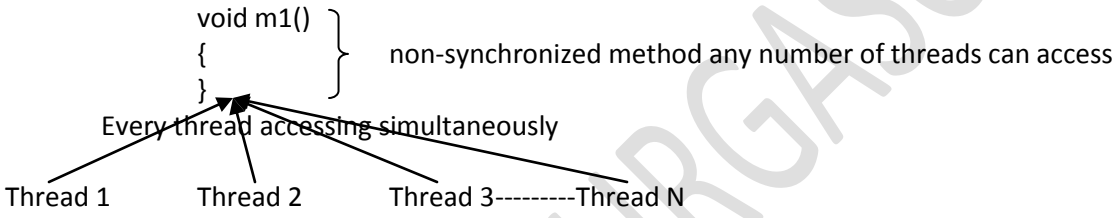
```
D:\DP>java ThreadDemo
main thread is running
main thread is running
main thread is running
shoutdown hook
```

while running Main thread press Ctrl+C then hook thread will be executed.

Synchronized :-

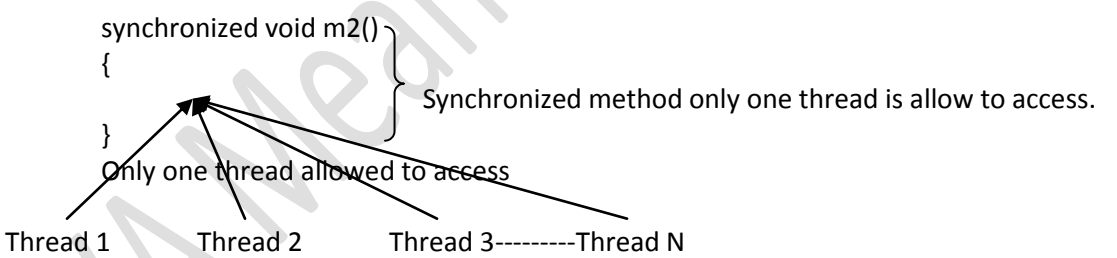
- Synchronized modifier is the modifier applicable for methods but not for classes andvariables.
- If a method or a block declared as synchronized then at a time only one Thread is allowed tooperate on the given object.
- The main advantage of synchronized modifier is we can resolve data inconsistency problems.
- But the main disadvantage of synchronized modifier is it increases the waiting time of the Thread and effects performance of the system.Hence if there is no specific requirement it isnever recommended to use.
- The main purpose of this modifier is to reduce the data inconsistence problems.

Non-synchronized methods



1) In the above case multiple threads are accessing the same methods hence we are getting data inconsistency problems. These methods are not thread safe methods.
2) But in this case multiple threads are executing so the performance of the application will be increased.

Synchronized methods



1) In the above case only one thread is allow to operate on particular method so the data inconsistency problems will be reduced.
2) Only one thread is allowed to access so the performance of the application will be reduced.
3) If we are using above approach there is no multithreading concept.

Hence it is not recommended to use the synchronized modifier in the multithreading programming.

Example :-

```
class Test
{       public static synchronized void x(String msg)      //only one thread  is able to access
        {       try{
                System.out.println(msg);
                Thread.sleep(4000);
```

```
                System.out.println(msg);
                Thread.sleep(4000);
                }
                catch(Exception e)
                {e.printStackTrace();}
        }
}
class MyThread1 extends Thread
{       public void run(){Test.x("ratan");}      };
class MyThread2 extends Thread
{       public void run(){Test.x("anu");}        };
class MyThread3 extends Thread
{       public void run(){Test.x("banu");}       };
class TestDemo
{       public static void main(String[] args)//main thread  -1
        {       MyThread1 t1 = new MyThread1();
                MyThread2 t2 = new MyThread2();
                MyThread3 t3 = new MyThread3();
                t1.start();     //2-Threads
                t2.start();     //3-Threads
                t3.start();     //4-Threads
        }
}
```

If method is synchronized:

```
D:\DP>java ThreadDemo
anu
anu
banu
banu
ratan
ratan
```

if method is non-synchronized:-

```
D:\DP>java ThreadDemo
banu
ratan
anu
banu
anu
ratan
```

synchronized blocks:-

if the application method contains 100 lines but if we want to synchronized only 10 lines of code use synchronized blocks.

The synchronized block contains less scope compare to method.

If we are writing all the method code inside the synchronized blocks it will work same as the synchronized method.

Syntax:-

```
synchronized(object)
{       //code
}
class Heroin
{       public void message(String msg)
        {       synchronized(this){
                System.out.println("hi "+msg+" "+Thread.currentThread().getName());
```

```
                try{Thread.sleep(5000);}
                catch(InterruptedException e){e.printStackTrace();}
                }
                System.out.println("hi Sravyasoft");
        }
};
class MyThread1 extends Thread
{       Heroin h;
        MyThread1(Heroin h)
        {this.h=h;
        }
        public void run()
        {       h.message("Anushka");
        }
};
class MyThread2 extends Thread
{       Heroin h;
        MyThread2(Heroin h)
        {this.h=h;
        }
        public void run()
        {       h.message("Ratan");
        }
};
class ThreadDemo
{
        public static void main(String[] args)
        {       Heroin h = new Heroin();
                MyThread1 t1 = new MyThread1(h);
                MyThread2 t2 = new MyThread2(h);
                t1.start();
                t2.start();
        }
};
```

Daemon threads:-

The threads wchich are executed at background is called daemon threads.

Ex:- garbage collector,ThreadSchedular.default exceptional handler.

Non-daemon threads:-

The threads which are executed fore ground is called non-daemon threads.

Ex:- normal java application.

- When we create a thread in java that is user defined thread and f it is running JVM will not terminate that process.
- If a thread is marked as a daemon thread JVM does not wait to finish and as soon as all the user defined threads are finished then it terminates the program and all associated daemon threads.
- Set the daemon nature to thread by using setDaemon() method
  - MyThread t = new Mythread();
    t.setDaemon(true);
- To know whether a thread is daemon or not use isDaemon() method
  - Thread.currentThread().isDaemon();

```
class MyThread extends Thread
{       void message(String str)
        {       try     {
                        System.out.println("message="+str);
                        Thread.sleep(1000);     }
                catch (InterruptedException e)
                {e.printStackTrace();     }
        }
        public void run()
        {       if (Thread.currentThread().isDaemon())
                {       while (true)
                        {       message("print hi ratan");
                        }
                }
        }
};
class ThreadDemo
{       public static void main(String[] args)
        {       MyThread t = new MyThread();
                t.setDaemon(true);//setting daemon nature to Thread
                t.start();
                try{Thread.sleep(5000);}
                catch(InterruptedException e)
                {e.printStackTrace();}
                System.out.println("main thread completed");
        }
};
```

Note :- in above example make the setdaemon() is comment mode then the program never terminates even main thread finished it's execution.

```
class MyThread extends Thread
{       int total;
```

```
        public void run()
        {       synchronized(this){
                for (int i=0;i<10 ;i++)
                {       total=total+i;
                }
                notify();
         }
        }
}
class ThreadDemo
{       public static void main(String[] args)
        {       MyThread t = new MyThread();
                t.start();
                synchronized(t)
                {       System.out.println("MyThrad total is waiting for MyThread completion...");
                        try{
                        t.wait();}
                        catch(InterruptedException ie){System.out.println(ie);}
                }
                System.out.println("MyThrad total is ="+t.total);
        }
};
```

Volatile:-

- Volatile modifier is also applicable only for variables but not for methods and classes.
- If the values of a variable keep on changing such type of variables we have to declare withvolatile modifier.
- If a variable declared as a volatile then for every Thread a separate local copy will be created.
- Every intermediate modification performed by that Thread will take place in local copy instead of master copy.
- Once the value got finalized just before terminating the Thread the master copy value will be updated with the local stable value. The main advantage of volatile modifier is we can resolve the data inconsistency problem.
- But the main disadvantage is creating and maintaining a separate copy for every Thread
- Increases the complexity of the programming and effects performance of the system.