Multitasking: Executing severa ltasks simultaneously is the concept of multitasking.There are two types of multitasking's.

- Process based multitasking.
- Thread based multitasking.

Process-based Multitasking

Process-based multitasking allows processes (or programs) to run concurrently on a computer. For example , running different applications at the same time like downloading a file and printing a file .

A process has a self-contained execution environment. A process has a complete , private set of basic resources . Each process has its own memory space.

Thread-based Multitasking

Thread-based multitasking allows different parts of the same process (program) to run concurrently .
A good example is printing and formatting text in a word processor at the same time.
Thread-based multitasking is only feasible if the two parts are independent of each other or more precisely they are independent paths of execution at run time.

Threads are also referred as lightweight processes and they also provide an execution environment . Threads exists within a processes and share its resources like memory and open files.

We have several advantages of Thread-based multitasking over Process-based multitasking .

1. Threads shares the same address space.
2. Context switching between the threads is less expensive than in Processes.

In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.

The main important application of multithreading are:

- To implement multimedia graphics.
- To develop animations.
- To develop video games etc.

We can define a Thread in the following 2 ways.

- By extending Thread class.
- By implementing Runnable interface.

Defining a thread by extending "Thread class ":

```
class MyThread extends Thread
{
 public void run()          //run()  method of java.lang.Thread class is overridden here
 {                          //  job of a thread is defined by run() method
  for(int i=0;i<10;i++)
    {
     System.out.println("Child Thread  ");
    }
  }
}

class MyThreadDemo
 {
   public static void main(String[] args)
    {
      MyThread pt = new MyThread ();    // Instantiation of a Thread
      MyThread pt1 = new MyThread ();
      pt.start();
     pt1.start();
      for(int i=0;i<10;i++)     // job of main thread
            {  System.out.println("Main Thread  ");        }
      }
}
```

Note:

1. ThreadScheduler:

• If multiple threads are waiting to execute which Thread will execute first is decided by "Thread Scheduler" which is part of JVM.

• Which algorithm or behavior followed by Thread Scheduler we can't expect exactly it is JVM vendor dependent.

2. importance of Thread class start() method.

• For every thread required mandatory activities like registering the thread with thread scheduler will takes care by Thread class start() method and programmer is responsible just to define the job of the thread inside run() method. That is start() method acts as best assistant to theprogrammer.

start() {

1. Register Thread with Thread Scheduler
2. All other mandatory low level activities.
3. Invoke or calling run() method.

}

•We can conclude that without executing Thread class start() method there is no chance of starting a

new Thread in java.

3. If we are not overriding run() method:

• If we are not overriding run() method then Thread class run() method will be executed which has empty implementation and hence we won't get any output.
• It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

4. Overloading of run() method.

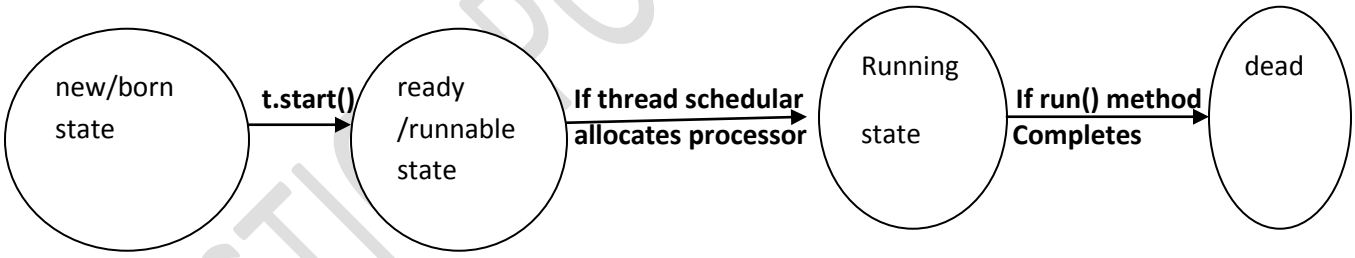
•We can overload run() method but Thread class start() method always invokes no argument run() method the other overload run() methods we have to call explicitly then only it will beexecuted just like normal method.

5: Overriding of start() method:

•If we override start() method then our start() method will be executed just like a normal method call and no new thread will be started.

6. Life cycle of the thread :

MyThread t = new Mythread()



7.
• After starting a thread we are not allowed to restart the same Thread once again otherwise we will get runtime exception saying "IllegalThreadStateException"

Defining a thread by implementing Runnable interface:
We can define a thread even by implementing Runnable interface also. Runnable interface present in java.lang and contains only one method run().

```
class MyThread1 implements Runnable
{
 public void run()          //run()  method of java.lang.Runnable interface is overridden here
```

```
{                              //  job of a thread is defined by run() method
 for(int i=0;i<10;i++)
   {
    System.out.println("Child Thread  ");
   }
 }
}

class MyThreadDemo1
 {
  public static void main(String[] args)
   {
     MyThread1 pt = new MyThread1 ();
    Thread t = new Thread (pt);
     t.start();

     for(int i=0;i<10;i++)     // job of main thread
           {  System.out.println("Main Thread  ");        }
     }
}
```

Best approach to define a thread:

- Among the two ways of defining a Thread, implements Runnable approach is always recommended.
- In the first approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance.
- But in the second approach while implementing Runnable interface we can extend some other class also. Hence implements Runnable mechanism is recommended to define a thread.

Thread class constructors:

- Thread t=new Thread();
- Thread t=new Thread(Runnable r);
- Thread t=new Thread(String name);
- Thread t=new Thread(Runnabler,String name);
- Thread t=new Thread(ThreadGroup g,String name);
- Thread t=new Thread(ThreadGroup g,Runnable r);
- Thread t=new Thread(ThreadGroup g,Runnable r,String name);
- Thread t=newThread(ThreadGroup g,Runnable r,String name,long stackSize);

Some More Examples:

currentThread() method of Thread class returns current thread Object and getName() method returns name of a thread.

```
class PrestigeThread extends Thread
{
 public void run()
 {
  for(int i=0;i<10;i++)
    {  System.out.println("Child Thread  "+ Thread.currentThread().getName());
       }
  }
}
```

Every thread has a name in JAVA. Name can be provided by JVM or we can set and get the name by using setName() and getName() methods

```
class PrestigeThread1  extends Thread
{
 public void run()
 {
  for(int i=0;i<10;i++)
    { System.out.println("Child Thread  1 " + Thread.currentThread().getName());
                                  }
} }

class PrestigeThreadDemo
 {
   public static void main(String[] args)
    {
      PrestigeThread pt = new PrestigeThread();
      PrestigeThread pt1= new PrestigeThread();
     PrestigeThread1 pt2 = new PrestigeThread1();
    pt.start();
    pt2.start();
    pt1.start();

     for(int i=0;i<10;i++)
       { System.out.println("Main Thread  "+ Thread.currentThread().getName() );

      }
          //pt.start();     //R.E. IllegalThreadStateException
     }       }
```

Getting and setting name of a Thread:

•Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.
• Thread class defines the following methods to get and set name of a thread.
public final String getName()
public final void setName(String name)

```
class MyThreadSetName extends Thread
{
public void run()
{
 for(int i=0; i<10; i++)
   System.out.println(Thread.currentThread().getName());
}
}

class TestSetNameThread
{
 public static void main(String[] args)
  {
   MyThreadSetName t = new MyThreadSetName();
   t.start();
   MyThreadSetName t1 = new MyThreadSetName();
   t1.setName("Hello");
    t1.start();
   Thread.currentThread().setName("Hi");
   for(int i=0; i<10; i++)
    System.out.println(Thread.currentThread().getName());
   Thread t2 = new Thread("Hello 1");
   System.out.println(t2.getName());
   }
}
```

Thread Priorities

JVM selects to run a Runnable thread with the highest priority.
All Java threads have a priority in the range 1-10.
Top priority is 10, lowest priority is 1.
Normal priority ie. priority by default is 5.

Thread.MIN_PRIORITY - minimum thread priority
Thread.MAX_PRIORITY - maximum thread priority
Thread.NORM_PRIORITY - maximum thread priority

Whenever a new Java thread is created it has the same priority as the thread which created it.

We can get and set the priority of a Thread by using the following methods.

- public final int getPriority()
- public final void setPriority(int newPriority); //the allowed values are 1 to 10

The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".

Default priority:

•The default priority only for the main Thread is 5. But for all the remaining threads the default priority will be inheriting from parent to child. That is whatever the priority parent has by default the same priority will be for the child also.

Example 1:

```
class MyThread2 extends Thread
{
 public void run() {
System.out.println("Child "+ Thread.currentThread().getPriority());  }
}

class ThreadPriority
 {
   public static void main(String[] args)
    {
      System.out.println(Thread.currentThread().getPriority());
      MyThread2 pr = new MyThread2 ();
     System.out.println(pr.getPriority());
     pr.setPriority(7);
      pr.start();
       System.out.println(pr.getPriority());

         }
}
```

Example 2:

```
 class MyThread3 extends Thread
   { public void run()
       { for(inti=0;i<10;i++)
           { System.out.println("child thread");
           }
   }}

 class ThreadPriorityDemo
{
```

```
public static void main(String[] args) {
  MyThread3 t=new MyThread3();
  //t.setPriority(10);  //------ line 1
   t.start();
   for(inti=0;i<10;i++)
   { System.out.println("mainthread"); }
}}
```

•If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.
• If we are not commenting line 1 then child Thread has the priority 10 and main thread has the priority 5 hence child Thread will get chance for execution and after completing child thread main thread will get the chance.

Note: Some operating systems(like windowsXP) may not provide proper support for thread priorities. We have to install separate bats provided by vendor to provide support for priorities. The Methods to Prevent a thread from execution.

Methods to prevent a thread from Execution:

• We can prevent(stop) a thread execution by using following methods.

```
public static native void yield()
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanos) throws InterruptedException
public final void join() throws InterruptedException
public final void join(long millis)  throws InterruptedException
public final void join (long millis, int nanos) throws InterruptedException
```

yield():
•yield() method causes "to pause current executing thread for giving chance of remaining waiting threads of same priority".
• If all waiting threads have the low priority or if there is no waiting threads then the same thread will be continued its execution.
•If several waiting threads with same priority available then we can't expect exact which thread will get chance for execution.
• thread which is yielded, when it get chance once again for execution is depends on thread scheduler.

```
class MyThread4 extends Thread
{ public void run() {
    for(int i=0;i<5;i++)
        { Thread.yield();
         System.out.println("childthread");
```

```
   }
}}

class ThreadYieldDemo
{
 public static void main(String[]args)
    {
   MyThread4 t=new MyThread4();
    t.start();
    for(int i=0;i<5;i++)
        { System.out.println("main thread"); }
  }}
```

join():

- If a thread wants to wait until completing some other thread then we should go for join()method.
  For Example: If a thread t1 executes t2.join() then t1 should go for waiting state until completing t2.
- Every join() method throws InterruptedException ,which is checked exception hence compulsory we should handle either by try catch or by throws keyword.

```
class MyThread5 extends Thread
 {
  public void run() {
         for(inti=0;i<5;i++)
           {
        System.out.println("See the Thread");
         try { Thread.sleep(2000); } catch (InterruptedExceptione){}
       }
}}

class ThreadJoinDemo
{ public static void main(String[] args) throws InterruptedException
       { MyThread5 t=new MyThread5();
         t.start();
        //t.join(); 1
       for(inti=0;i<5;i++)
        { System.out.println("Prestige Thread"); }
   }}
```

•If we are commenting line 1 then both threads will be executed simultaneously and we can't expect exact execution order.

• If we are not commenting line 1 then main thread will wait until completing child thread.

sleep() :

• If a thread don't want to perform any operation for a particular amount of time then we should go for sleep()method.

```
class ThreadSleepDemo
{ public static void main(String[] args) throws InterruptedException
      { System.out.println("M");
      Thread.sleep(3000);
     System.out.println("E");
    Thread.sleep(3000);
   System.out.println("G");
  Thread.sleep(3000);
 System.out.println("A");
}}
```

Interrupting a thread:

• We can interrupt a sleeping or waiting thread by using interrupt()(breakoff) method of Thread class.

Example:

```
class MyThread6 extends Thread
  { public void run()
    { try { for(inti=0;i<5;i++)
              { System.out.println("I am lazy thread:"+i);
                Thread.sleep(2000);
              }}
      catch(InterruptedException e)
           { System.out.println("I got interrupted");
       }}}

 class ThreadInterruptDemo
   { public static void main(String[] args)
      { MyThread6 t=new MyThread6();
        t.start();
      //t.interrupt(); 1
System.out.println("end of main thread"); }
  }
```

•If we are commenting line 1 then main thread won't interrupt child thread and hence child thread will be continued until its completion.

• If we are not commenting line 1 then main thread interrupts child thread and hence child Thread won't continued until its completion in this case the outputis.