Thread

A thread is a small piece of code specifying a separate path of execution.

There are two types of multitasking:

1. Process-based multitasking :- it is a feature that allows a computer to run 2 or more programs simultaneously.

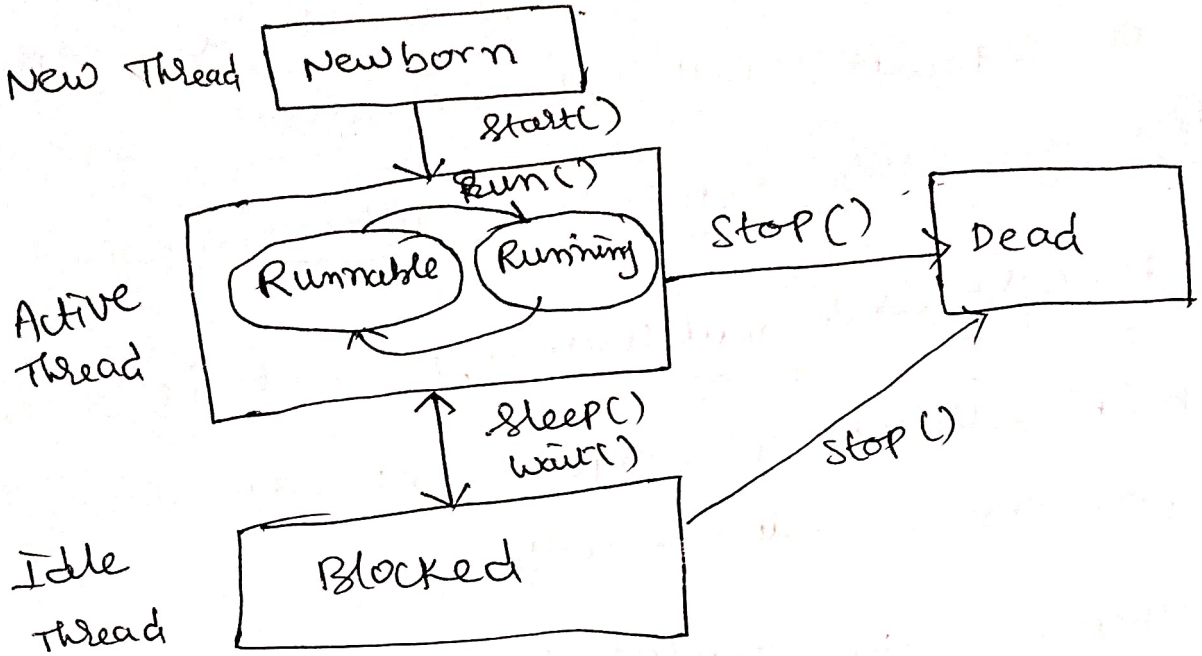
2. Thread-based multitasking :- In a thread-based multitasking environment, the single program is divided into two or more tasks. Each task is assigned to a thread.

Difference between thread-based and process-based multitasking

Thread-based multitasking	Process-based multitasking
• Execution of a multiple parts of a single program simultaneously.	• Execution of multiple programs simultaneously.
• These are light-weight	• These are heavy-weight
• Less overhead - switching between threads is faster	• More over-head Context switching between processes is slower.

Running State :- Running

Life cycle of a Thread



New born state :- When we create a thread object, the thread is born and is said to be in new born state. The thread is not yet scheduled for running. Schedule it for running using start() method. Once a start() method is called thread moves to the runnable state.

Runnable state :- The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution.

Running state :- Running means that the Processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

Blocked state :- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.

Dead state :- A running thread ends its life when it has completed execution. It is a natural death. However, it can kill by sending the stop() message to it at any state thus causing a premature death.

How to create a thread

Threads can be create by using 2 methods

1. By extending a thread class,
2. By using Runnable interface,

How to create a thread by extending a thread class

```
class A extends Thread
{
    Public void run()
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println("Natural number Thread " + i);
            try
            {
                Thread.sleep(500);
            }
            catch(InterruptedException e)
            {
                system.out.println("Thread interrupted");
            }
        }
    }
}
class B extends Thread
{
    Public void run()
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println("Odd number Thread " + (i*2-1));
```

```
        try
          {
            Thread.sleep(500);
          }
        Catch(InterruptedException e)
          {
            System.out.println("odd Thread
                                Interrupted");
          }
    }
  }
}

class C extends Thread
  {
    Public void run()
      {
        for(int i=1; i<=10; i++)
          {
            System.out.println("even thread " + (i*2));
            try
              {
                Thread.sleep(500);
              }
            Catch(InterruptedException e)
              {
                System.out.println("even thread
                                    interrupted");
              }
          }
      }
  }
```

```
class Threaddemo
{
    public static void main(String args[])
    {
        A t1 = new A();
        B t2 = new B();
        C t3 = new C();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Create a thread using Runnable interface

```
class A implements Runnable
{
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println("ThreadA" + i);
        }
    }
}
```

```
class RunnableDemo
{
    Public static void main(String args[])
    {
        A t1= new A();
        Thread t2 = new Thread (t1);
        t2. start();
    }
}
```

Methods of Threads

1) start() :- This method is used to start a thread by calling its run() method.
2) run() :- Entry point for the thread execution.
3) sleep() :- This method is used to suspend a thread for a period of time.
4) setPriority() :- This method is used to assign the priority for the thread.
5) isAlive() :- This method is used to determine if a thread is still running.
6) stop() :- This method is used to kill a thread.

(7) yield :- This method is used to move a thread from running state to runnable state.

(8) wait() :- Tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify() method.

(9) notify() :- wakesup the first thread that called wait() method on the same object.

(10) notifyAll() :- wakesup all the threads that called wait() method on the same object. The highest priority thread will be run first.

Inter-Thread communication

Inter thread communication is achieved in Java by the following 3 methods :

1) wait()
2) notify()
3) notifyAll()

Thread Priorities

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. Threads of the same priority are given equal treatment by the java scheduler. Therefor, they share the processor on a first come first serve basis. Java permits us to set the priority of the thread using setPriority() method as follows.

Syntax

```
Threadname.setPriority(int number);
```

Here, 'int number' is an integer value to which the thread priority is set. The thread class defines several priority constants.

MIN_PRIORITY = 1

NORM_PRIORITY = 5

MAX_PRIORITY = 10

The 'int number' may assume one of the above constants or any value between 1 and 10.

The default priority of a thread 5.

```
/* Program to demonstrate Thread Priorities */

class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println("natural Thread "+i);
            try
            {
                Thread.sleep(520);
            }
            catch(InterruptedException e)
            {
                System.out.println("natural
                        Thread interrupted");
            }
        }
    }
}
class B extends Thread
{
    public void run()
    {
        for(int i=1; i<=10; i++)
        {
            system.out.println("even Thread "+(i*2));
```

```
        try
          {
            Thread.sleep(500);
          }
        catch(InterruptedException e)
          {
            System.out.println("even Thread
                                Interrupted");
          }
      }
    }
}

class ThreadPriorityDemo
  {
    public static void main(String args[])
      {
        A t1 = new A();
        B t2 = new B();
        t1.setPriority(3);
        t2.setPriority(8);
        t1.start();
        t2.start();
      }
}
```

Thread Synchronization

When 2 or more Threads need to access to a shared resource, they need some way to ensure that the resource will be using by only one thread at a time. The process by which this is achieved is called Synchronization.

Key to synchronization is the concept of the monitor also called semophore.

A monitor is an object that is used as a mutually exclusive lock or mutex. only one thread can 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 thread exists the monitor.

We can synchronize the code by using Synchronized keyword