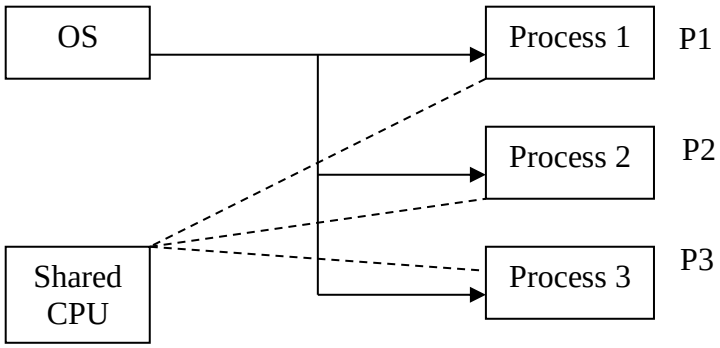
Multitasking :-

Multitasking is an OS concept by virtue of which two or more processes can run simultaneously that is at the same point of time. A process is actually a program in execution. Hence several programs can be run at the same time by virtue of the multitasking property of OS. In fact when several processes are executed, each one is allocated a time lice. For that quantum of time, the processes is executed by the CPU and then thrown out of it to execute the next waiting process. This mechanism is known as time sharing.



Process P_i is given time slice T_i

Multithreading :-

Multithreading is a programming concept which simulate multitasking multithreading in a feature of a program which allows multiple tasks to be performed at the same time, that is simultaneously. Here each tasks (actually method) is attached to a thread.

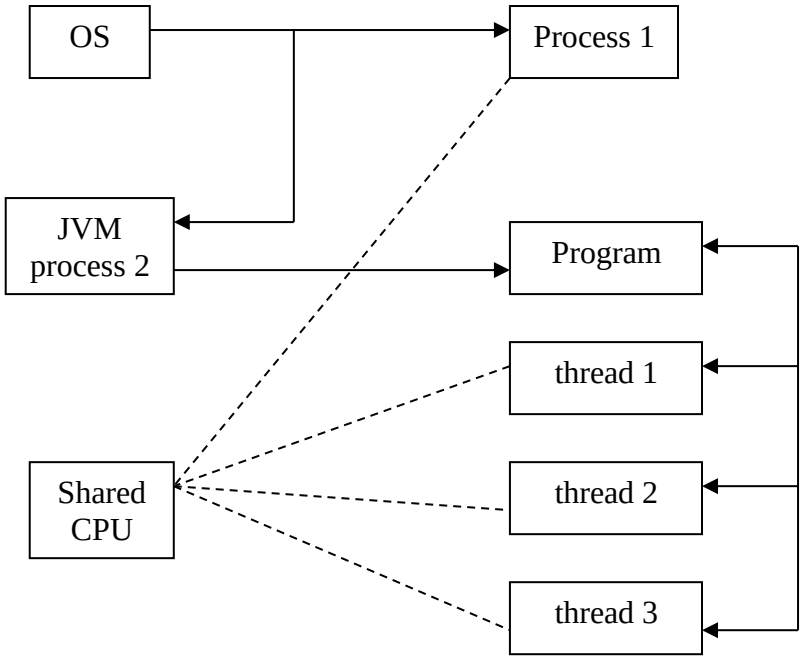
Thread :-

A thread is a single sequential flow of control. It can be thought of as a segment of code executing independently of any other job. A program may have multiple thread, each attached to a different method running simultaneously. For example, under multithreading one module of Bank A/c program may allow withdraw of money while another may show the balance.

Multithreading Vs. Multitasking :-

1. Multithreading is a programming concept which multitasking is an OS concept.
2. In Multithreading the smallest unit of code is a thread. In multitasking smallest unit of code is process.
3. Since threads are lightweight context switching and inter process communication (IPC) are less expensive. But processes being heavyweight context switching and IPC become more expensive.

JAVA AND MULTITHREADING :-



A Multithreaded Environment :-

Java provides built in support for multithreading that is java enables us to use multiple threads (flows of control) in a single java program. Each thread actually representing a method.

When a java program is run the main thread (representing the main() method) is triggered and run until the end of the method. The main thread can create several other thread (say for example— thread a, thread b, thread c). Once initiated all the four threads (main, a, b, c) runs simultaneously until the end.

Note:- All java enable web browsers (Internet Explorer, Netscape Navigator) support multithreading. For this reason we view a webpage search for another page and take print out of a third page all at the same time.

Writing Multithreaded Program :-

Multithreaded programs can be written in java by either extending the Thread class or implementing the runnable interface.

Extending the Thread class :-

1. Create a thread class of yours own which extends the class Thread.
2. Override the run() method of the Thread class in your own thread class, write the functionality of thread within this method.
3. Instancetiate your own Thread class.
4. Start the Thread by invoking the start() method using the instance created in step3.

Example :-

```
class ThreadA extends Thread {
        public void run() {
                for(int i=0; i<=3; i++)
                        system.out.println("ThreadA:i="+i);
                system.out.println("Exit from ThreadA");
}       }
class ThreadB extends Thread {
        public void run() {
                for(int i=3; i>0; i--)
                        system.out.println("ThreadB:i="+i);
                system.out.println("Exit from ThreadB");
}       }
class Thread Test {
        public static void main {
                Thread A ta = new ThreadA();
                Thread B tb = new ThreadB();
                ta.start();
                tb.start();
                system.out.println("Exit from Main Thread");
}       }
```

Output

```
ThreadA:i=0
ThreadA:i=1
ThreadB:i=3
ThreadA:i=2
Exit from Main Thread
ThreadA:i=3
Exit from ThreadA
ThreadB:i=2
ThreadB:i=1
Exit from ThreadB
```

Note :- We can design a name to a thread.

Example :-

```
class Mythread extends Thread {
        Mythread (String name) {
                Super(name);
                }
        public void run () {
                system.out.println("Hello");
                }
}
class Named Thread {
        public static void main () {
                Mythread mt = new Mythread ("India");
                system.out.println(mt);
```

Output

```
Mythread[India, 5, main]
Hello
```

```
                mt.start();
}       }
```

Note 2:- We could use the run() instead of the start method to execute the thread. But using the run method would make the program behave like a unithreaded one. That means if in the above example we use ta.run() and tb.run() then ThreadA would have executed in totally and then ThreadB.

Implementing Runnable interface :-

Multithreaded java program can also be written by implementing the interface called runnable. Following are the steps in writing a multithreaded program using runnable.

Step1:- Create your own thread class which implements the runnable interface.

Step2:- Implement the run() method of the runnable interface inside your own thread class. Within the run() write what the thread class is ought to do.

Step3:- Create an instance of your thread class. Create an instance of the built in Thread class passing the object of the user defined thread class as an argument to the built in Thread class.

Step4:- Start the thread by invoking the start method of the Thread class.

Example :-

```
class ThreadA implements Runnable {
        public void run() {
                for(i=0; i<=3; i++)
                        system.out.println("ThreadA: i="+i);
                system.out.println("Exit from A");
}       }
class ThreadB implements Runnable {
        public void run() {
                for(i=3; i>0; i--)
                        system.out.println("ThreadB: i="+i);
                system.out.println("Exit from B");
}       }
class RunnableTest{
        public static void main() {
                Thread ta = new ThreadA();
                Thread t1 = new Thread(ta);
                Thread t2 = new Thread(tb);
                t1.start();
                t2.start();
                system.out.println("Exit from Main");
}       }
```

Output

```
ThreadA:i=0
ThreadA:i=1
ThreadB:i=3
ThreadA:i=2
Exit from Main Thread
ThreadA:i=3
Exit from ThreadA
ThreadB:i=2
ThreadB:i=1
Exit from ThreadB
```

Note: What to chose?

The runnable interface only containing the run method. So when ever we want to use the run method only we use runnable interface. But when we need to use the methods like sleep(), yield(), wait(), notify(), etc. we need to use the Thread class.

Thread Priority :-

In java all threads are assigned a priority implicitly or explicitly. If not assign explicitly java sets a default priority to all its threads. Threads having the same priority level are executed in the CPU in either a round-robin fashion or any other scheduling scheme depending upon the OS. Explicitly we can set the priority of each thread by using the setPriority() method.

Syntax ThreadName setPriority(int value); // integer value is any integer from 1 – 10.

e.g.

```
thread Ata = new ThreadA();
ta.setPriority(8);
```

Java has some predefined priority levels i.e., MAX_RIORITY=10 NORM_PRIORITY=5, MIN_PRIORITY=1.

E.g. ta.setPriority(Thread.MAX_PRIORITY); //final variable

Note :- Whenever a thread with higher priority is brought to the runnable state, all lower priority threads are send to the blocked state.

Example :-

```
class ThreadA implements Runnable {
        public void run() {
                for(i=0; i<3; i++)
                        system.out.println("ThreadA: i="+i);
                system.out.println("Exit from ThreadA");
}       }
class ThreadB implements Runnable {
        public void run() {
                for(i=0; i<3; i++)
                        system.out.println("ThreadB: i="+i);
                system.out.println("Exit from ThreadB");
}       }
class TereadPriority {
        public static void main() {
                ThreadA ta = new ThreadA();
                ThreadB tb = new ThreadB();
                ta.setPriority(Thread.MIN_PRIORITY);
                tB.setPriority(TA.GETpRIORITY+5);
                ta.start();
                tb.start();
                system.out.println("Exit from Main");
}       }
```

Output

```
ThreadA:i=0
ThreadB:i=0
ThreadB:i=1
ThreadB:i=2
Exit from ThreadB
ThreadA:i=1
ThreadA:i=2
Exit from ThreadA
```

Example :- // Hashing the System time and date and cumulative sum of random numbers after every 1seconds.

```
import java.utill.*;
class DateThread estends Thread {
        public void run() {
                int p=0;
                try{
                        while(p<10){
                                system.out.println(newDate());
                                p++;
                                Thread.sleep(1000);
                                }
                }
                catch(Interrupted Exception ie){
                }
        }
}
class RandomSum extends Thread {
        int sum=0, p=0;
        public void run() {
                try{
                        while(p<10){
                                Random r = new Random();
                                int a = r.nextInt(10);
                                sum = sum+a;
                                system.out.println("Random No is"+a);
                                system.out.println("Sum is"+sum);
                                p++;
                                Thread.sleep();
                                }
```

```
                }
                catch(Interrupted Exception ie){
                }
}       }
class DatedRandomNo {
        public static void main() {
                DateThread dt = new DateThread();
                RandomSum rs = new RandomSum();
                dt.start();
                rs.start();
}       }
```

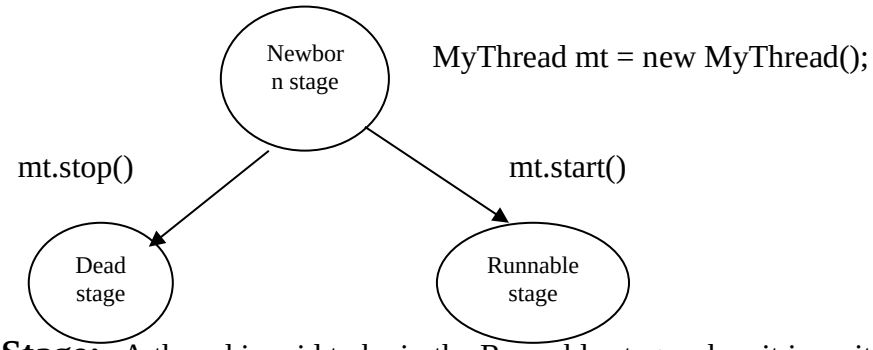
Lifecycle of a Thread :-

Throughout its life thread passes through several phases. The stages can be summarized as below:-

1. Newborn Stage.
2. Runnable Stage.
3. Running Stage.
4. Blocked Stage.
5. Dead Stage.

1. Newborn Stage:- A thread is in the newborn stage when it is just entered, it is neither in running as runnable stage. We can do the following things to a newborn thread:-

a. Kill the thread using the stop() method.
b. Start the thread using the start() method.



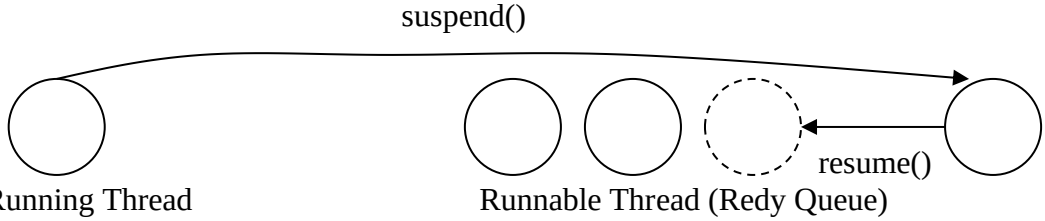
2. Runnable Stage:- A thread is said to be in the Runnable stage when it is waiting for its share of CPU time for being executed. The thread can run when ever it is sent to CPU.

3. Running Stage:- A thread is said to be in the running stage when it is enjoy its CPU time. The fate of the thread can be decided in the following way:-

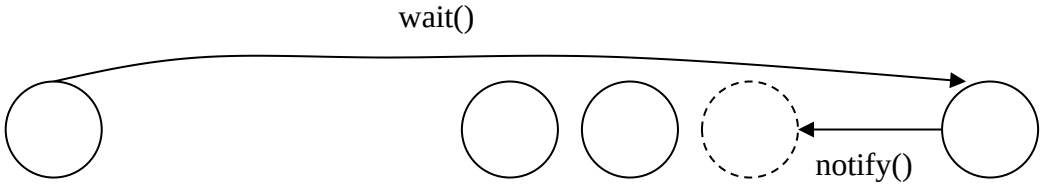
a. The thread completes its execution within that amount of time.
b. The thread relinquishes control to some other thread deliberately (next only).
c. The thread is sent to the blocked state by some mechanism.

4. Blocked Stage:- A thread can be sent to a blocked stage in the following ways.

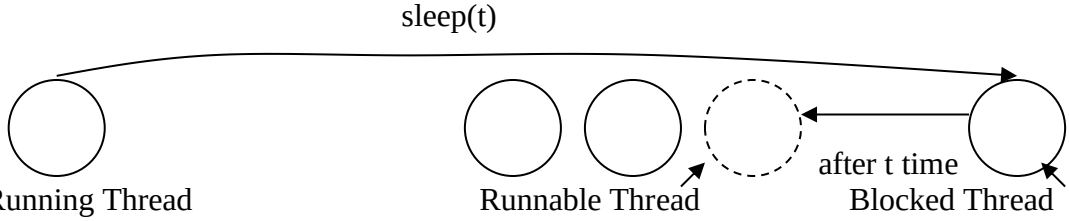
a. It can be paused for the time being by using the suspend() method and then it can be brought into the ready queue by using the resume() method.



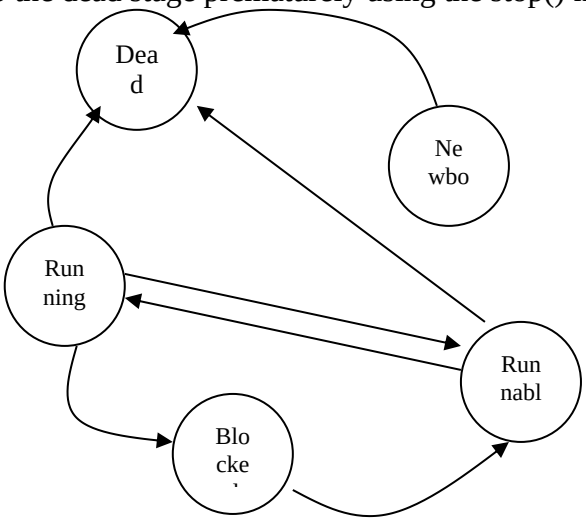
b. A thread can be made to wait for some condition to be happened using the wait() method and then re-invoked using the notify() method.



c. A thread can be made to sleep for certain period of time by using the sleep() method. Thread will be re-invoked after the time specified in the sleep() method



5. dead Stage:- A thread is sent to the dead stage when it has completed its execution. A thread can also be sent to the dead stage prematurely using the stop() method.



Complete Thread life cycle

Thread Methods :-

Some of the important thread methods are :-

1. yield() -> final void yield();
2. start() -> final void start();
3. stop() -> final void stop();
4. suspend() -> final void suspend();
5. wait() -> final void wait();
6. resume() -> final void resume();
7. notify() -> final void notify throws Interrupted Exception.
8. sleep(int value) -> final static void sleep(int value) throws Interrupted Exception.

Example:-

```
class ThreadA extends Thread {
        public void run() {
                for(int i = 0; i<5; i++){
                        system.out.println("ThreadA: i =" +i);
                        if(i==2)
                                yield();
                }
                system.out.println("Exit from ThreadA");
}       }
class ThreadB extends Thread {
        public void run() {
                for(int i = 0; i<5; i++){
```

```
                system.out.println("ThreadB: i =" +i);
                if(i==3)
                        stop();
        }
        system.out.println("Exit from ThreadB");
}       }
class ThreadC extends Thread {
        public void run() {
                for(int i = 0; i<5; i++){
                system.out.println("ThreadC: i =" +i);
                try{
                        if(i==1)
                                sleep(1000);
                }
                catch (Interrupted Exception ie){
                system.out.println(ie);
                }
        }
        system.out.println("Exit from ThreadC");
}       }
class ThreadMethodTest {
        public static void main() {
        ThreadA ta = new ThreadA();
        ThreadB tb = new ThreadB();
        ThreadC tc = new ThreadC();
        ta.start();
        tb.start();
        tc.start();
}       }
```

Synchronization :-

Multithreading, although it brings dynamicity to real time application, can give rise to several anomalies. E.g. — in a multithreaded Banking application, both the input thread and the show balance thread are invoked then operation may not reveal actual balance, giving rise to erroneous result. This happens to the fact that both threads run on an asynchronous manner.

Thus when two or more threads share the same resources, then there should be some mechanism to ensure that only one thread access the resources at a time. This serializibility is achieved by the method of synchronization. It can be done in two ways:-

1. By synchronized methods.
2. By synchronized statement.

1. Synchronized Method:-

We can achieve synchronization by declaring a method synchronized. This means that the method is accessible to a single thread at a time. When one thread has extended this synchronized method, it is provided with a mutually exclusive block/mutex/semaphore variables. So that no other thread can enter this method at the same point at time.

Example:-

```
class CallMe {
        void call(string msg) {
                system.out.println("[" + msg);
                try{
                        Thread.sleep(1000);
                }
                Catch(Interrupted Exception ie){
                        system.out.println(ie);
```

```
                }
        system.out.println("]");
}       }
class Caller extends Thread {
        CallMe obj = new CallMe();
        String msg;
        Caller(CallMe obj, String msg) {
                this.obj = obj;
                this.msg = msg;
        }
        public void run() {
                obj.call(msg);
}       }
class AsynchronousTest {
        public static void main() {
                CallMe obj = new CallMe();
                Caller C1 = new Caller(obj, "Hello");
                Caller C2 = new Caller(obj, "Java");
                Caller C3 = new Caller(obj, "How R U");
                C1.start();
                C2.start();
                C3.start();
}       }
```

Possible Output
[Hello [Java]
[How R U]
]

The reason for the above undesired is that all the three threads are using the call method simultaneously. When one thread is send to the blocked state using sleep(). Another thread is invoking the same method, thus disrupting the original sequence of output. To have the derived output, we have to synchronized the thread by declaring the call() as synchronized. By this we can ensure that even when one thread is sleeping no other thread can invoke the call method.

```
Synchronized void call(String msg) {
        //body
}
```

By doing this we have the output as

```
[Hello]
        [Java]
                [How R U]
```

2. Synchronized Statement :-

When two or more threads are executing same method simultaneously, we need to synchronize that method. But suppose the class where this method is written is inaccessible to us. Therefore, we just can't synchronized the method. In such a situation we can synchronized the call to that method (which was required to be synchronized) using a synchronized statement.

```
class CallMe {
        void call(string msg) {
                system.out.println("[" + msg);
                try{
                        Thread.sleep(1000);
                }
                Catch(Interrupted Exception ie){
                        system.out.println(ie);
                }
        system.out.println("]");
}       }
class Caller extends Thread {
        CallMe obj = new CallMe();
        String msg;
        Caller(CallMe obj, String msg) {
```

```
                this.obj = obj;
                this.msg = msg;
        }
        public void run() {
                synchronized(obj) {
                        obj.call(msg);
                }
}       }
class AsynchronousTest {
        public static void main() {
                CallMe obj = new CallMe();
                Caller C1 = new Caller(obj, "Hello");
                Caller C2 = new Caller(obj, "Java");
                Caller C3 = new Caller(obj, "Bye");
                C1.start();
                C2.start();
                C3.start();
}       }
```

Possible Output
[Hello [Java]
[Bye]
]

Deadlock :-

A special type of Error that may creep in to a multithread java program is Deadlock. A deadlock can occur when two thread are accessing two synchronized methods and the methods are such that they used the complesion of each other to complete themselves. Suppose threads t1 and t2 are using synchronized methods x and y. If the thread in x try to invoke the synchronized method y. It will be blocked as expected. If on the other hand the thread in y tries to invoke the synchronized methods x, the threads have to wait for ever. Because to access x, it would have to release its own lock on 'y' to show that the first thread may complete.

```
class A {
        Synchronized void meth1(B b) {
                system.out.println("Meth1 of A");
                try{
                        Thread.sleep(1000);
                }
                catch(Interrupted Exception ie){ }
                system.out.println("Trying to call last of B");
                b.last();
        }
        Synchronized void last() {
                system.out.println("Inside last of A");
}       }
class B {
        Synchronized void meth2(A a) {
                system.out.println("Meth2 of B");
                try{
                        Thread.sleep(1000);
                }
                catch(Interrupted Exception ie){ }
                system.out.println("Trying to call last of B");
                a.last();
        }
        Synchronized void last() {
                system.out.println("Inside last of B");
}       }
class DeadlockTest extends Threads {
        A a = new A();
        B b = new B();
```

```
        DeadlockTest() {
                this.start();
                a.meth1();
        }
        public void run() {
                b.meth2();
        }
        public void run() {
                DeadlockTest dt = new Deadlock();
}       }
```

Output
Meth1 of A
Meth2 of A
Trying to call last of A
Trying to call last of B
ctrl+c

Use of Multithreading :-

Multithreading enables us to write very efficient programs which make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive networked environment in which java works. E.g. — the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pack than they can be processed by the CPU and user input is also much slower than processor. In a traditional single threaded environment the program has to wait for each of these jobs to finish before it can proceed to the next one, even though the CPU is sitting idle for the most of the time. Multithreading leads us gain access to this idle time and put it to good use.