CHAPTER 5 : MUTITHREADING

A single sequential flow of control in a Java program is known as a Thread. Similarly multiple sequential flow of controls in a Java program is known as multiple thread or Multithreading. The concept by which OS becomes multiprocessing has been implemented in Java to make it Multithreading.

Advantages :

1- It allows a Java program to execute more than one task simultaneously i.e. a Java program can execute a music file at the same time it will insert the data on the database.
2- It allows a programmer to concentrate upon what they want to achieve by multithreading without concentrating on how the multiple threads is implemented by jvm on the OS(Operating System) because Sun has given different jvm for different OS, as well as it is dependent on the speed of processor.

How to create Thread ?

A class in which we like to provide multithreading functionally must inherit from Thread class or from Runnable interface available in java.lang package.

Creating a Thread by inheriting from Thread class.

A user defined class must inherit from the Thread class and it should override the predefine run() method of the Thread class as given below :

public void run ( )

When the above method will execute, then we will get a new flow of control in the Java program i.e. a child thread. But the run method must be invoked by the following start method of the Thread class.

public void start ( ) throws IllegalThreadStateException → unchecked exception.

```
class A extends Thread {
    public void run () {
      int x=0;
      while( x < 100 )
        System.out.println( "running : " + x++ );
        }
};
```

```
class B {
    public static void main(String args[]) {
        System.out.println( "begin main" );
        A a1 = new A();
        a1.start ();
        int y.1000;
        while( y < 1100 )
   System.out.println( "end main" + y++);
}};
```

start() method :

When this method is executed by the control of the main method, then it will pass a value i.e. process ID (PID) to the run method. The control of main method will never execute the run() method, it will never wait to see the execution of run method, rather it will ask the interpreter to execute run method, then it will come back to main method to execute other statement.

run() method:

When this method will get PID from the control of the main method, then it may execute immediately or may after sometime (if the a1 object in the above program has the higher priority, then the run for a1 object will execute immediately. If the a1 object has no priority or equal priority, the run method for a1 will start only when a1 will get the time slice). But whenever the run method will execute, then we will get a new flow of control in the Java program. This flow of control is called child thread or multiple threads. The flow of control execute is the main method is called main thread.

If we invoke the start ( ) more than once on the same object of the Thread class i.e. on a1, then it will generate illegal Thread state exception in the main thread.

run method VS. main method :

As the program execution start from main method similarly a child Thread's execution starts from run method.

As program will get some input before its execution from the run ( ), similarly command line, similarly a newly created thread will get some I/P before its execution from the start ( ).

WAP to create class A by inheriting from Thread class execute 4 threads of class A in class B to perform individual arithmetic operation (+, -, *, /) of two numbers given from command line of class B.

Process based Multitasking : -

Executing several tasks simultaneously where each task is independent of other is known as Process based multitasking.

Example: While writing a java program we are printing, listening to the music as well as we can download a file. These tasks are independent of other task and their executing simultaneously .Hence these are called Process based multitasking.

The Process based multitasking is based suitable at OS level.

Thread based Multitasking : -

Executing several tasks simultaneously where each task is independent of other task, but they are a part of same program and they are known as Thread based multitasking and each part is called a thread.

The Thread based multitasking is best suitable for a program. Whether its Process based multitasking or Thread based multitasking the main purpose of multitasking is to reduce the time and to improve the performance of a system or a program.

The main important applications of multitasking are to develop multimedia, graphics, animation, games & web application etc.

major difference between Process and Thread is that, each process has its own separate memory space but Threads from same process same memory space.

1. Both process and thread are independent path of execution but one process can have multiple threads.
2. Every process has its own memory space, executable code and a unique process identifier (PID) while every thread has its own stack and unique identification in Java but it uses process main memory and share it with other threads.
3. Threads are also refereed as task or light weight process (LWP) in operating system.
4. Threads from same process can communicate with each other by using Programming language construct like wait and notify in Java and much simpler than inter process communication.
5. Another difference between Process and Thread in Java is that , how Thread and process are created. It's easy to create Thread as compared to Process which requires duplication of parent process.
6. All Threads which is part of same process share system resource like file descriptors , Heap Memory and other resource but each Thread has its own Exception handler and own stack in Java.

There were some of the fundamental difference between Process and Thread in Java. Whenever you talk about Process vs Thread, just keep in mind that one process can spawn multiple Thread and share same memory in Java. Each thread has its own stack.

Analysis of a Thread : - **Case-1: Creating a Simple Thread**

```
class A extends Thread{
public void run(){
int x=0;
while(x<6)
    System.out.println("Running"+x++);
}}
```

```
class B{
public static void main(String ar[]){
    System.out.println("Begin main");
    A a1=new A(); //Thread instantiated
    a1.start(); //Thread start (i.e it is in runnable state)
    System.out.println("End main");
}};
```

If we have multiple threads which are waiting to execute then they are dependent on this thread scheduler to decide the order of execution. The thread scheduler is the part of the JVM and it may execute any thread randomly because when all the threads are in equal priority then they wait in an area is known as thread pool and thread scheduler will choose any one of them randomly to execute. For the above reserve we can't expect the exact output of a thread. Therefore same program may generate different output on different system.

Output 1	Output 2	Output 3	outputs 4
Begin main	Begin main	Running 0	Begin main
End main	Running 0	.	Running 0
Running 0	Running 1	.	.
.	End main	Running 4	.
.	Running 2	Running 5	Running 4
Running 4	.	Begin main	Running 5
Running 5	Running 5	End Main	End Main

Case-2 : Difference between a1.start () and a1.run ()

In case of a1.start () the new thread which has already been created by object a1 may become runnable i.e. ready to execute i.e. start () is responsible to complete the execution of run ().

If we write a1.run () in the above program, then even if the thread object is created (a1) still the new thread i.e. a1 will never execute because during this time the main thread will execute body/job of the thread i.e. run () and we will never get the output from the new thread.

Case 3 : Importance of predefined Thread class

The start () of the thread class is responsible to register our thread instance (a1) with the thread scheduler and all other activities like calling the run () from the predefined start ().

```
public class Thread extends Object implements Runnable{
    -------------
    public void run(){
      --------------
    }
    public void start() throws IllegalThreadStateException{
       try{
         1.Register the thread with thread scheduler by PID.
         2.Process all other activities.
         3.Invoke the run() of child class/user defined class
         }catch(Exception e){}
    }
};
```

If we don't invoke the start () we can't get separate control i.e. separate thread in our java program & the output is never considered as multithreading rather it will become single thread output. Therefore start () is considered as heart of multithreading.

Case-4

If we overload the run() in our class, then the start() will never call the overloaded run() rather it will execute the overridden run().

```
class A extends Thread{
public void run(){
    System.out.println(" I am run");
}
public void run( int p){
    System.out.println(" I am overloaded run");
}}
```

```
class B{
    public static void main(String ar[]){
        A a1=new A();
        a1.start();
}};
```

```
Output
---------
I am run
```

class A extends Thread{}
class B{
public static void main(String ar[]){
A a1=new A();
a1.start();
}};
	Output –I t will compile but there will be no output because in the thread class in run() there will be no parameter given.
	Therefore it is recommended that we should override the run(),otherwise we will not get the multithreading functionality.

Case-5 : Overriding the start()

class A extends Thread{
public void start(){
System.out.println("Hi start");
}
public void run(){
int x=0;
while(x<6)
System.out.println("Running :
"+x++);
}};
	class B{
	public static void main(String ar[]){
	System.out.println("Begin Main");
	A a1 = new A();
	a1.start();
	System.out.println("End Main");
	}};
		Output
		--------- 	
		Begin Main
		Hi start
		End Main

class A extends Thread{
public void start(){
super.start();
System.out.println("Start in A");
}
public void run(){
System.out.println("Run in A");
}};
	class B{
	public static void main(String ar[]){
	System.out.println("Begin Main");
	A a1=new A();
	a1.start();
	System.out.println("End Main");
	}};
		Output
		---------- 	
		Begin Main
		Start in A
		End Main
		Run in A

Constructor of Thread class:

1) Thread ()
2) Thread (Runnable rb)
3) Thread (Runnable rb, String name)
4) Thread (String name)

If we create a thread object by 1st or 2nd constructor, then each thread will get a name as given below:

"Thread-Count"

Where Count=0, for 1st thread object. Count =1, for 2nd Thread objects and so on.

If we create the thread object by 3rd or 4th constructor, then the count value will not increase for these objects because these constructors will accept the name as thread in the parameter.

```
class A extends Thread {
 A() { }
 A(String p) { super(p); }
  public void run( ) {
   int x=0;
   while(x<5)
System.out.println(getName()+ " : "+ x++);
} };
class B {
   public static void main(String gs[]){
        System.out.println("begin main");
        A a1 = new A();
        A a2 = new A("ram");
        A a3 = new A();
        a3.setName("ravan");
        A a4 = new A();
        a1.start();
        a2.start();
        a3.start();
        a4.start();
        System.out.println("end main");
}}
```

Methods of Thread class :

public final void **setName**(String name) – This method of the Thread class will assign a name to a thread.
public final String **getName**() – This method of the Thread class will return the name of a thread.

public static Thread **currentThread**() – This method of the Thread class is used to return an object of the Thread that executes it.

public final boolean **isAlive**() – This method of the Thread class is used to determine wheather a thrad is active or not. It'll return true, if a Thread has invoked by start() or the thread is running. It will return false is a thread has not been invoked by start() or it has completed the run().

public final void **join**() throws InterruptedException – This method of the Thread class is used to suspend the execution of a thread for another thread. The thread which will execute this method will become suspended for the thread that has invoked this method. The thread will resume only when the thread that invoked this method will complete.

```
class B extends Thread {
 public void run() {
int x=0;
while(x<15)
System.out.println(getName() + " : " +x++);
}
public static void main(String ar[]) throws Exception {
System.out.println("begin main");
B b1 = new B();
```

```
B b2 = new B();
System.out.println(b1.isAlive());
b1.start();
b2.start();
b1.join();
System.out.println(b1.isAlive());
System.out.println(b2.isAlive());
System.out.println("end main");
}};
```

public static final void **sleep**(long millisecond) throws InterruptedException :- This method of the Thread class is used to suspend the execution of a thread for given millisecond. The thread will resume when its sleep interval will complete. The thread which will execute this method will remain suspended. Since it is static method we can use it anywhere to provide a delay.

1. Thread.sleep() is a static method & it always puts the current thread to sleep.
2. You can wake-up a sleeping thread by calling interrupt() method on the thread which is sleeping.
3. The sleep method doesn't guarantee that the thread will sleep for exactly that many milliseconds, its accuracy depends on upon system timers and it's possible for a thread to woke before.
4. It doesn't release the lock it has acquired.

```
class A extends Thread {
  public void run() {
    int x=0;
    while(x<15) {
  if(x==5 && getName().equals("Thread-0") )
    try {
       Thread.sleep(100);
    }catch(Exception e) { }
   System.out.println(getName() +" :" + x++);
}}}
```

```
class B {
public static void main(String ag[]) throws Exception {
 System.out.println("begin main");
 A a1 = new A();
 A a2 = new A();
 a1.start();
 a2.start();
 Thread.sleep(100);
// a1.sleep(100);
 System.out.println("end main");
}};
```

Daemon thread : Daemon is a never ending process that runs at background e.g. system clock.
The thread class has not given any function or method to create a daemon thread, but according to its specification we can create a daemon thread by using a indefinite or never ending loop in the run ().

```
class A extends Thread {
 public void run() {
  int x=0;
  while(true)
System.out.println(getName() +" : " + x++);
}
```

```
public static void main(String args[]) {
System.out.println("begin main");
A a1 = new A();
a1.setDaemon(true);
a1.start();
System.out.println("end main"); }};
```

public final void **setDaemon**(boolean on) - This method of the Thread class determines whether a thread is daemon or not. If we provide true to this method the setDaemon(), then the daemon thread will terminate after the main thread, otherwise the daemon thread will not terminate.

Priority: It is an integer value by which a thread will get weightage. The thread class has given the following method & constants to set the priority.

```
public static final int MAX_PRIORITY = 10;
public static final int NORM_PRIORITY = 5;
public static final int MIN_PRIORITY = 1;
```

The Thread class has allowed us to provide the priority value from 1 to 10. If we provide other value than this then it will generate run time exception.
public final void. **setPriority**(int value) :- It will assign int value as the priority of the thread.
public final int **getPrority** () :- It will return the priority of the thread.

The algorithms by which the Threads are scheduled depending upon their priority are known as scheduling algorithm. When we don't provide any priority of threads, then Java uses fixed priority scheduling in which it will provide equal priority i.e. normal priority or 5 for each thread.

```
class Prio extends Thread {
 public void run() {
   int x=0;
   while(x<15)
System.out.println(getName()+" : "+ x++);
}
public static void main(String args[]) {
  System.out.println("begin main");
  Prio p1 = new Prio();
  Prio p2 = new Prio();
  p1.start();
  p2.start();
  p1.setPriority(Thread.MIN_PRIORITY);
  p2.setPriority(10);
  System.out.println("end main");
}};
```

When we set any priority for multiple threads, then java uses 'preemptive' scheduling in which, when a highest priority thread will become ready to execute , then the lower priority threads will be preempts (removes by force) so that the higher priority thread will execute.

Creating a thread by inheriting from Runnable interface : - The predefined Thread class has inherited from predefined Runnable interface. This interface has declared only the run(). Hence a class which will inherit from the Runnable interface to create a thread must override the run().

```
class C implements Runnable{
  public void run() {
  int x=0;
   Thread t = Thread.currentThread();
  while(x<4)
 System.out.println( t.getName()+ " run in C " +
x++);
   }};
```

```
class D {
  public static void main(String args[]) {
  C c1 = new C();
//  c1.start();
Thread t1=new Thread(c1);
  t1.start();
  Runnable a1 = new C();
 Thread t2 = new Thread(a1, "rahul");
   t2.start();
}};
```

Thread vs Runnable in Java

Here are some of my thoughts on whether I should use Thread or Runnable for implementing task in Java.

1. Java doesn't support multiple inheritance, which means you can only extend one class in Java so once you extended Thread class you lost your chance and cannot extend or inherit another class in Java.
2. In Object oriented programming extending a class generally means adding new functionality, modifying or improving behaviors. If we are not making any modification on Thread than use Runnable interface instead.

3. Inheriting all Thread methods are additional overhead just for representing a Task which can can be done easily with Runnable.

Synchronized method :-

```
class Que {
static int cnt=72;
void booking(String na) {
System.out.println("[ booking ticket for : " + na);
int x=0;
while(x<123456789) {
 x=x+2; x=x-1; }
 System.out.println("  ticket no for : " +na+ "  is  " + cnt--);
}};
class Person extends Thread {
String p;
Que ob;
 Person(String pp, Que qq) {
   p=pp;   ob= qq;   }
 public void run() {
     ob.booking(p);
 }};
```

```
class Station {
public static void main(String gs[]) {
Que q1 =new Que();
Person p1 = new Person("king",q1);
Person p2 = new Person("ming",q1);
Person p3 = new Person("ping",q1);
Person p4 = new Person("ting",q1);
p1.start();
p2.start();
p3.start();
p4.start();
}};
```

When multiple thread access a resource simultaneously, then there is a competition among the threads to acquire the resource. When some of them will succeed to access the resource then others may fail or others will acquire the resource after some time. Hence they will produce incorrect result. (When p1,p2, p3, p4 Threads will access the booking method on q1 object, then there will be a competition among the threads to access the booking method. Hence they will generate incorrect result). These conditions of multiple threads are called race condition.

OS uses Semaphore or Mutex lock or lock to avoid race condition between multiple processes. The same concept has been implemented in Java by **synchronized** keyword.

Example:- let's provide synchronized keyword before the return type of booking method of Que class, then it is called synchronized method.

When a thread will access a Synchronized Method, then it will acquire a lock on the Synchronized Method and the object by which Synchronized Method is invoked. During this time other threads are not allowed to access the same Synchronized Method on same object. When the thread will complete the execution of Synchronized Method then it will release the lock, so that another thread will acquire the lock to execute Synchronized Method. (When p1 will access the booking method on q1 object then it will acquire a lock on q1 & booking method at the same time p2, p3, p4 threads are not allowed to invoke the booking method on q1 object. When p1 will complete booking method, then p1 will release the lock it has hold on the booking method and q1 object, so that p2/p3/p4 will access the booking method on q1 object by acquiring the lock).

When a thread will access a Synchronized Method on an object at the same time another thread is also allowed to access the same Synchronized Method on another object. (When p1 thread will access the booking method on q1 object, at the same time p2 Thread is also allowed to access the booking method on q2 object, say q2 is another object of class Que.)

Synchronized Block:- If we provide synchronized keyword on an object then it is called Synchronized Block For example:- lets remove the synchronized keyword from booking ( ) of Que class and lets modify the run ( ) of the person class as given below :

```
public void run ( ) {
   synchronized (ob){
     ob.booking (p);
}}
```

When a thread will access a Synchronized Block on an object, then it will acquire a lock on the object, during this time other threads are not allowed to access the Synchronized Block on the same object(When p1 thread will access the Synchronized Block on q1 object, then it will acquire a lock on q1 object, during this time p2/p3/p4 threads are not allowed to access the Synchronized Block on the same q1 object. But when p1 will release the lock on q1 object then p2/p3/p4 is allowed to access the Synchronized Block on q1 object).

When a thread will access a Synchronized Block on an object at the same time another thread is also allowed to access the same Synchronized Block on another object(When p1 thread will access the Synchronized Block on q1 object, at the same time p2 thread is allowed to access the same Synchronized Block on another object of class Que called q2).

If a thread will access 3 different Synchronized Method on an object then it has to acquire and release the lock 3 times. But if a thread will access 3 different methods from a Synchronized Block, then it will acquire and release the lock only once. Hence we are saving some clock cycles of CPU in SB by acquiring & releasing the lock only once. Hence Synchronized Block is faster than Synchronized Method.

If two threads will invoke 3 different Synchronized Method on an object, then we cannot guarantee that a thread will execute all Synchronized Method Sequentially. But in two thread will access 3 different method from a Synchronized Block then we can guarantee that a thread will execute all method sequentially.

Object Level Lock: When Synchronized Method is non static and a thread executes the Synchronized Method, then it is said that the thread has acquired object level lock.

Class Level Lock: When a Synchronized Method is static and a thread executes the Synchronized Method, then it is said that the thread has acquired class level lock.

About Synchronize Key word :

- ✓ The synchronized keyword is NOT considered part of a method's signature.
- ✓ It is not automatically inherited when subclasses override super class methods.
- ✓ Methods in Interfaces cannot be declared synchronized.
- ✓ Constructors cannot be declared synchronized however they can contain synchronized blocks.
- ✓ If you make main() method as synchronized and it will never affect the child thread.

How to write Thread-Safe Code in Java.

thread-safety or thread-safe code in Java refers to code in which if one thread is operating on any code, class or object then other threads are not allowed to access on it, the code which will allow multiple thread to access it on mutil threading environment is not thread-safe, thread-safety is one of the risk introduced by using .

```
public class Counting {
   private int count;
   public int getIncreValue(){
      return count++;
   }
}
```

Above example is not thread-safe because ++ (increment operator) is not an atomic operation and can be broken down into read, update and write operation. if multiple thread call getIncreValue() approximately same time each of these three operation may coincide or overlap with each other for example while thread 1 is updating value, thread 2 reads and still gets old value, which eventually let thread 2 override thread 1 increment and one count is lost because multiple thread called it concurrently.

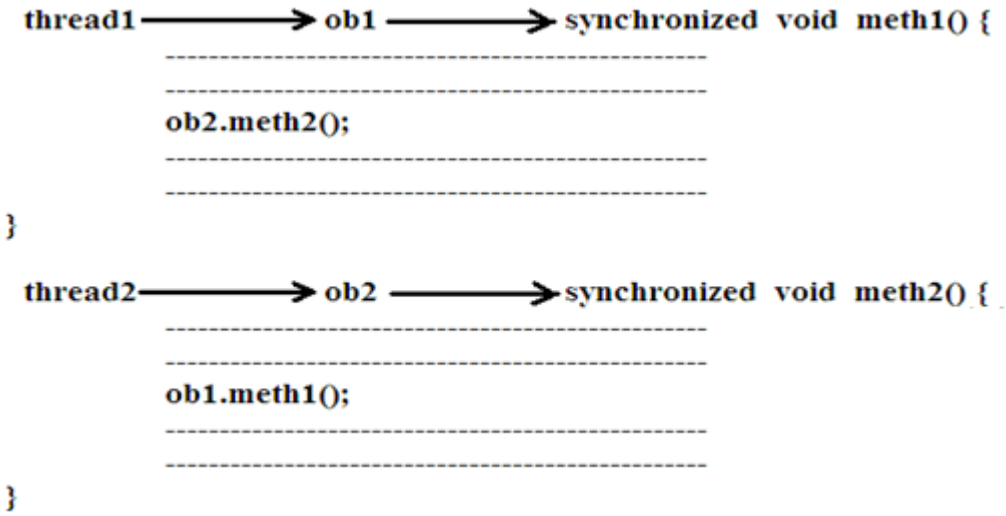
There are multiple ways to make this code thread safe in Java:

1) Use synchronized keyword in Java and lock the getIncreValue() method so that only one thread can execute it at a time which removes possibility of coinciding or interleaving.

2) Use AtomicInteger class (it creates an int value that may be updated atomically), which makes this ++ operation atomic and since atomic operations are thread-safe and saves cost of external synchronization.

```
public class Counting1 {
   private int cnt;
   AtomicInteger a1 = new AtomicInteger(0);
   public synchronized int getIncreValue(){  //we've applied syhnchronized
      return cnt++;
   }
   public int getCountAtomically(){
      return a1.incrementAndGet();  //it will not allow multiple thread to operate simultaneously.
   }
}
```

Deadlock : - When two threads will access 2 different Synchronized Method on two objects and during their execution they want to invoke the Synchronized Method executed by the other thread without releasing the lock they are holding on. Hence the [programme will halt & the situation is known as dead lock.



We can avoid the dead lock by making intercommunication between the threads.

Intercommunication between the threads :- The intercommunication between the threads can be done by the following methods of Object class which are inherited to Thread class. All the above methods must be used within any synchronized method.

```
class  Z  {
synchronized  void   test() {
  int  x=0;
  Thread  ob= Thread.currentThread();
  while(x<50)    {
    if(ob.getName().equals("Thread-0")
&&  x==12)
    try {
       wait();
    }catch(Exception  e) {  }
    System.out.println(ob.getName() +  " :
" + x++);
 }
 notify();
}};
```

```
class  A extends Thread  {
 Z  ob;
  A(Z  zz){  ob=zz;  }
  public void  run() {
      ob.test();
  }}
class  D  {
 public static void main(String ar[]) {
   Z   z1 =  new  Z();
  System.out.println("begin main");
  A   a1=  new   A(z1);
  A   a2=  new   A(z1);
  a1.start();
  a2.start();
  System.out.println("end main");
}};
```

```
class  Que {
  int x;
   synchronized  int  get(){
System.out.println("got : "+ x);
    return  x;
  }
synchronized void  put(int p) {
    x=p;
System.out.println("       put : " +x);
  }};

class Consumer extends Thread {
Que  q;
   Consumer(Que  qq){ q=qq; }
   public  void   run() {
     while(true)
       q.get();
  }};
```

```
class Producer extends Thread {
    Que  q;
    Producer(Que qq){ q=qq; }
    public void run() {
      int  i=0;
      while(true)
         q.put(++i);
  }};
  class Shop {
  public static void main(String  args[]) throws Exception {
    Que ob= new Que();
  Consumer c= new Consumer(ob);
  Producer p =new Producer(ob);
  c.setDaemon(true);
  p.setDaemon(true);
  c.start();
  p.start();
  Thread.sleep(500);
 }};
```

In the above program, the producer is producing the value without waiting for consumer. The consumer is consuming the same value over & over again without waiting for the producer, because there is no intercommunication between the threads. To make the intercommunication between the Producer and the Consumer threads, the class Que should be modified as given below :

```
class  Que {
  int x;
 boolean flag;
 synchronized  int  get(){
   if(!flag)
      try {
           wait();
      }catch(Exception  e){    }
   System.out.println("got : "+ x);
   flag=false;
   notifyAll();
   return  x;
  }
```

```
synchronized void  put(int p) {
    if(flag)
      try {
            wait();
     }catch(Exception  e){    }
     x=p;
System.out.println("       put : " +x);
   flag=true;
   notifyAll();
  }};
```

wait method:-If we use sleep method within a Synchronized Method, then the thread will remain Suspended but it will never release the lock it has hold on the Synchronized Method.

If we use wait method within a Synchronized Method, then it will remain suspended as well as it will release the lock. It has hold on Synchronized Method. The thread which is suspended by wait method will resume only when another thread will execute notify() method or notifyAll() method.

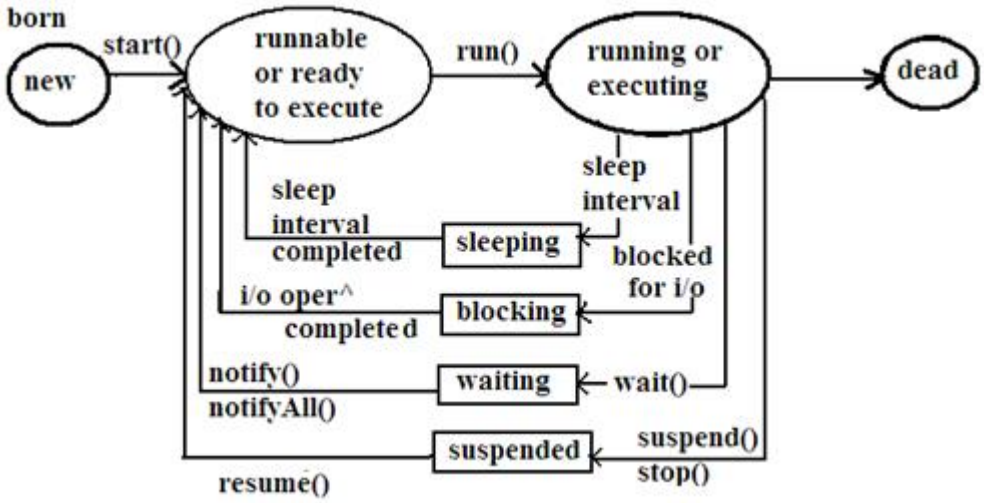
notify(): - This method of Object class must be executed by a thread which is about to complete the execution of a Synchronized Method. When a thread executes notify method then the jvm will wake up the thread suspended by wait method.

If there is a higher priority thread available among the bunch of thread suspended by wait method, then the jvm will notify the higher priority thread about the release of the lock, so that the higher priority thread will resume. If there is equal priority among the bunch of thread suspended by wait method, then after notify method the jvm will wake up any one of the thread randomly to provide the lock, it may happen that a thread which has been suspended by wait method for a long time may not acquire the lock hence the thread will starve for the lock it is called starvation of thread.

OS uses aging mechanism to avoid the Starvation by increasing the priority of a thread which has been suspended for a long time period(or for certain no of time slice). The same concept has been implemented in Java by notifyAll method.

notifyAll ( ):- It is similar to notify method except jvm calculates the no of times slices that has been elapsed since a thread is waiting so that the jvm will increase the priority to avoid Starvation.

LIFE CYCLE OF A THREAD : -



The stop(), suspend() & resume() are deprecated from java1.2 (A method which will not be used in current or in future version of java is known as deprecated, java does not encourage us to use these method).

Different States of a Thread :

- **New Born State:** When we create a Thread Object then a Thread is born.
- **Runnable State:** When we call the start( ) method on a thread then thread is in the Runnable state i.e., the threads in Runnable state are all waiting for the availability of CPU attention for their turns to come.
- **Running State:** When a thread got the time of CPU then it starts execution for a particular interval of time (time slice) this state is called as Running state.
- **Blocked state:** A thread is in the blocked state when it is prevented to enter into Runnable state. A Thread enters into blocked state under the following situation.
  - ✓ By sleeping a thread.
  - ✓ If a thread is waiting for I/O operation.
  - ✓ If a thread is Interrupted.
  - ✓ If a thread is waiting for lock available.
- **Dead state:** This is the end of life of a thread, when a thread completes execution comes to the dead state. Also we can dead state by calling stop( ) method.

ThreadGroup Class:

- Every Java thread is a member of a **Thread Group.**
- ThreadGroup facilitate to operate or manage threads as group.
- Java runtime system creates a Thread Group named "**main**".
- If any thread is created without a thread group specification , the runtime system put the thread into main.
- Once a thread became a member of a group it can't be removed from that group.

Creating a Thread Group :	ThreadGroup tg = new ThreadGroup("rama");
Putting Threads into ThreadGroup:	Thread t1 = new Thread(tg, this);
Getting a Thread's group:	ThreadGroup group = debThread.getThreadGroup();

```
public class GroupTest{
public static void main(String ar[]){
        ThreadGroup group=Thread.currentThread().getThreadGroup();
        System.out.println(group.getName());          //      main
        System.out.println(group.activeCount());              //      1
```

```
}}
```

volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory. So if we want to share any variable(int, Boolean) in which read and write operation is atomic by multiple thread, then it can be declared as volatile variable. volatile cannot be used with method or class and it can only be used with a variable. We can use Volatile variable to acheive followings:

1. If you want to read and write long and double variable atomically. long and double both are 64 bit data type and by default writing of long and double is not atomic and platform dependent. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this it is possible for a Thread to see 32 bit from two different write.We can avoid this issue by making long and double variable volatile in Java.
2. A volatile variable can be used as an alternative way of achieving synchronization in Java in some cases, like visibility. With volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed by a single thread, without volatile keyword different reader thread may see different values.

Stopping a Thread in Java by volatile variable : Here the main thread is first starting a thread and later it's stopping that thread by calling our stop() method, which uses a boolean volatile variable to stop running thread.

```
import    java.util.concurrent.TimeUnit;
public class StopDemo {
   public static void main(String args[]) throws
InterruptedException {
      A  a1 = new A();
      Thread t1 = new Thread(a1, "one");
      t1.start();

      String na=Thread.currentThread().getName();
      //Now, let's stop our A thread
      System.out.println(na + " is stopping A
thread");
      a1.stop();

      //Let's wait to see server thread stopped
      TimeUnit.MILLISECONDS.sleep(500);
      System.out.println(na + " is finished now");
}};
```

```
class A implements Runnable{
   private volatile boolean exit = false;
   public void run() {
      while(!exit){
         System.out.println("A is running.....");
      }
      System.out.println("A is stopped....");
   }
   public void stop(){
      exit = true;
}};
```

It is clear that our Server, which implements Runnable is running fine until main() method called the stop() method, the only then value of boolean volatile variable exit is set to true. In next iteration, our thread checks and find this value true, so it come out of the loop and completes run() method, which means thread is now stopped.

Thread Interrupts: A thread can signal another thread that is should stop executing by calling interrupt() method for that thread object. If this thread is blocked in an invocation of the wait(), join(), sleep(long) methods, then its interrupt status will be cleared and it will receive an InterruptedException.

```
class  A  extends Thread{
   A(){          start();                }
   public void run(){
      int i=1;
      while(i <2000){
         System.out.println(i++);
         if(Thread.interrupted())
         System.out.println("Thread Inrrupted..");
      }
}}
```

```
class  InterruptedDemo{
 public  static  void  main(String  args[])  throws
Exception{
    A  a1= new  A();
    Thread.sleep(50);
    if( !a1.isInterrupted()){
         a1.interrupt();
         System.out.println("Interrupted by main");
    }
}}
```