

# **Multithreading**

- 1) Introduction
- 2) The ways to define instantiate, and a start a thread.
- 3) Getting and setting name of a thread.
- 4) Thread priorities.
- 5) The methods to prevent thread execution.
  - a)yield();
  - b)join()
  - c)sleep
- 6) Synchronization
- 7) Inter Thread communication
  - a)wait();
  - b)notify
  - c)notifyall();
- 8) Deadlock
- 9)Daemon Threads.

# Introduction

## **Multitasking:**

Executing several task simultaneously is called “Multitasking”.  
There are two types of multitasking.

1)Process based multitasking

2)Thread based multitasking

Example:

Students in classroom-

a)listening

b)writing

c)sleeping

d)mobile operating

e)watching

## **1)Process based multitasking:**

Executing several task simultaneously, where each task is separate independent process is called ” process based multitasking.”

Example:

While typing a java program in editor we are able to listen audio songs by mp3 player in the system at the same time we can downloaded a file from the internet. All these task are executing simultaneously and independent of each other. Hence it is process based multitasking.

Process based multitasking is best suitable at “O.S.level”.

## **2)Thread based multitasking:**

Executing several task simultaneously,where each task is a separate independent part of the same program is called “Thread based multitasking”.

And each independent part is called “Thread”.

It is best suitable for programmatic level.

- Whether it is process based or thread based the main objective of multitasking is to improve performance of the system by reducing response time.
- The main important application areas of multithreading are developing video games, multimedia graphics, implementing animations.
- Java provide inbuilt support for multithreading by introducing a rich API (Thread, Runnable, ThreadGroup, ThreadLocal--). Being a programmer we have to know how to use this API and we are not responsible to define that API. Hence developing multithreading program is very easy in java when compared with c++.

## 2)The Way to define Instantiate and start a new Thread

**We can define a thread in the following two ways:**

- 1)By extending thread class.
- 2)By implementing Runnable Interface.

### **1)Define a thread by extending thread class:**

Example:

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

```
    }  
    }  
    }  
class ThreadDemo  
{  
    public static void main(String ar[])  
    {  
        threading t=new threading();  
        t.start();  
        for(int i=0;i<=10;i++)  
        {  
            System.out.println("Main Thread");  
        }  
    }  
}
```

Output:

Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
Main Thread  
child thread  
child thread  
child thread  
child thread  
child thread  
child thread

child thread  
child thread  
child thread  
child thread  
child thread

**Case 1:Thread Scheduler:**

- When ever multiple threads are waiting to get chance for execution which thread will get chance first is decided by thread scheduler.Whose behavior is JVM vendor dependent.Hence we can't expect exact execution order and hence exact output.
- Thread scheduler is part of JVM,due to this unpredictable behavior and thread scheduler we can't expect exact output from above program
- The following are various possible output:

p-1	p-2	p-3	p-4
Main Thread	child thread	child thread	Main Thread
Main Thread	child thread	Main Thread	Main Thread
Main Thread	child thread	Main Thread	child thread
-	-	Main Thread	child thread
-	-	-	Main Thread
-	-	-	Main Thread
child thread	Main Thread	child thread	Main Thread
child thread	Main Thread	child thread	-
child thread	Main Thread	-	-
-	-	Main Thread	-
-	-	-	-
-	-	-	-

Note:

Whenever the situation comes to multithreading the guarantee in behavior is very less. we can tell possible output but not exact output.

### Case 2: Difference between `t.start()` and `t.run()`:

- In the case of `t.start()` a new thread will be created and thread is responsible to execute `run()`.
- But in the case `t.start()` no new thread will be created and `run` method will be executed just like a normal method call.
- In the above program if we are replacing `t.start()` with `t.run()`
- the following is the output:

Output:

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

thread

child thread

child thread

child thread

child thread

child thread

Entire output produced by only main

child thread  
child thread  
child thread  
child thread  
child thread  
child thread

### **Case 3)Importance of Thread class start() method:**

- To start a Thread,the required mandatory activities(like registering Thrad with Thread Scheduler) will be perform automatically by Thread class start() method.Because these facility programmer is not responsible to perform these activity and he is just responsible to define job of the Thread.Hence Thread class start() method plays very important role and without executing that method there is no chance of starting a new thread.

Example:

```
class Thread
{
Start()
{
1.Register this thread with thread scheduler and perform
other initialization of activities.
2.run()
}
}
```

### **Case 4: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 hence we won't get any output.

**Example:**

```
public class mythread extends Thread
{
}
class threaddemo
{
    public static void main(String ar[])
    {
        mythread t=new mythread();
        t.start();
    }
}
```

Output:No output printing.

Note:

It is highly recommended to override run method to define our job.

**Case 5:Overloading of run() method:**

- Overloading of run() method is possible but Thrad class start() method will always call no argument run() only. But the other run method we have to call explicitly just like a normal method call.

Example:

```
public class mythread extends Thread
{
    public void run()
    {
        System.out.println("run()");
    }
    public void run(int i)
    {
        System.out.println("run(int i)");
    }
}
```



```
    }  
  }  
  class threaddemo  
  {  
    public static void main(String ar[])  
    {  
      mythread t=new mythread();  
      t.start();  
      t.run(5);  
    }  
  }
```

Output:

```
run(int i)  
run()
```

#### **Case 6)Overriding of start() method:**

- If we override start().Then start() will be executed just like a normal method call and no new Thread will be created.

Example:

```
public class startoverriding extends Thread{  
  
  public void start()  
  {  
    System.out.println("start method");  
  }  
  public void run()  
  {  
    System.out.println("run");  
  }  
}
```

```
    }  
}  
  
class Threaddemo  
{  
    public static void main(String ar[])  
    {  
        startoverriding t=new startoverriding();  
        t.start();  
    }  
}
```

Output: start method

Case 7:

```
public class startoverriding1 extends Thread{  
  
    public void start()  
    {  
        super.start();  
        System.out.println("start method");  
    }  
    public void run()  
    {
```

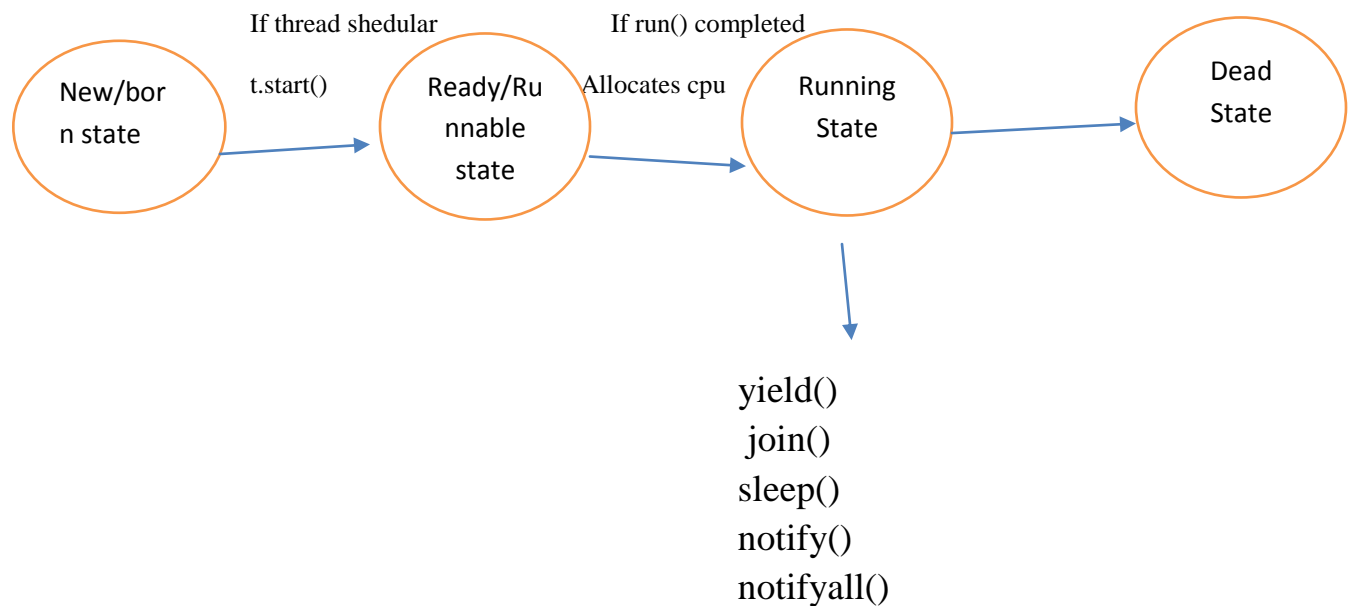
```
        System.out.println("run");
    }
}

class Threaddemo1
{
    public static void main(String ar[])
    {
        Startoverriding1 t=new startoverriding1();
        t.start();
        System.out.println("Main Thread");
    }
}
```

Output:

p-1	p-2	p-3	p-4
Start method	Run	start method	Main Thread
run	start method	Main Thread	Start method
Main Thread	Main Thread	run	run

### Case 8:Life Cycle of Thread:



### Description:

- **New state or born state:**

Once we created a thread object then it is said to be in New state or born state.

- **Ready/Runnable State:**

If we call `start()` then Thread will enter into ready or runnable state.

- **Running State:**

If thread scheduler allocates CPU, then the thread will enter into running state.

- **Dead State:**

If `run()` method completes then thread will be entered into dead state.

### Case 9:

- After starting a thread we are not allowed to restart the same Thread once again otherwise we will get Runtime Exception saying "IllegalThreadStateException".

Example:

```
Thread t=new Thread();
t.start();
-
-
-
t.start()    //R.E.IllegalThreadStateException
```

- With in the run() if we call super.start(),we will get the same Runtime Exception.

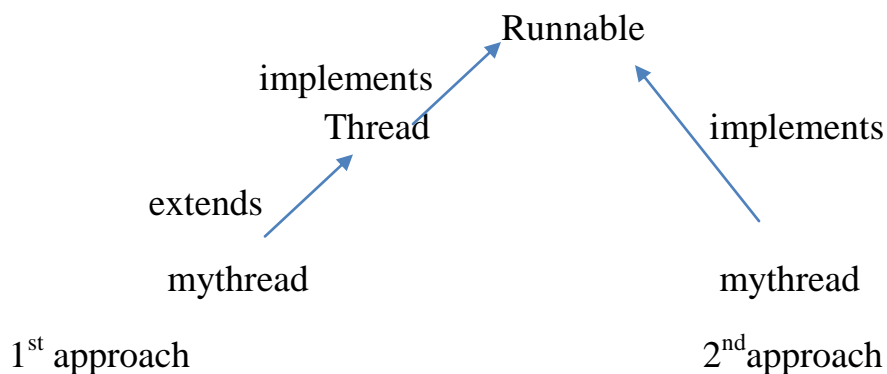
Note:

It's never recommended to override start(),but it is highly recommended to override run().

## Defining Thread by implementing Runnable Interface

We can define a Thread even by implementing Runnable Interface also.

Runnable Interface present in java.lang package and contains only one method run() method.



Example:

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

```
class ThreadDemo1  
{  
    public static void main(String ar[])  
    {  
        RunnableInterface r=new RunnableInterface();  
  
        Thread t2=new Thread(r);  
        t2.start();  
        for(int i=0;i<=10;i++)  
        {  
            System.out.println("main thread");  
        }  
    }  
}
```

```
}  
}
```

**Case Study:**

```
RunnableInterface r=new RunnableInterface();
```

```
Thread t1=new Thread();
```

```
Thread t2=new Thread(r);
```

**Case 1:**

```
1)t1.start();
```

A new Thread will be created which is responsible for execution of Thread class run();

**Case 2:**

```
1)t1.run();
```

No new Thread will be created and Thread class run() method will be executed just like a normal method call.

**Case 3:**

```
1)t2.start();
```

New Thread will be created which is responsible for the execution of RunnableInterface run().

**Case 4:**

```
1)t2.run();
```

New Thread will be created and RunnableInterface run() will be executed just like a normal method call.

**Case 5:**

1)r.start():

We will get compile time error saying start() method is not available in RunnableInterface.

C.E. cannot find symbol

Symbol: method start()

Location: class RunnableInterface

**Case 6:**

1)r.run():

No new Thread will be created and RunnableInterface run() will be executed just like a normal method call.

Qu)In which of the above cases a new thread will be created

t1.start() and t2.start()

**approach to define a Thread:**

- Among the two ways of defining a Thread implements Runnable mechanism is recommended to use.
- In first approach, Our class always extending Thread class and hence There is no chance of extending any other class, but in the second approach we can extend some other class also while implementing Runnable Interface hence second approach is recommended to use.

**➤ Thread class Constructor:**

- 1) Thread t=new Thread();
- 2) Thread t=new Thread(Runnable r);
- 3) Thread t=new Thread(Runnable r,String name);



- 4) Thread t=new Thread(ThreadGroup g ,String name);
- 5) Thread t=new Thread(ThreadGroup g ,Runnable r);
- 6) Thread t=new Thread(ThreadGroup g ,Runnable r,String name);
- 7) Thread t=new Thread(ThreadGroup g ,Runnable r,String name,long stacksize);

## Getting and Setting name of the Thread

- Every Thread in java has some name.It may be provided by the the programmer or default name generated by JVM.
- We can get and set name of a thread by using The following methods of the Thread class.

1) public final String getName();

2)public final String setName(String name);

Example:

```
public class getsetname {  
    public static void main(String arg[])  
    {  
        System.out.println(Thread.currentThread().getName());  
        Thread.currentThread().setName("cern");  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```

Output:

main

cern

Note:

We can get current executing Thread Referance by using the following method of thread class

Public static Thread currentThread()

## Thread priority

- Every thread in java has some priority but the range of thread priorities is “1 to 10”  
(1 is least and 10 is highest)
- Thread class defines the following constants to define some standard priorities.

- 1) Thread MIN-PRIORITY → 1
- 2) Thread NORM-PRIORITY → 5
- 3) Thread NORM-PRIORITY → 10
- 4) Thread LOW-PRIORITY //Invalid
- 5) Thread HIGH-PRIORITY //Invalid

- Thread scheduler will use these priority while allocating cpu
- The thread which is having Highest priority will get chance first.
- If two Threads having same priority then we can't expect exact execution order, it depends on Thread Scheduler.

### Default Priority:

- The default priority only for the main Thread is 5. But for all the remaining Thread it will be Inheritance from the parent that is whenever the priority parent has the same priority will be inheriting to the child.
- Thread class defines the following two methods to get and set priority of the thread.

- 1) Public final int getPriority();
- 2) Public final void setPriority(int p); //The allowed values are 1 to 10, otherwise we

will get

IllegalArgumentException

Example:

```
t.setPriority(5);
```

```
t.setPriority(10);
```

```
t.setPriority(100);//C.E.: IllegalArgumentException
```

Example:

```
public class MyThread1 extends Thread
{
    public void run()
    {
        for(int i=0;i<=10;i++)
        {
            System.out.println("Child class");
        }
    }
}

class demo
{
    public static void main(String args[])
    {
        MyThread1 t=new MyThread1();
        t.setPriority(Thread.MAX_PRIORITY);
        t.start();
        for(int i=0;i<=10;i++)
        {
```

```
        System.out.println("main Thread");
    }
}
}
```

- If we are commenting line(1) then both main & child Thread having The same priority(5) & hence we can't expect exact execution order and exact o/p.
- If we are not commenting line(1) then main Thread has the priority 5 and child Thread has the priority 10 and hence child Thread executed first and main Thread in this case the o/p is

Child Thread

-

-

10 times

Main thread

-

-

10 times

### **The Methods to prevent Thread Execution**

We can't prevent thread from execution by using the following methods

- 1) yield()
- 2) join()

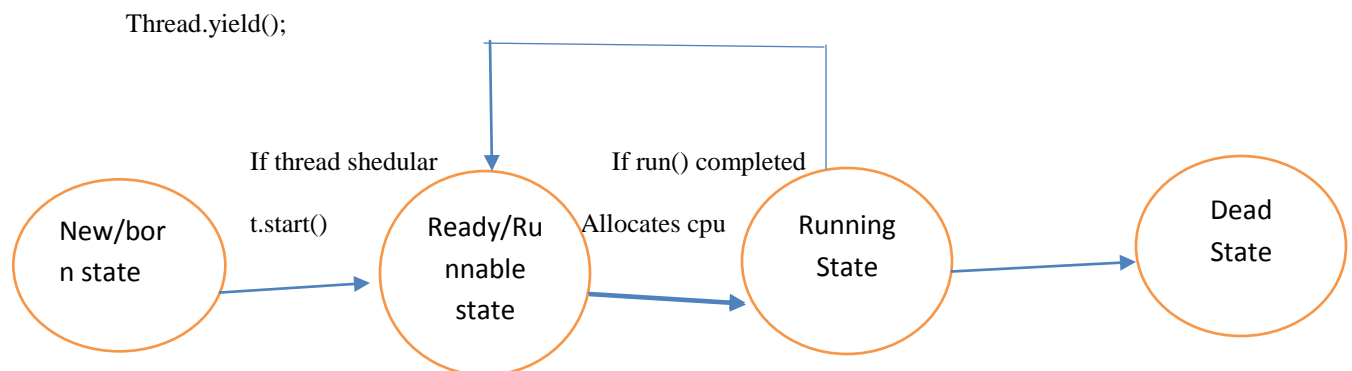
### 3) sleep()

#### 1) yield():-

- yield() method causes to pause current executing Thread for giving the chance to remaining waiting Threads of samePriority
- if there are no waiting Threads or all waiting Threads low priority Then the same Thread will continue its execution once again.

#### ➤ Signature of yield Method

public static void native void yield()



- The Thread which is yielded, when it will get chance one again for execution is decided by ThreadScheduler and we can't expect exactly.

Example:

```
public class MyThread1 extends Thread
{
    public void run()
    {
        for(int i=0;i<=10;i++)
        {
            Thread.yield();           (1) →
            System.out.println("Child Thread");
        }
    }
}

class ThreadYieldDemo
{
    public static void main(String args[])
    {
        MyThread1 t=new MyThread1();
        t.start();
        for(int i=0;i<=10;i++)
        {
            System.out.println("main Thread");
        }
    }
}
```

```
}
```

- If we are commenting Line(1) The both threads will be executed simultaneously and we can't expect exact execution order.
- If we are not connecting line(1) then the chance of completing main method first is high because child thread always calls yield().

## II) join():

- If a Thread want's to wait until completing some other Thread Then we should go for join().

Example:1)

Vienue fixing(t1)	cards printing(t2)	cards distributing(t3)
{	{	{
	t1.join	t1.join
}	}	}

Example2)

```
t1  
{
```

```
    t2.join()  
}
```

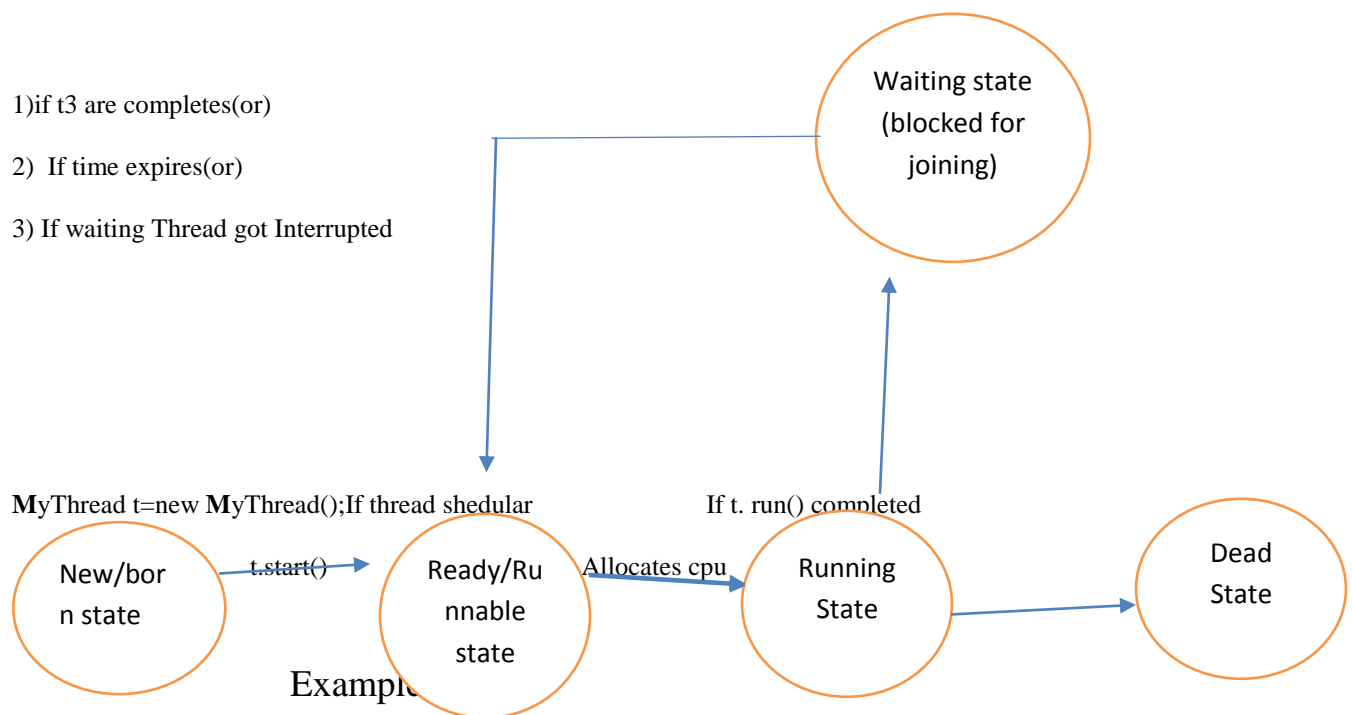
- If thread t1 executes t2.join() then t1 thread will entered into wating state until t2 completes. Once t2 complete then t1 will continue its execution.

i)publicfinal void join()throws InterruptedException

ii) public final void join(long ms)throwsInterruptedException


iii) public final void join(long ms,intns)throws InterruptedException

- `join()` is overloaded and every `join()` throw `InterruptedException`. hence when error we are using `join()` compulsory we should handle `InterruptedException` either by try-catch or by throws otherwise we will get compiletime error.



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



```
        try
        {
            Thread.sleep(20000);
        }
        catch (InterruptedException e)
        {
        }
    }
}
}
}
class ThreadJoinDemo
{
    public static void main(String args[])throws
    InterruptedException
    {
        MyThread t1=new MyThread();
        t1.start();
        t2.start();  (1)
        for(int i=0;i<=10;i++)
        {
            System.out.println("Sujata Thread");
        }
    }
}
```

- If we are commenting line(1) Then both Threads will be executed simultaneously and we can't expect exact execution order. And hence we can't expect exact o/p.
- If we are not commenting line(1) then main thread will wait until completing child Thread. Hence in this case o/p is expected.

**Output:** Priya Threads 10 Times.

Sujata Thread 10 times.

### iii)Sleep():

- If a Thread don't to perform any operation for a particular amount of time (just pauseing) Then we should go for sleep()

i) public static void sleep(long ms)throwsInterruptedException.

ii) public static void sleep(long ms,int ns)throws InterruptedException.

- Whenever we are using sleep() method compulsory we should handle InteruptedException other wise we will get Compiletime Error.

**static:**because sleep method calls thread.sleeps means t.start();  
t.start(); t is object it is instance(or)non-static.

1)if t3 are completes(or)

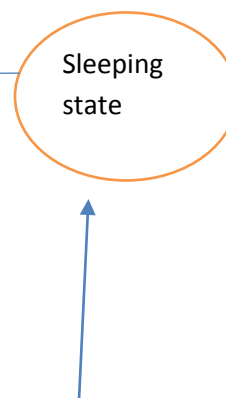
2) If time expired

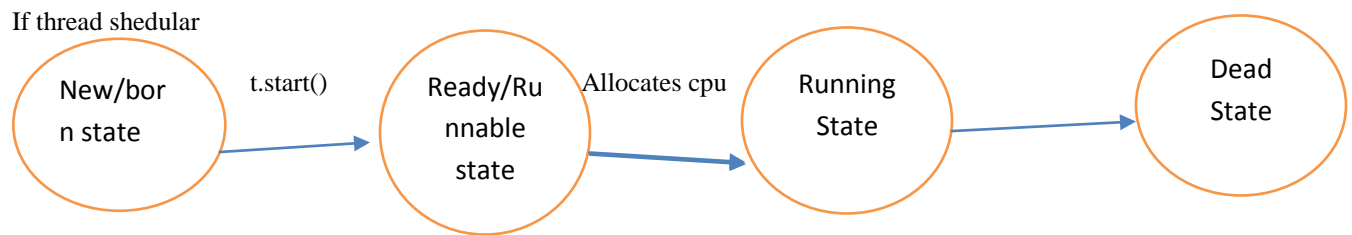
3) If sleep Thread got Interrupted

Thread.sleep(1000);

Thread.sleep(1000,1000);

[help@javat.in](http://help.javat.in)





Example: class test

```
{  
    public static void main(String args[])throws InterruptedException  
    {  
        System.out.println("Shaila");  
        Thread.sleep(5000);  
        System.out.println("Priya");  
        Thread.sleep(5000);  
        System.out.println("Sujata");  
    }  
}
```

## Interruption of Thread

- A Thread Can Interrupt another sleeping or waiting Thread.
- For this Thread class defines interrupt() method.

Public void interrupts()

**Example:**

```
class MyThread extends Thread
{
    public void run()
    {
        try
        {
            for(int i=0;i<=10;i++)
            {
                System.out.println("lazy Thread");
                Thread.sleep(5000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("I Got Interrupted");
        }
    }
}

class InterruptDemo
{
    public static void main(String args[])
    {
        MyThread t=new MyThread ();
        t.start();
        t.interrupt();      (1)  —————→
        System.out.println("End Of main");
    }
}
```

- If we are commenting line(1) Then main wan't Interrupt child hence both Thread will be executed until completion.
- If we are not commenting line(1) then main thread interruptes the child Thread raises Interrupted Exception.
- In this case output

Output:

I am Lazy thread

I got Interrupted

End Of main

Note:

- We may not see the impact of interrupt call immediately.
- When ever we are calling interrupt() method.if the target thread is not in sleeping or waiting state then there is no impact immediately interrupt call will wait until target Thread entered inyo sleeping or waiting state.once trget thread entered into sleep or wating state the interrupt call will impact the target thread.

### Comparision Between **yield()**,**join()**,**sleep()**:-

Property	<b>yield()</b>	<b>join()</b>	<b>sleep()</b>
<b>1)Purpose</b>	To purpose current executing Thread to give the chance for the remaining Threads of same priority.	If a Thread want to wait until completing some other Thread then we should go for join.	If a Thread don't wan't to perform any operation for a particular amount of time (pasuing)go for sleep()
<b>2)Static</b>	Yes	No	Yes
<b>3)It is overloaded</b>	No	yes	Yes
<b>4)It is final</b>	No	Yes	No
<b>5)It is Throw InterruptedException</b>	No	Yes	Yes
<b>6)It is native Method</b>	Yes	No	sleep(long ms)  //native

			sleep(long ms,long ns) //non- native
--	--	--	--

## Synchronization

- Synchronization is the modifier applicable only for methods and blocks and we can't apply for classes and variables.
- If a method or block is declared or synchronized then at a time only one Thread is allowed to execute that method or block on the given object.
- The main advantage of synchronized keyword is we can resolve data inconsistency problem.
- The main limitation of synchronized keyword is it increases waiting time of the Threads and affects performance of the system. Hence if there is no specific requirement it's never recommended to use Synchronized keyword.
- Every Object in Java has a unique lock. Synchronization concept is internally implemented by using this lock concept. Whenever we are using Synchronization then only lock concept will come into picture.
- If a Thread wants to execute any synchronized method on a given object, first it has to get the lock of that object. Once a Thread gets a lock then it is allowed to execute any synchronized method on that object.
- Once Synchronized method completes then automatically the lock will be released.
- While a Thread is executing any Synchronized method on the given object, the remaining Threads are not allowed to execute any synchronized method on the given object simultaneously. But remaining Threads are allowed to execute any non-Synchronized methods. Simultaneously (lock concept is implemented based on object but not based on method).

Example:

```
class display  
{
```

```
public synchronized void wish(String name)
{
    for(int i=0;i<=10;i++)
    {
        System.out.println("Good morning");
        try
        {
            Thread.sleep(30,000);
        }catch(InterruptedException e)
        {
        }
        System.out.println(name);
    }
}

public class synchronizedmethod extends Thread{
    display d;
    String name;

    public synchronizedmethod(display d,String name) {
        this.d=d;
        this.name=name;
    }
    public void run()
    {
        d.wish(name);
    }
}

class synchronization1
{
    public static void main(String[] args) {
        display d=new display();
        synchronizedmethod t1= new synchronizedmethod(d, "dhoni");
```

```
synchronizedmethod t2= new synchronizedmethod(d, "yuvraj");  
t1.start();  
t2.start();  
}  
}
```

Output:Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

dhoni

Good morning

yuvraj

Good morning

yuvraj

Good morning

yuvraj



Good morning  
yuvraj  
Good morning  
yuvraj  
Good morning  
yuvraj  
Good morning  
yuvraj  
Good morning  
yuvraj  
Good morning  
yuvraj  
Good morning  
yuvraj  
Good morning  
Yuvraj

- If we are not declaring wish method as Synchronized then both Thread ill be executed simultaneously and ewe can't expect exact output we will get irregular output.

Output:

Good morning  
dhoni  
Good morning  
yuvraj  
Good morning  
dhoni

- If we declare wish() method as synchronized then Threads will be executed one by one so that we will get regular output.

Good morning  
dhoni

-

-

-10 times

Good morning

yuvraj

-

-

-10 times

Case study:

```
display d1=new display();
```

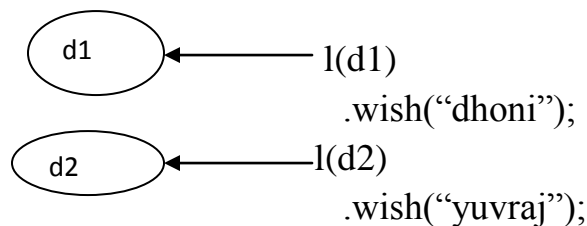
```
display d2=new display();
```

```
synchronizedmethod t1=new synchronizedmethod(d1,"dhoni");
```

```
synchronizedmethod t2=new synchronizedmethod(d2,"yuvraj");
```

```
t1.start();
```

```
t2.start();
```



- Eventhough wish() method is synchronized we will get irregular output in this case because ,The Thrads are operating on different objects.

Reason:

Whenever multiple Threads are operating on same object then only Synchronization play the role.If multiple Threads are operating on multiple objects then there is no impact of synchronization.

### **Classlevel Lock:**

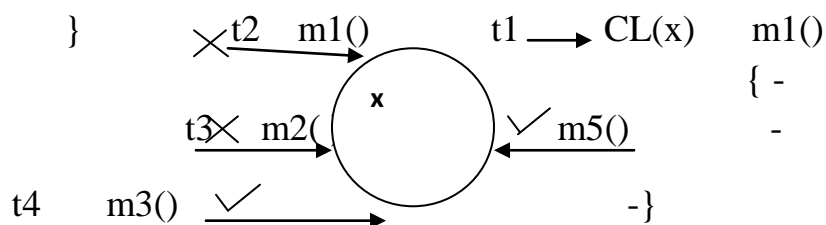
- Every class in java has a unique lock.
- If a Thread want's to execute a static Synchronized method then it required classlevel lock.

- While a Thread executing a static Synchronized method then the remaining threads are allowed to execute the following methods simultaneously

- 1) normal static methods
- 2) normal instance method
- 3) Synchronized instance method

Example:

```
class x
{
    static Synchronized m1();
    static synchronized m2();
    synchronized m3();
    static m4();
    m5();
}
```



Note:

There is no link between object level Lock and class level Lock both are independent of each other.

ClassLevel Lock is different and Object Level Lock is different

### . Synchronized Block:

- If very few lines of code requires Synchronization then it is never recommended to declare entire method as Synchronized, we have to declare those few lines of code inside Synchronized block.

- The main advantage of Synchronized block over Synchronized method is it reduces the waiting time of the threads and improve performance of the system.

Example 1:

We can declare Synchronized block to get current object lock as follows

```
Synchronized(this)
{
-
-
-
-
-
}
```

If Thread got lock of current object then only it is allowed to execute this block.

Example 2:

To get lock of a particular object b we can declare Synchronized block as follows.

```
Synchronized(b)
{
-
-
-
-
-
}
```

- If Thread get lock of 'b' then only it is allowed to execute that block.

Example 3:

Synchronized block concept is applicable only for object and classes but not for primitives otherwise we will get compile time error.

```
int x=10;
Synchronized(x)
```

```

{
-
-
-
}          //C.E.unexpected type
          required:int
          found:reference

```

Every object in java has unique lock, but a thread can acquire more than one lock at a time (of course from different objects)

Example:

Class X

```

{
Syn m1()
{
----
----
Y y=new Y();
y.m2();
--
}
}

```

class y

```

{
Syn m2()
{
---
---
}
}

```

t1 → l(X), l(Y)

```
{  
-  
-  
-  
m1()  
{  
-  —→ Y m2()  
-    {  
-      -  
-    }  
}
```

- The statements present in synchronized method and synchronized block are called as synchronized statement.

## Inter Thread Communication:

- If Threads will communicate with each other by using wait(),notify(),notifyAll() methods.The Thread which requires updation it has to call wait() method.The Thread which is responsible to update it has to call notify() method.
- Wait(),notify,notifyAll() methods are available in Object class but not in Thread class.Beca
- use Thread are required to call method on any shared Object.
- If a Thread wants to call wait(),notify(),notifyAll() methods compulsory the Thread should be owner of the object.i.e. The Thread has to get lock of that object.i.e.The Thread should be in the Synchronized area.
- Hence,we can call wait(),notify(),notifyAll() methods only perform synchronized area otherwise we will get runtime exception saying “IllegalMonitorStateException”.
- If a Thread class wait(), method it releases the lock immediately and entered into waiting state.Thread releases the lock of only current object but not all locks.After calling notify() and notifyAll() methods Thread releases the lock but may not

immediately. Except these wait(), notify(), notifyAll() methods there is no other case where thread releases the lock.

method	Is thread releases lock
yield()	No
join()	No
sleep()	No
wait()	Yes
notify()	Yes
notifyAll()	Yes

### **notify() Vs.notifyAll()**

- We can use notify() to notify only one waiting Thread. but which waiting thread notify we can't expect exactly all remaining threads have to for the further notification.
- But in the case of notifyAll() all waiting threads will be notifying but the threads will be executed one by one.

Note:

On which object we are calling wait(), notify(), notifyAll() we have to get the lock of that object.

## **DeadLock**

- If two Threads are waiting for each other forever such type of situation is called "DeadLock".
- There are no resolution technique for DeadLock, but several prevention techniques are possible.

Example:

```
class a
{
    Public Synchronized void foo(B b)
    {
```

```
        System.out.println("Thread1 starts execution foo");
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {

        }
        System.out.println("Thread1 trying to catch b's last()");
        b.last();
    }
    public Synchronized void last()
    {
        System.out.println("Inside A this is last()");

    }

}

class c
{

    public Synchronized void bar(A a)
    {
        System.out.println("Thread2 starts execution bar");
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException e)
        {
        }
        System.out.println("Thread2 trying to call a's last()");
    }
}
```



```
        a.last();
    }
    public Synchronized void last()
    {
        System.out.println("Inside B this is last");

    }

}

class DeadLock extends Thread
{
    A a=new A();
    B b=new B();
    DeadLock()
    {
        this.start();
        a.foo(b);           //executed by main thread
    }
    public void run()
    {
        b.bar(a) ;          //executed by child thread
    }
    public static void main(String ar[])
    {
        New DeadLock();
    }
}
```

```
}
```

```
}
```

Output:

Thread1 starts execution of foo

Thread2 starts execution of bar

Thread1 trying to call b's last()

Thread2 trying to call a's last()

-

-

-

- Synchronized keyword is the only one reason for DeadLock hence while using Synchronized keyword we have to take very much care .

### **DeadLock Vs. Starvation:**

- In the case of DeadLock waiting never ends.
- A long waiting of as Threads which ends at certain point of time is called "Starvation".

Example:

Least priority Thread has to wait until completing all the Threads but this long waiting should compulsory ends at certain point of time.

- Hence a long waiting which never ends is called "DeadLock".Where as long waiting ends of certain points of time is called "Starvation".

## **Daemon Threads**

- The Threads which are executing in the background are called 'Daemon Threads'

### Example: Garbage Collection

- The main objective of daemon Threads is to provide support for 'non-Daemon Threads'
- We can check whether the Thread is Daemon or not by using "isDaemon() method"

```
public final Boolean isDaemon()
```

- We can change Daemon nature of a Thread by using setDaemon() method

```
public final void setDaemon(boolean b)
```

- We can change Daemon nature of a Thread before starting only. If we are trying to change after starting a thread we will get runtime exception saying "IllegalThreadStateException".
- Main Thread is always non-daemon & it's not possible to change its daemon nature.

### Default Nature

- By default main Thread is always non-daemon but for all the remaining Threads daemon nature will be inheriting from parent to child. That is if the parent is daemon, child is also Daemon & if the parent is non-Daemon then child is also non-Daemon.
- Whenever the last non-Daemon Thread terminates all the Daemon Thread will be Terminated automatically.

### Example:

```
class MyThread extends Thread
{
    public void run()
```

```
{
    for(int i=0;i<10;i++)
    {
        System.out.println("lazy Thread");
    }
    try
    {
        Thread.sleep(5000);
    }
    catch(InterruptedException e)
    {
    }
}

class Test
{
    public static void main(String args[])
    {
        MyThread t=new MyThread ();
        t.setDaemon(true); ———→ (1)
        t.start();
        System.out.println("End Of main");

    }
}
```

- If we are commenting line(1) Then both main & child Thread are non-Daemon & hence both will be executed until there completion.
- If we are not commenting line(1) the main thread is non-daemon & child Thread is Daemon. Hence when ever main Thread terminates automatically child Thread will be terminated.

➤ **How to kill Thread**

- A Thread can stop or kill any other Thread by using stop() method then automatically running Thread will enter Thread into DeadState. It is a deprecated method & hence not recommended to use.

```
public void stop();
```

➤ **Suspending & reusing a Thread:-**

- A Thread can suspend another Thread by using suspend() method.
- A Thread can resume a suspended Thread by using resume() method.
- But these methods are deprecated methods & hence not recommended to use.

## Threading Phases

