Multithreading

09/04/2011

① Introduction

② The ways to define instantiate, and start a thread

③ Geting & Setting name of a thread

**④ thread priorities

⑤ the methods to prevent thread execution

- → yield()
- → join()
- → sleep()

* ⑥ Synchronization

⑦ Inter thread Communication

- → wait()
- → notify()
- → notifyAll()

⑧ Deadlock

⑨ Daemon threads

Multitasking:-

→ Executing Several tasks Simultaneously is Called "multitasking".
There are 2 types of multitasking.

(1) process-based multitasking.
(2) thread-based multitasking.

Ex:- Students in Class Room.

(1) process-based multi-tasking:-

→ Executing Several tasks Simultaneously, where each task is a Seperate independent process, is Called process based multitasking.

Ex:- While typing a Java program in editor we Can able to listen audio Songs by MP3 player in the System. at the Same time we Can download a file from the net. all these tasks are Executing Simultaneously & 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 tasks Simultaneously where each task is a Seperate independent part of the Same program is Called "thread based multitasking" & each independent part is Called "thread".

→ It is Best Suitable for "programatic Level".

→ wheather 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 areaas of multithreading are developing video games, multimedia Graphics, implementing animations,....

→ Java provides 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 programs is very easy ~~when~~ in Java when Compared with C++.

(2) The Ways to define, Instanciate & Start a new thread :-

→ we Can define a thread in the following 2 ways.

(i) By extanding Thread class.

(ii) By implementing Runnable Interface.

Defining a thread by extanding Thread class :-

defining a Thread by Extanding Thread class:-

Ex:-

defining a thread

```
Class Mythread extends Thread
{
    public void run()
    {
        for(int i=0; i<=10; i++)
        {
            S.o.pln(" child thread");
        }
    }
}
```

Executing child thread (mythread)

Job of thread

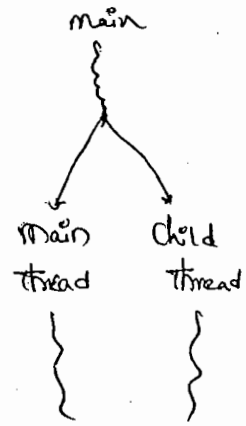
```
Class threadDemo
{
    p.s.v.m(String[] args)
    {
        MyThread t = new Mythread(); // instantiation of thread
        t.start(); // Starting of a thread
        for(int i=0; i<=10; i++)
        {
            S.o.pln(" main thread");
        }
    }
}
```

main thread

child thread

Two threads

executing main thread



Case 1:-

Thread Schedular :-

→ when ever multiple threads are waiting to get chance for execution which thread will get chance first is decided by thread schedular whose behaviour is JVM vendor dependent. Hence we can't expect exact execution order & hence exact o/p.

→ thread schedular is the part of JVM. due to this unpredictable behaviour of thread schedular we can't expect exact o/p for the above program. the following are various possible o/p.

P-1	P-2	P-3	P-4
main thread	child thread	child thread	main thread
=	=	main thread	main "
=	=	main thread	child "
=	=	=	child "
child thread	main thread	child thread	main thread
=	=	=	=
=	=	=	
=	=	main thread	
=	=	=	

Note:-

→ when ever the situation comes to multithreading the guarantee in behaviour is very less. we can tell posible o/p but not exact o/p.

Case 2:-

difference b/w t.start() & t.run() :-

→ In the case of t.start() a new thread will be created & that thread is responsible to execute run().

→ But in the case of t.run() no new thread will be created & 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 o/p.

o/p:-

```
child thread
child thread
8 times
main thread
10 times
```

entire o/p produced by only main thread.

Case 3:-

Importance of Thread class start() method:

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

Eg:-

```
class thread
{
    start()
    {
      1. Register this thread with thread scheduler & perform other
                                                initialization activities
      2. run()
    }
}
```

Case 4:-

* If we are not overriding run() method :-

→ if we are not overriding run() method, then thread class run() will be executed which has Empty implementation & Hence we won't get any o/p.

ex:-

```
Class Mythread extends Thread
{
}
class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    Mythread t = new Mythread();
    t.start();
  }
}
```

O/P:- no o/p printing

Note:

* It is highly recommended to override run() to define our Job.

Case 5:-

Overloading of run():-

→ Overloading of the run() is possible, but thread class Start() will always call no arguement run() only. but the other run() we have to call Explicitly Just like a normal method call.

Ex:-

```
Class MyThread extends Thread
{
    Public void run()
    {
     S.o.pln("run()");
    }
    public void run(int i)
    {
     S.o.pln("run(int i)");
    }
}

Class ThreadDemo
{
 p.s.v.m (String[] args)
 {
  MyThread t = new MyThread();
  t.start();
 }
}
```

O/P:- run()

Case 6:-

Overriding of Start():-

→ If we override Start() then Start() will be executed just like a normal method call & no new thread will be created.

Ex:-

```
Class MyThread extends Thread
{
    Public void Start()
    {
```

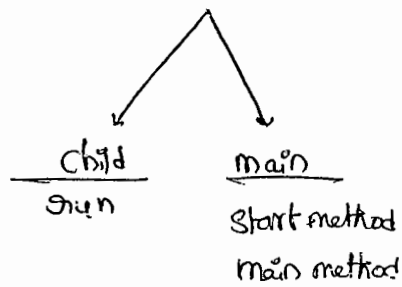
```
        S.o.pln("Start method");
        }
        public void run()
        {
          S.o.pln("run");
        }
      }
      Class ThreadDemo
      {
        p.s.v.m(String[] args)
        {
          MyThread t = new MyThread();
          t.start();
        }
      }
```

O/P:- Start method.

Case(7):-

```
      class MyThread extends Thread
      {
        public void start()
        {
          Super.start();
          S.o.pln("Start method");
        }
        public void run()
        {
          S.o.pln("run");
        }
      }
```

```
Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    Mythread t = new Mythread();
    t.Start();
    S.o.pln("main method");
  }
}
```

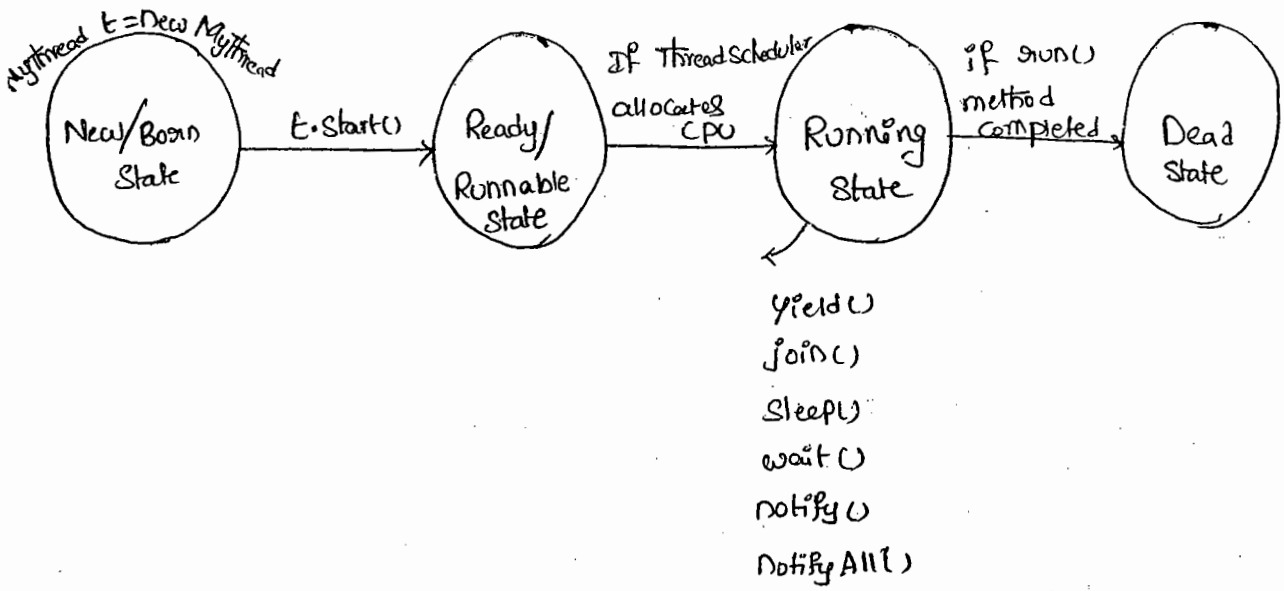


O/P:-

P-1	P-2	P-3	P-4
Start method	run	Start method	main method
run	Start method	Main method	Start method
Main method	main method	run	run

Case-8:-

* **Life Cycle of a Thread:-**



→ Once we Created a thread Object then it is Said to be in New State or born State.

→ If we Call Start() method then the thread will be entered into ready (or) Runnable State.

→ If ThreadScheduler allocates cpu, then the thread will entered into Running State.

→ If run() method Completes then the thread will 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 IllegalRuntimeException Saying "IllegalThreadStateException".

eg:

```
Thread t = new Thread()
    t.start();
    :
    t.start(); ✗   R.E:- IllegalThreadStateException.(ITSE)
```

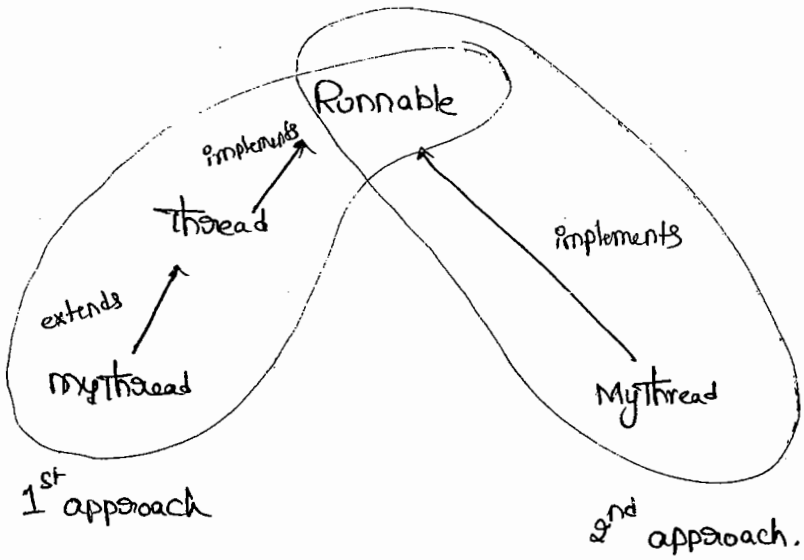
→ with in the run() if we Call Super.start() we will get the Same Run-time Exception.

Note:-

→ It's never recommended to Override Start(), but it is highly recommended to Override run().

(2) defining a Thread by implementing "Runnable Interface":-

* We can define a thread even by implementing Runnable Interface also.
* Runnable Interface present in Java.lang package & Contains only one method run() method.




```
Runnable
{
  run()
  {
  }
}

class Thread
{
  start()
  {
    run()
  }
}
```

Ex:-

```
Class MyRunnable implements Runnable
{
  public void run()
  {
    for (int i=0 ; i<10 ; i++)
    {
      S.o.pln("child Thread");
    }
  }
}
```

defining a thread ~~as~~ ~~Great~~

Job of thread

```
Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    MyRunnable r = new MyRunnable();
    Thread t = new Thread(r);
                 └──→ target Runnable
    t.start();
    for(int i=0; i<=10; i++)
    {
      S.o.pln("main thread");
    }
  }
}
```

→ We can't get Exact o/p & we will get mixing o/p

Case Study :-

```
MyRunnable r = new MyRunnable();
Thread t1 = new Thread();
Thread t2 = new Thread(r);
```

Case(1) :-

(i) t_1.start() :-

→ A new thread will be Created which is responsible for Execution of Thread class run().

Case(2) :- t_1.run() :-

→ No new thread will be Created & Thread class run() will be Executed Just Like a normal method call.

Case 3:- t_2.Start():-

→ New thread will be Created which is responsible for the Execution of MyRunnable run() method.

Case 4:- t_2.run():-

→ No new thread will be Created & MyRunnable run() will be Executed just like a normal method Call.

Case 5:- r.Start():-

→ We will get Compile-time Error Saying Start() is not available in MyRunnable class

C.E:- Cannot find Symbol

Symbol : method Start()

location : class MyRunnable

Case 6: r.run():-

→ No new thread will be Created & MyRunnable run() will be Executed Just like a normal method Call.

Q). In which of the above cases a new thread will be Created

A) t_1.Start() & t_2.Start()

Q) In which of the above Cases MyRunnable class run() will be executed Just like a normal method?

t_2.run() & r.run()

Best Approach to define a Thread :-

→ Among the two ways of defining a thread implements Runnable mechanism is recommended to use.

→ In the first approach, ~~thread~~ our class always extending Thread class & 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 2nd approach is Recommended to use.

Thread Class Constructors :-

(1) Thread t = new Thread();

(2) Thread t = new Thread(Runnable r);

(3) Thread t = new Thread(String name);

(4) Thread t = new Thread(Runnable r, String name);

(5) Thread t = new Thread(ThreadGroup g, String name);

(6) Thread t = new Thread(ThreadGroup g, Runnable r);

(7) Thread t = new Thread(ThreadGroup g, Runnable r, String name);

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

Durga's approach to define a Thread (not recommended to use):

ex:-

```
Class MyThread extends Thread
{
  public void run()
  {
    S.o.pln("run method");
  }
}
Class Test
{
  p.s.v.m(String[] args)
  {
    MyThread t = new MyThread();
    Thread t1 = new Thread(t);
    t1.Start();
    S.o.pln("main");
  }
}
```

o/p:-

run	main
main	run

3) Getting & Setting name of a thread :-

→ Every thread in Java has some name. It may be provided by the programmer or default name generated by JVM.

→ We can get & set name of a thread by using the following methods of thread class.

(i) public final String getName();

(ii) public final void setName(String name);

Ex:-

```
class Test
{
  p.s.v.m(String[] args)
  {
    S.o.pln(Thread.currentThread().getName());   // main
    Thread.currentThread().setName("prabas");
    S.o.pln(Thread.currentThread().getName());   // prabas.
  }
}
```

Note:-

→ We can get current executing thread reference by using the following method of thread class.

public static thread currentThread();

4) Thread priorities:-

→ Every thread in Java has some priority but the range of thread priorities is "1 to 10". (1 is least & 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.MAX_PRIORITY → 10
4) ~~Thread.LOW_PRIORITY~~
5) ~~Thread.HIGH_PRIORITY~~

→ Thread Schedular will use these priorities 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 schedular.

Default priority:-

→ The default priority only for the main thread is 5. But for all the remaining threads it will be inheriting from the parent. i.e whatever the priority parent has the same priority will be inheriting to the child.

* Thread class defines the following 2 methods to get & set priority of a 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.

Ex:-

```
t.setPriority(5);
t.setPriority(10);
t.setPriority(100); X  R.E:- IAE (IllegalArgumentException).
```

Ex:-

```
class Mythread extends thread
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      S.o.pln("Child thread");
    }
  }
}

class ThreadPriorityDemo
{
  p.s.v.m(String[] args)
  {
    Mythread t = new Mythread();
    // t.setPriority(10);  ---> ①
    t.start();
    for(int i=0; i<=10; i++)
    {
      S.o.pln("main method");
    }
  }
}
```

→ If we are Commenting line① then Both main & child threads having the Same priority (5) & Hence we Can't Expect Exact Execution order and Exact o/p.

→ If we aren't Commenting line① then main thread has the priority 5 & child thread has the priority 10 & Hence child thread will be Executed first & then 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 prevent a thread from Execution by using the following methods.

(i) yield()

(ii) join()

(iii) Sleep()

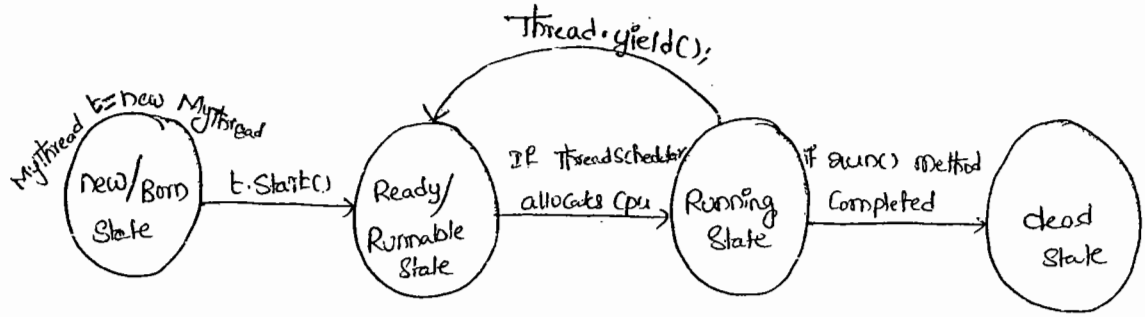
(i) yield() :-

→ yield() method Causes, to pause Current Executing thread for giving the chance to remaining waiting threads of Same priority.

→ If there are no waiting threads or all waiting threads have low priority then the Same thread will Continue it's execution once again.

→ Signature of yieldmethod

```
public static native void yield()
```



→ The thread which is yielded, when it will get chance once again for Execution is decided by Threadschedular. & we Can't Expect Exactly.

Ex:-

```
class Mythread extends Thread
{
    public void run()
    {
        for (int i=0; i<10; i++)
        {
            Thread.yield();  ---------> ①
            S.o.pln("child thread");
        }
    }
}
class ThreadYieldDemo
{
    p.s.v.m(String[] args)
    {
        Mythread t = new Mythread();
        t.Start();
        for (int i=0; i<10; i++)
        {
            S.o.pln("main thread");
        }
    }
}
```

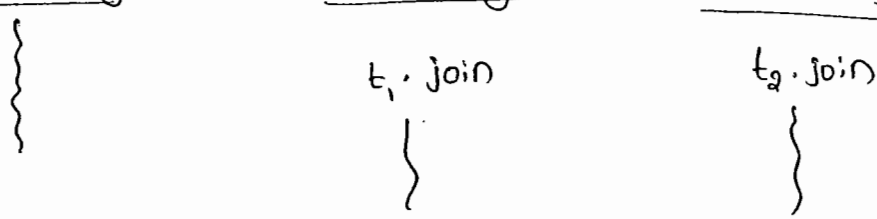
→ If we are Commenting Line① the both threads will be Executed Simultaneously & we Can't Expect Exact Execution Order.

→ If we are not Commenting Line① then the chance of Completing main thread first is high because child thread always calls yield().

ii) join() :-

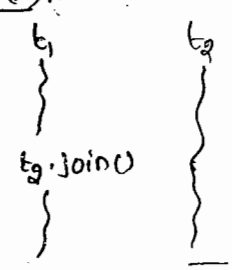
→ If a thread wants to wait until Completing Some other thread then we should go for join() method.

Ex:- (1) Venue firing (t_1) Cards printing (t_2) Cards distributing (t_3)



→ If thread t_1 Executes t_2.join() then t_1 thread will entered into waiting state until t_2 Completes. Once t_2 Completes then t_1 will Continue its execution.

ex(2):-

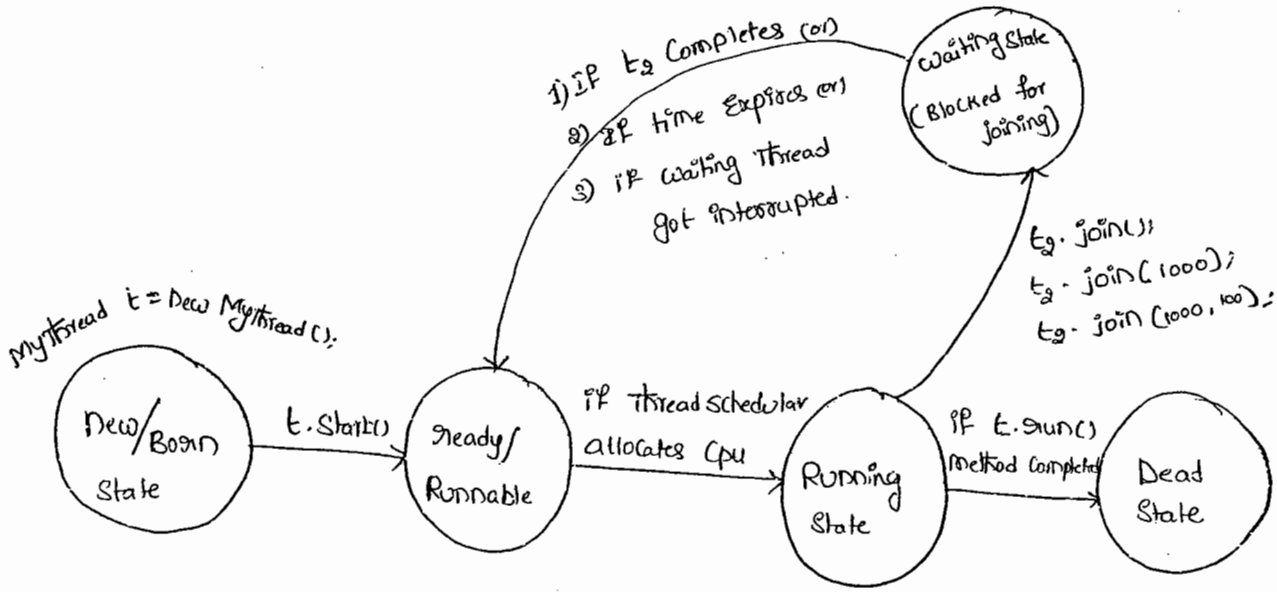


(i) public final void join() throws InterruptedException

(ii) public final void join(long ms) throws InterruptedException

(iii) public final void join (long ms, int ns) throws InterruptedException.

→ join() method is Overloaded and every join() throws InterruptedException. Hence, when ever we are using join() Complusary we should handle InterruptedException, either by try-catch or by throws other wise we will get Compiletime Error.




```
Class Mythread extends Thread
{
  Public void run()
  {
    for(int i=0 ; i<10 ; i++)
    {
      S.o.pln(" Sitka Thread");
      try
      {
        Thread.Sleep(2000);
      }
      Catch(IE e)
      {
      }
    }
  }
Class ThreadJoinDemo
{
  p.s.v.m(String[] args) throws InterruptedException
  {
    Mythread t1 = new mythread();
    t1.Start();
    t1.join(); ———→ ①
```

```
for(int i=0 ; i<10 ; i++)
{
 S.o.pln("Rama thread");
}
}}}
```

→ If we are Commenting Line① 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 Line① then main thread will wait until Completing child thread. Hence in this case the o/p is Expected.

O/p :- Sitathread 10 times
Ramathread 10 times

(iii) Sleep() :-

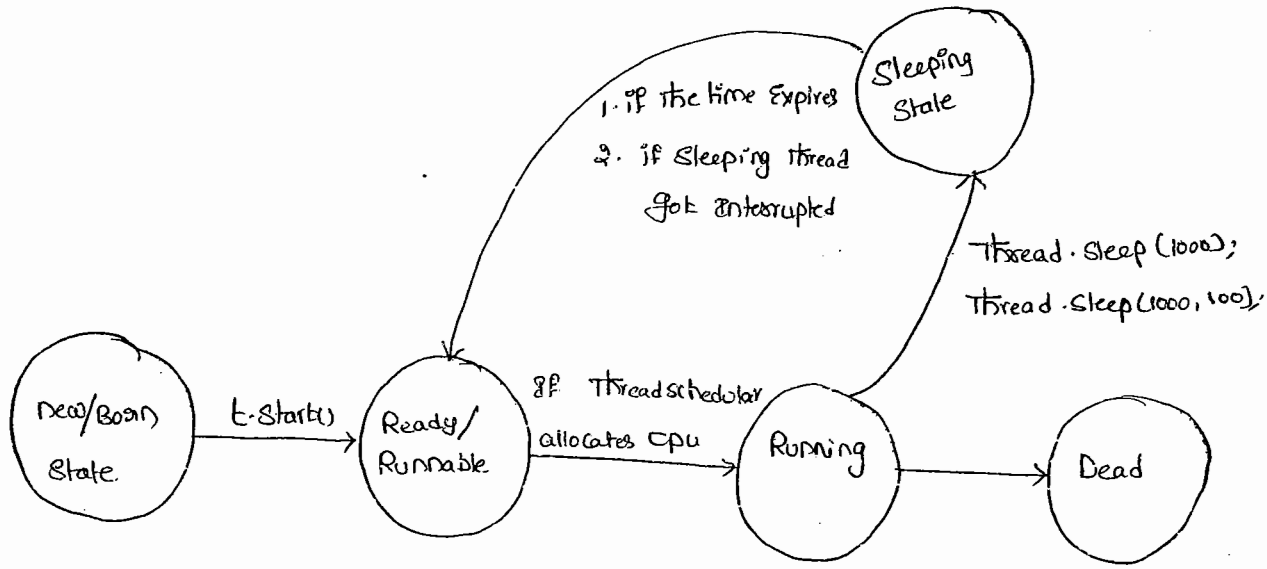
→ If a thread don't want to perform any operation for a particular amount of time (Just pausing) then we should go for Sleep()

1) Public Static void Sleep(long ms) throws InterruptedException

2) public Static void Sleep(long ms, int ns) throws InterruptedException

→ when ever we are using Sleep() method Compulsary we should handle Interrupted Exception other wise we will get Compile time Error

Static :- because sleep method Calls thread.Sleep() means Class name
t1.start(); t1 is object So it is instance (or) Non static



Ex:-

```
Class Test
{
  p.s.v.m(String[] args) throws InterruptedException
  {
    S.o.pln("Durga");
    Thread.sleep(5000);
    S.o.pln("Software");
    Thread.sleep(5000);
    S.o.pln("Solutions");
  }
}
```

Interruption of a Thread :-

* A Thread Can Interrupt another Sleeping or waiting Thread.
* for this Thread class defines interrupte() method.

```
public void interrupte()
```

Ex:-

```
Class MyThread extends Thread
{
  Public void run()
  {
   try
    {
    for (int i=0 ; i<100; i++)
    {
     S.o.pln(" Lazy Thread");
     Thread . sleep (5000);
    }
   }
  Catch (IE e)
   {
    S.o.pln(" I got Interrupted");
   }
}
Class InterruptDemo
{
  P.S.v.m (String[] args)
  {
    MyThread t = new MyThread();
      t.Start();
  →   t . interrupt();  ——→ ①
      S.o.pln(" end of main");
  }
}
```

→ If we are Commenting line ① then main thread won't Interrupte Child thread Hence both threads will be executed untill Completion

→ If we are not Commenting line① then main thread interruptes the Child thread ~~hence child thread wont Conti~~ raises Interrupted Exception.

→ In this case the o/p is

o/p:- I am Lazy thread

I got Interrupted

End of main

**Note:-

*→ We ~~can't~~ may'nt 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 untill target thread entered into Sleeping or waiting State. once target thread entered into Sleeping (or) waiting state the interrupt call will impact the target thread.

* Comparision table for yield(), join(), Sleep() :-

Property	yield()	join()	Sleep()
1) Purpose ?	to pause 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 want to perform any operation for a perticular amount of time (pausing) go for sleep()
2) Static	yes	No	yes
3) Is it over-loaded	No	yes	yes
4) Is it final	No	yes	No
5) Is it throws InterruptedException	No	yes	yes
6) Is it native method	yes	No	Sleep(long ms) → native Sleep(long ms, int ns) → non-native

Synchronization :-

→ Synchronized is the modifier applicable only for methods & blocks & we can't apply for classes & variables.

→ If a method or block declared as 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 key-word is we can resolve data inconsistency problem.

→ The main limitation of Synchronized keyword is it increases waiting time of the threads & effects performance of the system. Hence if there is no specific requirement it's never recommended to use Synchronized key-word.

→ Every object in Java has a unique lock Synchronization concept internally implemented by using this Lock concept. when ever we are using Synchronization then only Lock concept will come into the picture.

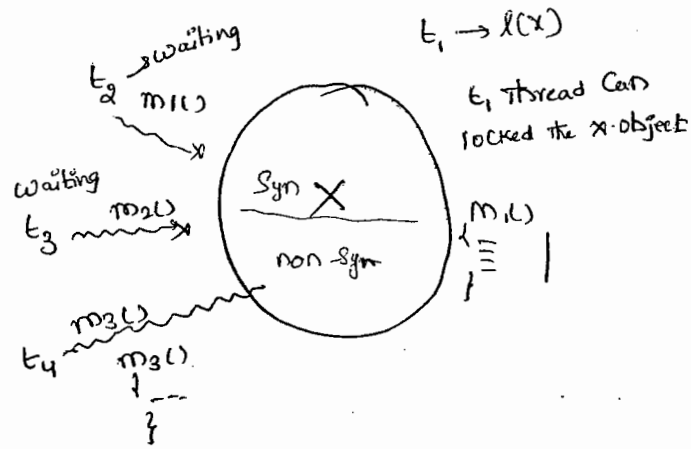
→ If a thread wants to execute any Synchronized method on the given object, first it has to get the lock of that object. Once a thread get a lock then it allowed to execute any Synchronized method on that object.

→ Once Synchronized method completes then automatically the lock will be released.

↳ While a thread Executing any Synchronized method on the given object the remaining threads are not allowed to Execute any Synchronized method on the given Object ~~Symmult~~ 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).

Ex:-

```
Class X
{
    Sync m1()
    {
    }
    Sync m2()
    {
    }
    m3()
    {
    }
}
```



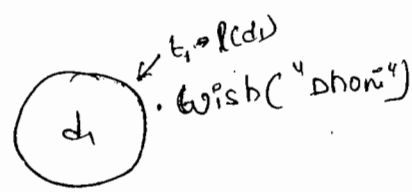
Lock:- It is an Object associated with for every Object

Ex:-

```
Class Display
{
  Public  void wish(String name)   → Synchronized
  {
    for (int i=0 ; i<10 ; i++)
    {
      S.o.print(" Good morning: ");
      try
      {
        Thread.sleep(3000);
      }
      Catche(IE e) { }
```

```
        S.O.pln(name);
    }
}
}

Class MyThread extends Thread
{
    Display d;
    String name;
    MyThread(Display d, String name)
    {
        this.d = d;
        this.name = name;
    }
    public void run()
    {
        d.wish(name);
    }
}
Class SynchronizedDemo
{
    p.s.v.m(String[] args)
    {
        Display d1 = new Display();
        MyThread t1 = new MyThread(d1, "Dhoni");
        MyThread t2 = new MyThread(d2, "Yuvaraj");
        t1.Start();
        t2.Start();
    }
}
```



→ If we are not declaring wish() method as Synchronized then both Threads will be Executed Simultaneously & we Can't Expect Exact o/p we will get irregular o/p.

o/p:-

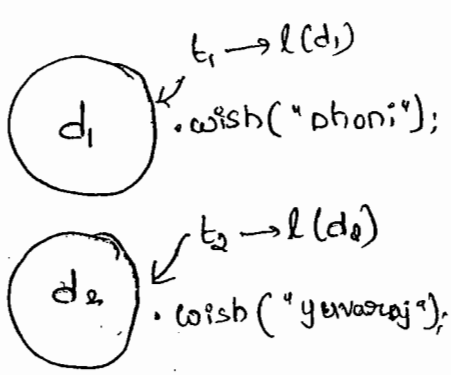
Goodmorning : Goodmorning : Dhoni
Goodmorning : yuvaraj
" : Dhoni
" "

→ If we declare wish() method as Synchronized then threads will be Executed One by one So that we will get regular o/p.

o/p:- Goodmorning : Dhoni
⋮ 10times
Goodmorning : yuvaraj
⋮ 10-times

Case Study:-

```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "yuvaraj");
t1.Start();
t2.Start();
```



→ Eventhough wish() method is Synchronized we will get irregular O/P in this case. Because, the threads 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 wants to Execute a Static Synchronized method then it required classlevel lock.

→ while a thread executing a Static Synchronized method then the remaining threads are not allowed to execute any Static Synchronized method of that class simultaneously but remaining threads are allowed to execute the following methods simultaneously.

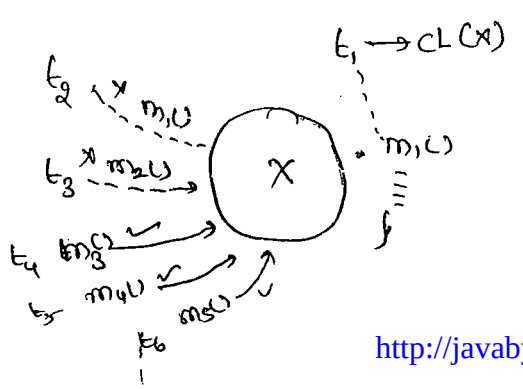
✓ 1. Normal Static methods.

✓ 2. Normal instance methods.

✓ 3. Synchronized instance methods.

Ex:-

```
Class X
{
  Static Syn m1()
  Static Syn m2()
     Syn m3()
     Static m4()
     m5()
}
```



Note :-

→ There is no link between object Level lock & class Level Lock both are independent of Each other.

→ ClassLevel lock is different & ObjectLevel 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 & improves performance of the System.

Ex(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.

Ex(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 blocks.

Ex(3):-

→ To Get class level lock we can declare Synchronized block as follows.

```
Synchronized(classname.class)
{
  --
  --
}
```

→ if thread got Classlevel lock of classname (ex Display) class then only it is allowed to Execute that block.

Ex(4):-

Synchronized block concept is applicable only for Objects & classes but not for primitives otherwise we will get Compiletime Error.

```
int x=10;
Synchronized(x)
{
  --
  --
}
```

C.E:- unexpected type

found: int

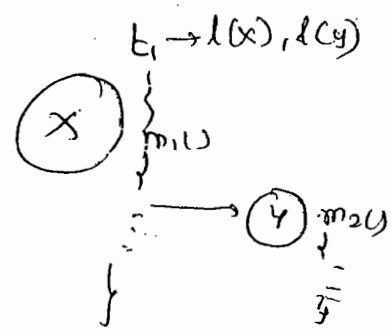
required: reference

→ Every object in java has a unique lock, But a thread can acquire more than one lock at a time (ofcourse from diff. objects)

Exp:

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

Class Y
{
Syn m2()
{
  --
}
}
```



FAQ:-

① Explain about Synchronized Keyword & what are various Advantages & disAdvantages?

② what is Object lock & when it is required?

③ While a thread Executing an instance Synchronized method on the given Object then is it possible to Execute any other Synchronized method Simultaneously by other threads? Ans:- Not possible

④ what is ClassLevel Lock & when it is required.

⑤ What is the diff. b/w Object lock & class Level lock

⑥ what is the Advantage of Synchronized block over Synchronized method

⑦ How to declare Synchronized block to get class level lock?

⑧ What is Synchronized Statement? (Interview people Created terminology)

→ The Statements present in Synchronized method & Synchronized block are called as Synchronized Statement.

30/04/11

Inter Thread Communication :-

↳ Two 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. because threads are required to call these method on any shared object.

* If a thread wants to call wait(), notify(), & notifyAll() methods compulsary 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 from Synchronized area otherwise we will get runtime Exception saying "IllegalMonitorStateException".

→ If a thread calls wait() method it releases the lock immediatly and entered into waiting state. A thread releases the locks of only current object but not all locks. after calling notify() and notifyAll() methods thread releases the lock but may not immediatly. Except these wait(), notify(), notifyAll() there is no other case where thread releases the lock.

javap java.lang.object

method	is thread releases lock?
yield()	No
join()	No
Sleep()	No
wait()	yes
notify()	yes
notifyAll()	yes

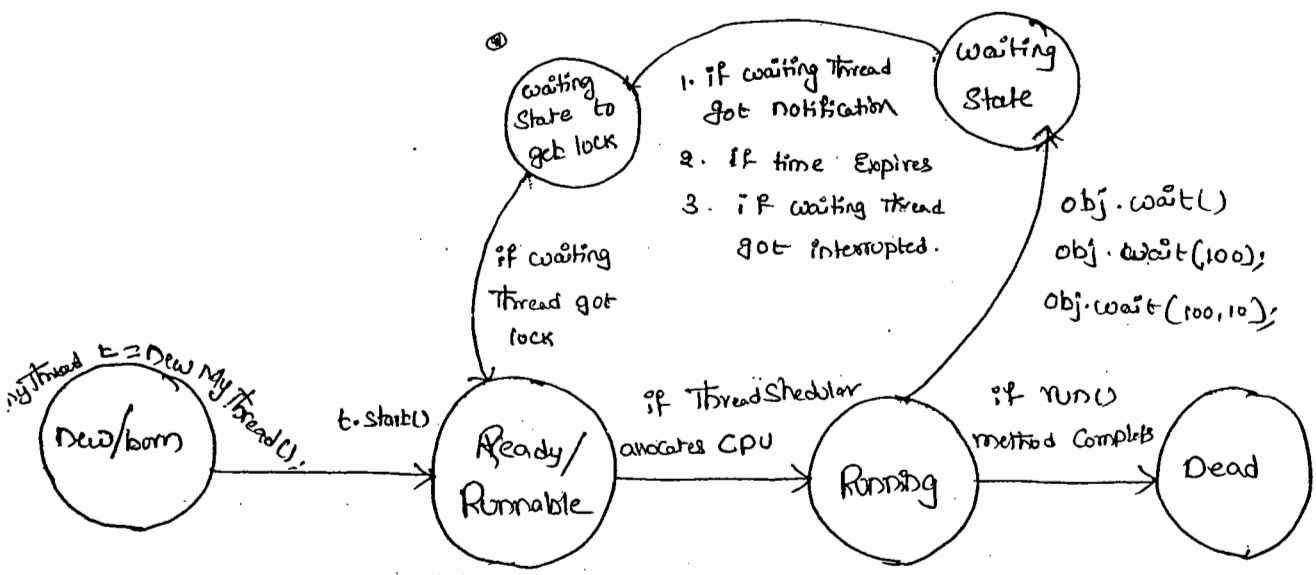
1) public final void wait() throws IE

2) public final native void wait(long ms) throws IE

3) public final void wait(long ms, int ns) throws IE

4) public final native void notify()

5). public final native void notifyAll()



Ex:-

```
Class ThreadA
{
 P.S.V.m(String[] args) throws InterruptedException
 {
   ThreadB b = new ThreadB();
   b.start();
   Synchronized (b)  ---------> Thread.sleep(1000);
   {
 ① S.o.pln("main Thread trying to Call wait()");
   b.wait();   // b.wait(1000);
 ④ S.o.pln("main thread got notification").
 ⑤ S.o.pln(b.total);
   }
 }

Class ThreadB extends ThreadA
{
  int total=0;
  Public void run()
  {
   Synchronized (this)
   {
 ② S.o.pln("child Thread strats notification");
   for(int i=1 ; i<=100 ; i++)
   {
     total = total + i;
   }
 ③ S.o.pln("child thread trying to given notification");
   this.notify();
}}}
```

O/P:- main Thread Calling wait method

Child thread starts Calculation

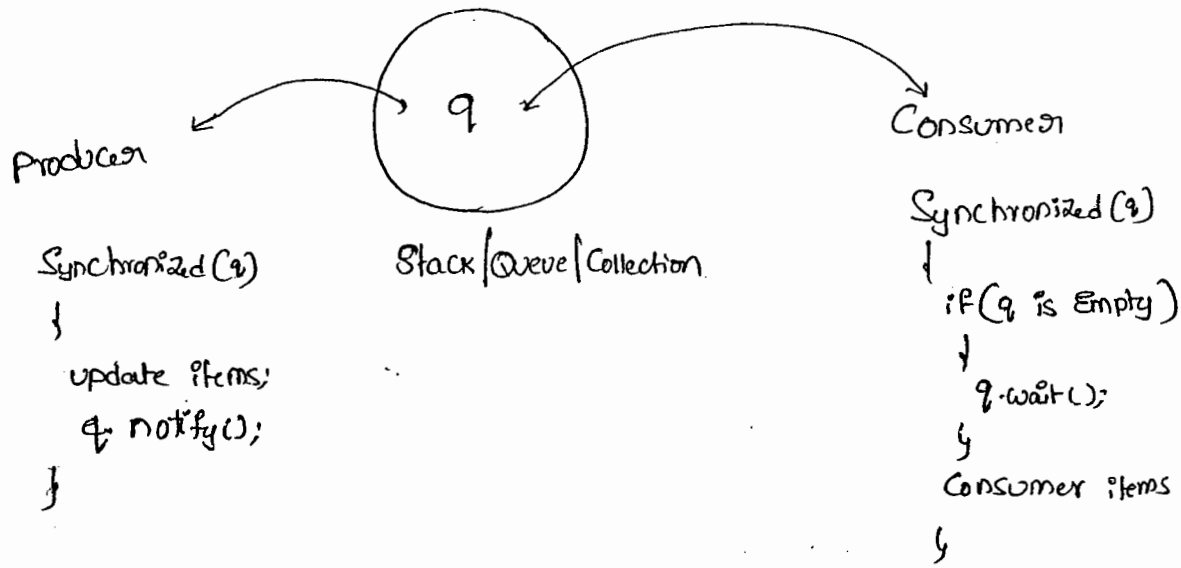
Child giving notification

Main Thread got notification

5050

Ctrl+C
KILL

Producer-Consumer problem:-



↳ Consumer has to Consume items from the Queue

↳ if Queue is Empty, He has to Call wait() method.

↳ producer has to produce items into the Queue.

↳ After producing the items, He has to Call notify() method so that all waiting Consumers will get notification.

notify() vs notifyAll():

→ we can use notify() to notify only one waiting thread. but which waiting thread will be notified we can't expect exactly. All remaining threads have to wait for further notifications.

→ 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.

```
Stack s1 = new Stack();
Stack s2 = new Stack();
```

```
synchronized(s1)
{
  ==
  ==
  s2.wait();
  =
}
```

R.E: IllegalMonitorStateException

```
synchronized(s1)
{
  ==
  s1.wait();
  =
}
```

DeadLock :-

→ If two threads are waiting for each other for ever. Such type of Situation is Called "DeadLock."

→ There are no resolution techniques for deadLock but Several prevension techniques are possible.

ex:-

```
class A
{
 public synchronized void foo(B b)
 {
  S.o.pln("thread1 starts execution foo");
  try
  {
   Thread.Sleep(1000);
  }
  Catch(IE e)
  {
  }
  S.o.pln("thread1 taying to Catch b's last()");
  b.last();
 }
 Public Synchronized void last()
 {
  S.o.pln("Inside A this is last()");
 }
}
```

```
Class B
{
 Public Synchronized void bar(A a)
 {
  S.o.pln("thread2 starts bar");
  try {
      Thread.sleep(5000);
     }
  Catch(IE e) { }
 S.o.pln("thread 2 trying to call a's last");
  a.last();
 }
 Public synchronized void last()
 {
  S.o.pln("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
  }
  P.S.v.m(———).
}}{ new DeadLock()}
```

o/p :-

Thread1 Starts execution of foomethod

Thread2 Starts execution of bar method

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 a thread which ends at certain point of time is Called "Starvation."

Ex:-

least priority Thread has to wait untill Completing all the Threads but this long waiting should Compulsary ends at certain point of time.

→ Hence, A long waiting which never Ends is Called "Deadlock" where as a Long waiting which Ends at certain point of time is Called "Starvation"

Daemon Threads :-

→ The threads which are executing in the background are called "Daemon threads". Ex:- Garbage Collector

→ 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 RuntimeException saying "IllegalThreadStateException".

→ main thread is always non-Daemon & its not possible to change it's 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. i.e, 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 threads will be terminated automatically.

Ex:-

```
Class MyThread extends Thread
{
    Public void run()
    {
      for (int i=0 ; i<10 ; i++)
      {
        S.o.pln("lazy thread");
        try {
            Thread.sleep(2000);
        }
        Catch (InterruptedException e) { }
      }
    }
Catch
Class Test
{
  p.s.v.m(String[] args)
  {
    MyThread t = new MyThread();
    t.setDaemon(true);  ——— ①
    t.start();
    S.o.pln("end of main");
  }
}
```

→ If we are Commenting Line① Then both main & child threads are non-Daemon & hence both will be Executed until their Completion.

→ If we are not Commenting Line① then main thread is non-Daemon & child Thread is Daemon. Hence when ever main thread terminates automatically child thread will be terminated.

How to Kill a Thread :-

→ A Thread Can Stop or Kill another Thread by using Stop() method then automatically running Thread will entered into Dead State. It is a deprecated method & Hence not recommended to use.

```
public void Stop();
```

Suspending & Resuming a Thread :→

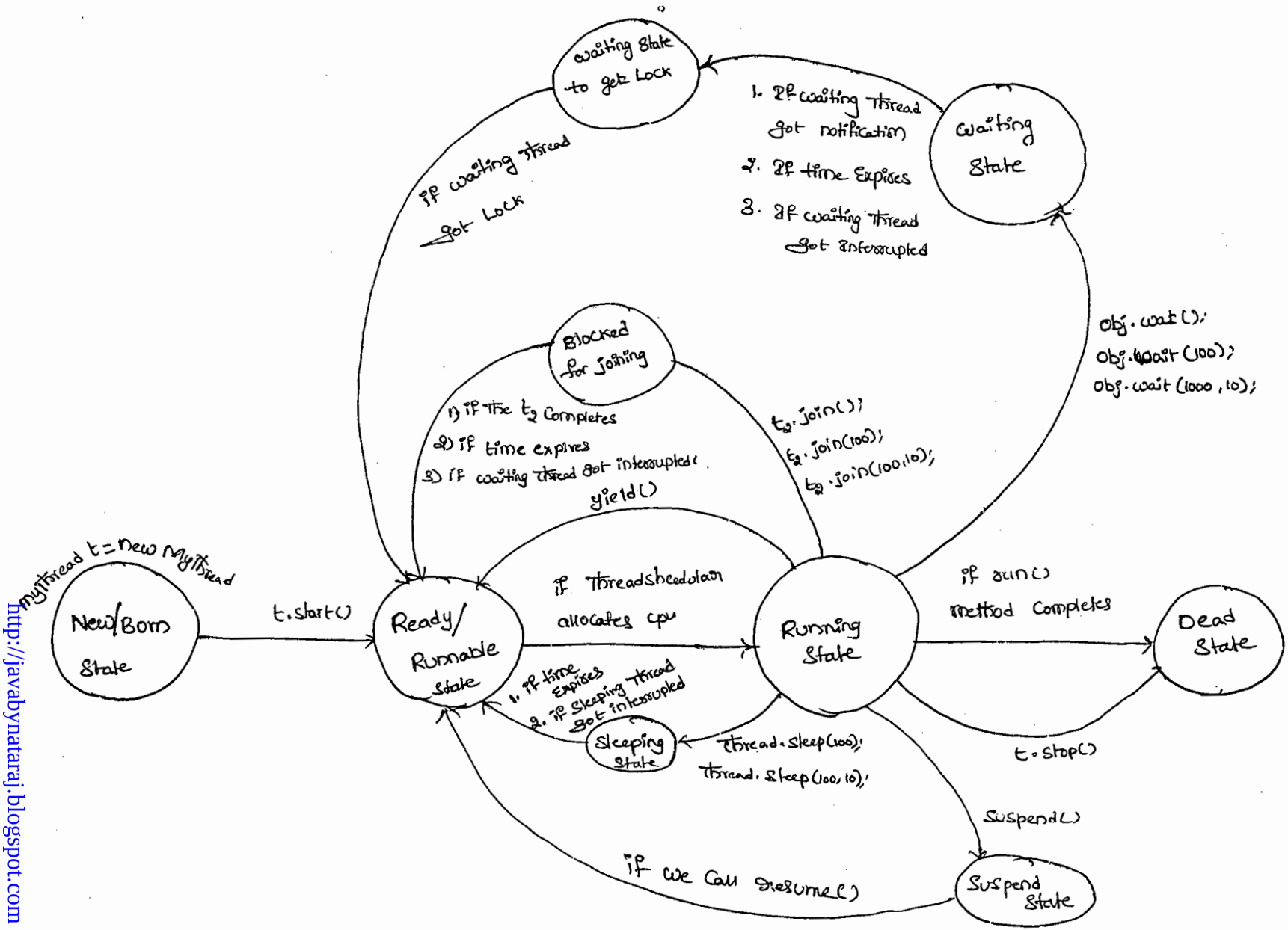
→ A Thread Can Suspend another thread by using Suspend() method.

→ A Thread Can resume a Suspended Thread by using resume() method.

→ Bot these methods are deprecated methods & Hence not recommended to use.

Q) What is a Green Thread?

Q) What is ThreadLocal?


waiting State to get Lock
1. If waiting thread got notification
2. If time Expires
3. If waiting thread got interrupted
Waiting State
if waiting thread got Lock
Blocked for joining
1) if the t2 Completes
2) if time expires
3) if waiting thread got interrupted
t2.join();
t2.join(100);
t2.join(100,10);
obj.wait();
obj.wait(100);
obj.wait(1000,10);
yield()
mythread t=new MyThread
New/Born State
t.start()
Ready/ Runnable State
if threadsheduler allocates cpu
Running State
if run() method Completes
Dead State
1. if time Expires
2. if Sleeping thread got interrupted
Sleeping State
thread.sleep(100);
thread.sleep(100,10);
t.stop()
Suspend()
if we Call resume()
Suspend State

Case 4 :-

```
Class Test
{
 public void m1(int i, float f)
 {
   S.o.pln("int-float version");
 }
 public void m1(float f, int i)
 {
   S.o.pln("float-int version");
 }
 P.S.v.m(____)
 {
   Test t1 = new Test();
   t1.m1(10, 10.5f);
   t1.m1(10.5f, 10);
 X t1.m1(10, 10);   X C.E:- reference to m1() is ambiguous
 X t1.m1(10.5f, 10.5f);  X C.E:-
 }
}
```

Can not find Symbol.

Symbol: method m1(float, float)

location: Class Test.

Case 5 :-

```
Class Animal
{
}
Class Monkey extends Animal
{
}

Class Test
{
  public void m1(Animal A)
  {
    S.o.pln("Animal Version");
  }
  public void m1(monkey m)
  {
    S.o.pln("monkey version");
  }

  P.S.V.m(———)
  {
    Test t = new Test();
    Animal a = new Animal();
      t.m1(a); // Animal version
    Monkey m = new Monkey();
      t.m1(m) // monkey-
    Animal a1 = new monkey();
      t.m1(a1); // Animal
  }
}
```

→ In overloading method resolution always takes Place by Compiler based on reference type and Runtime Object never play any role in overloading.

Over riding :→

03/05/11

→ "What ever the parent has by default available to the child. if the child not Satisfied with parent class implementation then child is allowed to redefine its implementation in its own way." this process is Called "overriding"

→ The parent class method which is overridden is Called overridden method & the child class method which is overriding is Called overriding method.

Ex:-

```
Class P
{
    public void property()
    {
        S.o.pln("Cash + Gold + land");
    }
    Public void marry()
    {
        S.o.pln("Sobba Laxmi");
    }
}

Class C extends P
{
    Public void marry()
    {
        S.o.pln("Kajal | 3sha | 4tara | 4me");
    }
}
```

overridden method

overriding

overriding method

Ex2:-

```
Class P
{
  public void m1()
  {
    S.o.pln("parent");
  }
}
Class C extends P
{
  public void m1()
  {
    S.o.pln("child");
  }
}
Class Test
{
  p.s.v.m(——)
  {
    P p = new P();
      p.m1();  // parent
    C c = new C();
      c.m1();  // child
    P p1 = new C();
      p1.m1();  // child
  }
}
```

overriding

→ In overridding the method resolution always takes care by JVM based on runtime object & in overridding reference type never play any role.