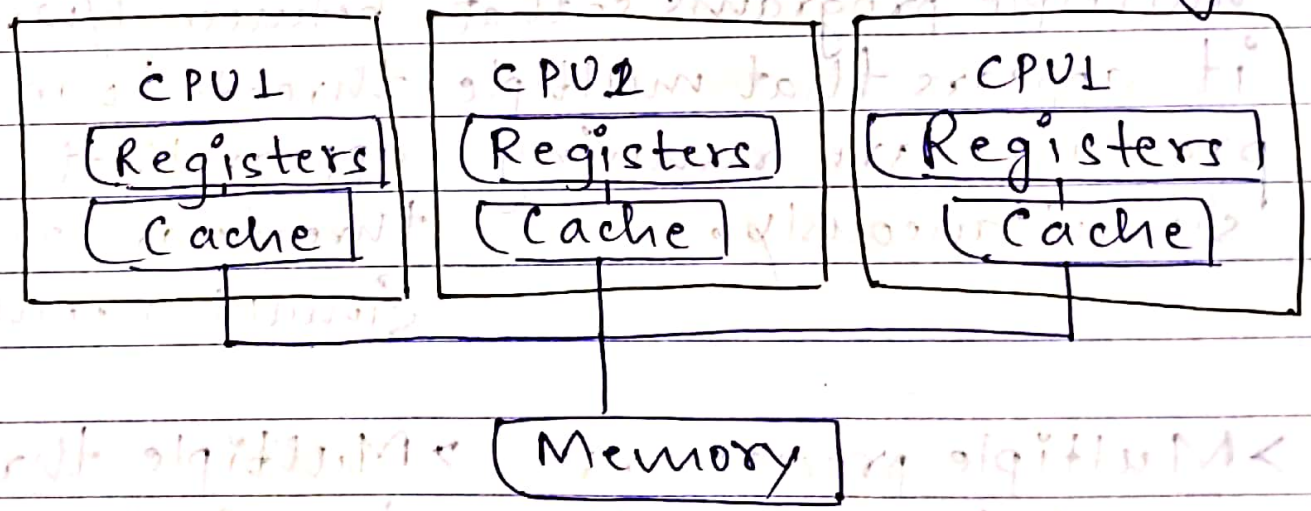
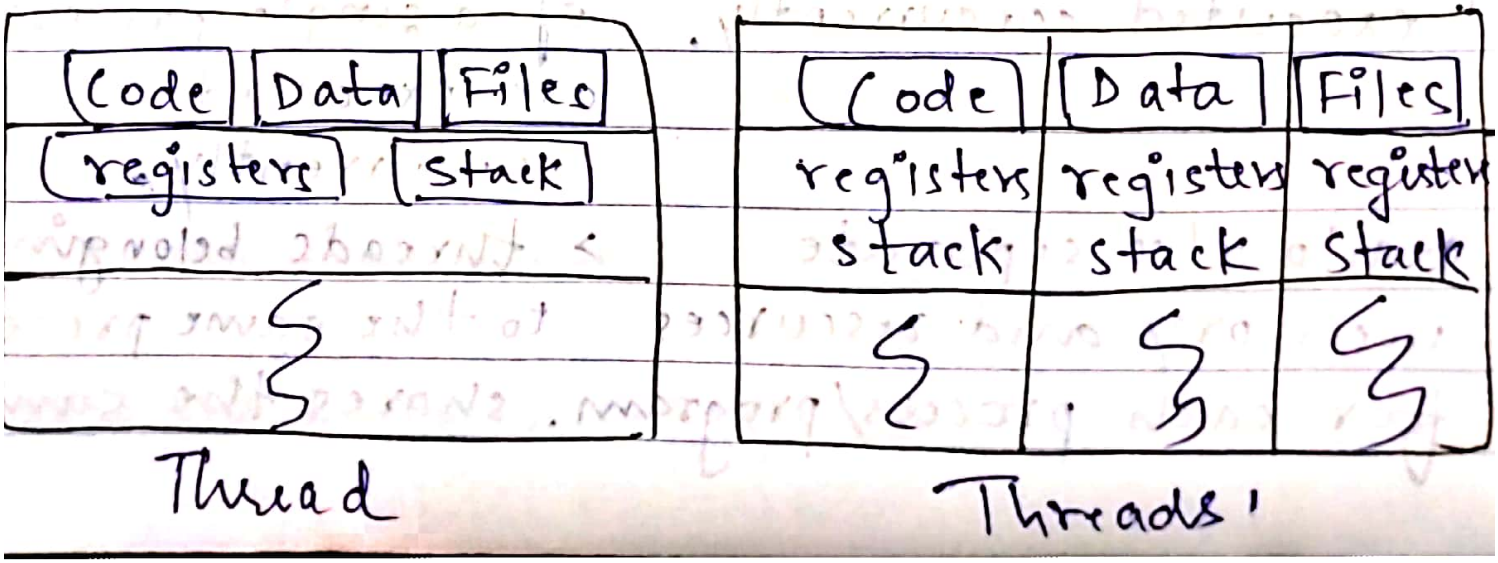
MULTITHREADING IN JAVA.

is a process of executing multiple threads simultaneously.

Multitasking

> is a process of executing multiple tasks simultaneously.
> used to utilize CPU and increase its performance.
> Achieved in two ways:
> > Process-based (multiprocessing)
> > Thread-based (multithreading)





Process-based Multitasking (Multiprocessing)	Thread-based Multitasking. (Multithreading)
> System allows executing multiple programs and tasks at a time.	> System executes multiple threads of same or different processes at the same time.
> Increases the computing speed of the system.	> Increase the responsiveness of the system.
> CPU has to switch b/w multiple programs so that it appears that multiple programs are running simultaneously.	> CPU has to switch between b/w multiple threads to make it appear that all threads are running simultaneously.
> Multiple processes are executed concurrently.	> Multiple threads of a single process are executed concurrently.
> allocates separate memory and resources for each process/program.	> threads belonging to the same process shares the same

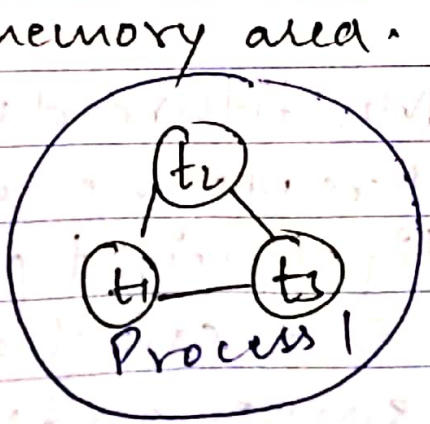
memory and resources as that of the process.

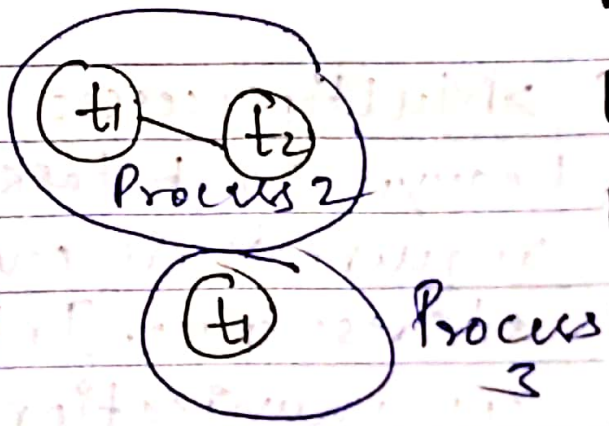
> It is sharing of computer resources among the number of processes or execution of different processes at the same time like working on MS office while VLC player is running.	> It is execution of the number of different tasks within the same process like in MS word while you are writing the auto-correction is done simultaneously.
> is forking a process to work in non-blocking mode, allocates separate code and data space for each child process.	> is a multitasking within a process, where child threads share the same code and data space with parent thread.
> Multiprocesses are heavyweight tasks that require their own address space. Interprocess communication is very	> Multithreading threads are a lightweight process and can share same address

expensive and context switching from one process to another process is costly since they are running in diff. add. spaces.	space and inter-thread communication is less expensive than interprocess.
> Relies on pickling objects in memory to send to other process.	> Avoids pickling.
> Can be symmetric or asymmetric.	> Multithreading is not classified.

- Thread is a lightweight process, the smallest unit of processing.
> It is a separate path of execution.

> Threads are independent. It uses shared memory area.





#At least one process is required for each thread.

Java Thread Class

> provides constructors and methods to create and perform operations on thread.

> Thread class extends Object class and Implements Runnable interface.

Case 1: Thread Schedular

> It is part of JVM.

> It is responsible to schedule threads i.e if multiple threads are waiting to get the chance of execution then in which order threads will be executed is decided by thread schedular.

> We can't expect the algorithm used by Thread Schedular, it is varied from JVM to JVM and because of that we can't expect thread execution exact order and output.

```
MyThread t = new MyThread();
t.start(); → method of Thread class
```

It internally calls run() method.

responsible to start thread.

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

Ans → t.start()
A thread (flow) will be created which is responsible to execute our job.
t.run()
No thread will be created and run method will execute like a normal method call by object t.

Case 3: Why only start() method to execute a thread.

Ans:
> Register thread with Thread Schedular
> Perform all other mandotory activities
> Invoke run()

Case 4: Overloading of run() method

> Yes, it can be done but start method method calls only no argument run method.

The other overloaded run() methods have to call explicitly.

Case 5 : If we are not overriding run method, then

> Thread class run method will be executed which has empty implementation. Hence, there will be no output.

> It is highly recommended to override run method of Thread Class, else don't go for multithreading.

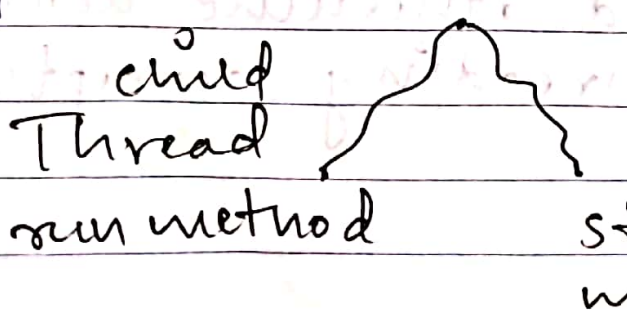
Case 6 : Overriding of start method.

> If we override the start method then our method will execute just like normal method call and new thread won't be created.
→ called by only main method.

> It is highly recommended not to override start method otherwise don't go for multithreading concepts.

Case : 7

```
class MyThread extends Thread
{
    public void start()
    { * super.start();
        sop("start method!");
    }
    public void run()
    {
        sop("run method");
    }
}

class Test{
    public static void main(String arg[])
    {
        MyThread t = new MyThread();
        t.start();
        SOP("main method");

    }
}
```

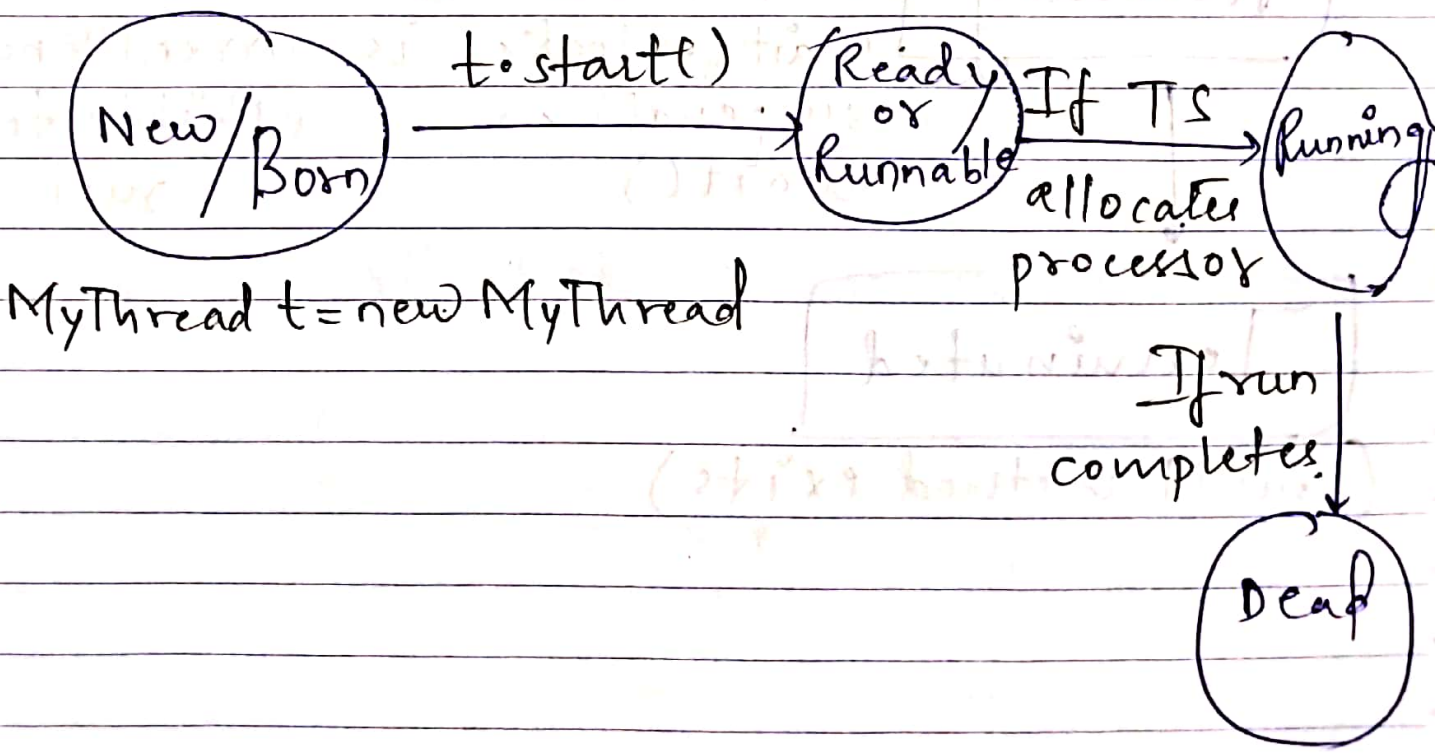


Case 8 : ⊨| Thread cannot be started again :

```
Thread t = new Thread();
t.start();
!
t.start();
```

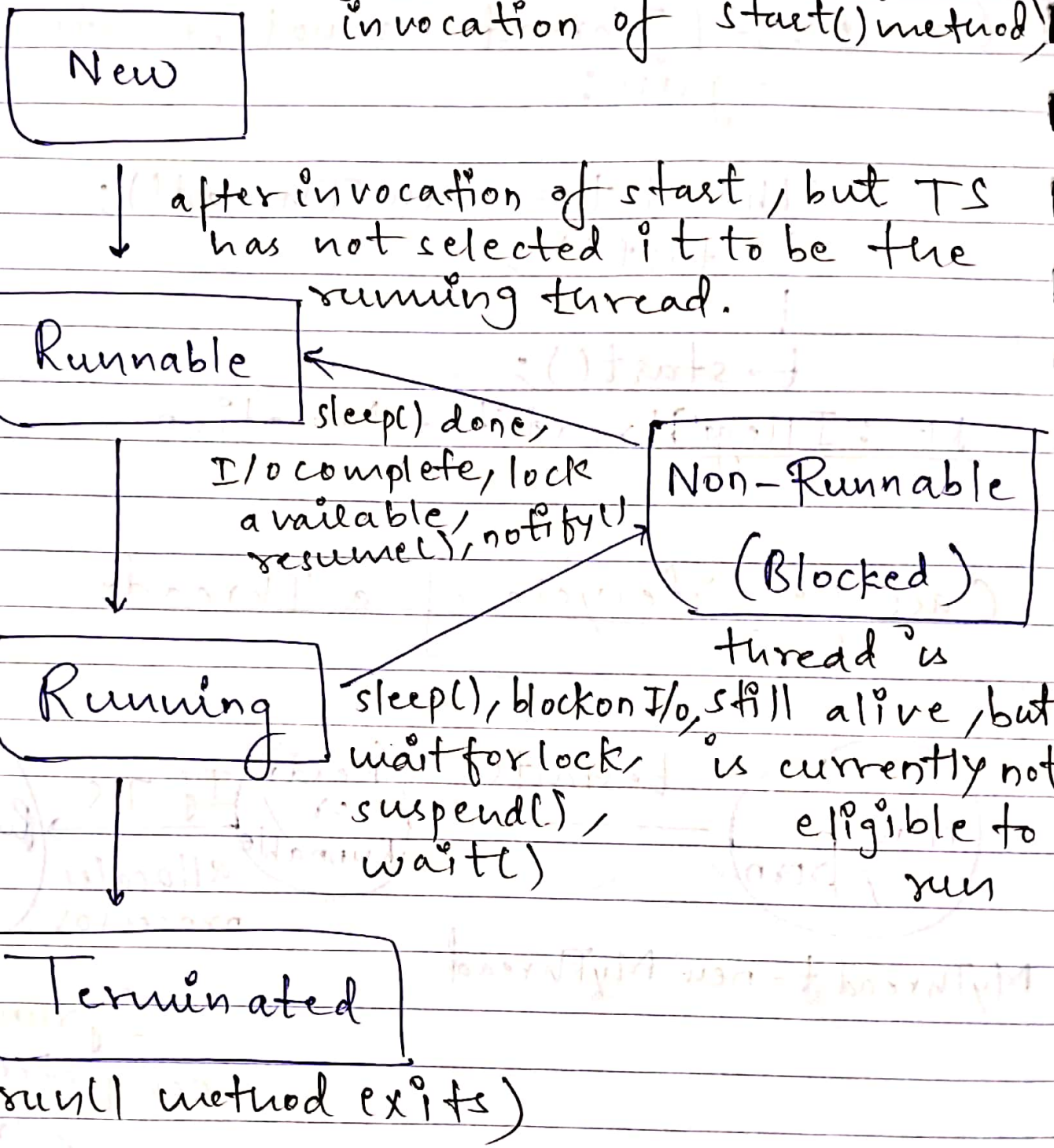
RE : IllegalThreadStateException

Case 9 : Lifecycle of a Thread :

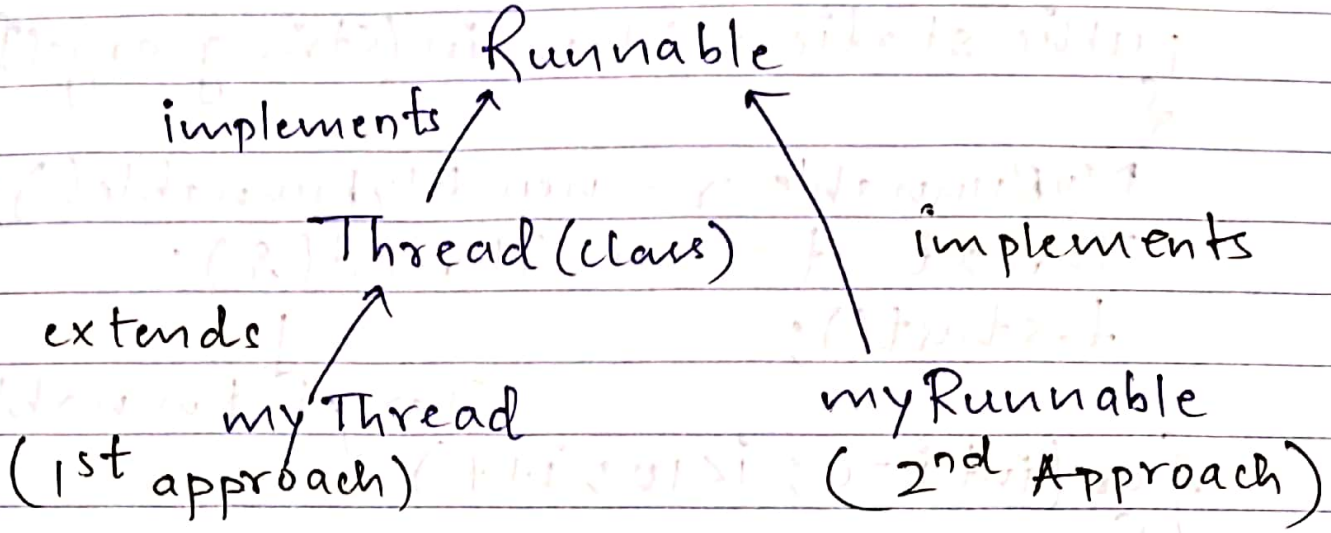


MyThread t = new MyThread

(create instance of Thread class but before invocation of start() method)



2) By Implementing Runnable Interface.



> We can define a thread by implementing Runnable interface.
> Runnable interface present in java.lang package.
> It contains only one method run() method. (public void run(){}).

```
class MyRunnable implements Runnable
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            SOPln("Child Thread");
        }
    }
}
```

```
class ThreadDemo
{
    public static void main (String args[])
    {
     MyRunnable r = new MyRunnable();
     Thread  t = new Thread(r);
     t.start();                          ↓
                                  Target Runnable.
    for (int i=0; i<10; i++)
    {
        sOPn("main Thread");
    }
}
```

↳ executed by main Thread.

Case Study :

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

case 1 : t1.start()

> A new thread will be created and which is responsible for execution of Thread class run method, which has empty implementation.

Case 2: t1.run();
> No, new thread will be created and thread class run method will be executed just like a normal method call.

Case 3: t2.start();
> A new thread will be created which is responsible for the execution of MyRunnable class run method.

Case 4: t2.run();
> A new thread won't be created and myRunnable run method will be executed just like a normal method call.

Case 5: r.start();
> Compile-time Error
> : My Runnable class doesn't have start capability.

Case 6: r.run();
> No, new thread will be created and MyRunnable run method will be executed like normal method call.

Ques. Which approach is best to define a thread.

Answer - Among two ways of defining a thread implements Runnable approach is recommended.

> In the 1st approach our class always extends Thread class, there is no chance of extending any other class. Hence we are missing inheritance benefit.

> But in the 2nd approach while implementing Runnable interface we can extend any other class, Hence we won't miss any inheritance benefit.

Because of above reason implementing Runnable approach is recommended than extending Thread class.

Thread Class Constructors :

> Thread t = new Thread();
> Thread t = new Thread(Runnable r);
> Thread t = new Thread(String name);
> Thread t = new Thread(Runnable r, String name);
> Thread t = new Thread(ThreadGroup g, String name);
> Thread t = new Thread(ThreadGroup g, Runnable r);
> Thread t = new Thread(ThreadGroup g, Runnable r, String name);
> Thread t = new Thread(ThreadGroup g, Runnable r, String name, long stacksize);

How we can get name of a Thread :

> Getting & Setting name of a Thread.

Every Thread in java has some name, it may be default name generated by JVM or customized name provided by programmer.

> We can get and set name of a Thread by using the following two methods of thread class :

- public final String getName()
- public final void setName(String name);

```
// for main thread
    Thread.currentThread().getName();
Thread.currentThread.setName("Abhi");
// for other
    MyThread t = new MyThread();
    t.getName();
    t.setName("Abhi");
```

The currentThread() method returns a reference of currently executing thread.

Thread Priorities :

> Every thread in java has some priority, it may be default priority generated by JVM or customized priority explicitly provided by the programmer.

> Valid range of priorities : 1-10
> where 1 is min-priority and 10 is max-priority.

> Thread class defines the following constants to represent some standard priorities

Thread.MIN_PRIORITY → 1
Thread.NORM_PRIORITY → 5
Thread.MAX_PRIORITY → 10

> Thread scheduler will use thread priorities while allocating processor. The thread which is having highest priority will get chance first.

> If two threads having same priority then we can't exact execution order, it depends on Thread Scheduler.

> Thread class defines the following methods to get and set priority of a thread.

public final int getPriority()

public final void setPriority(int p)

Allowed values range 1 to 10

otherwise RE: IllegalArgumentExcep.

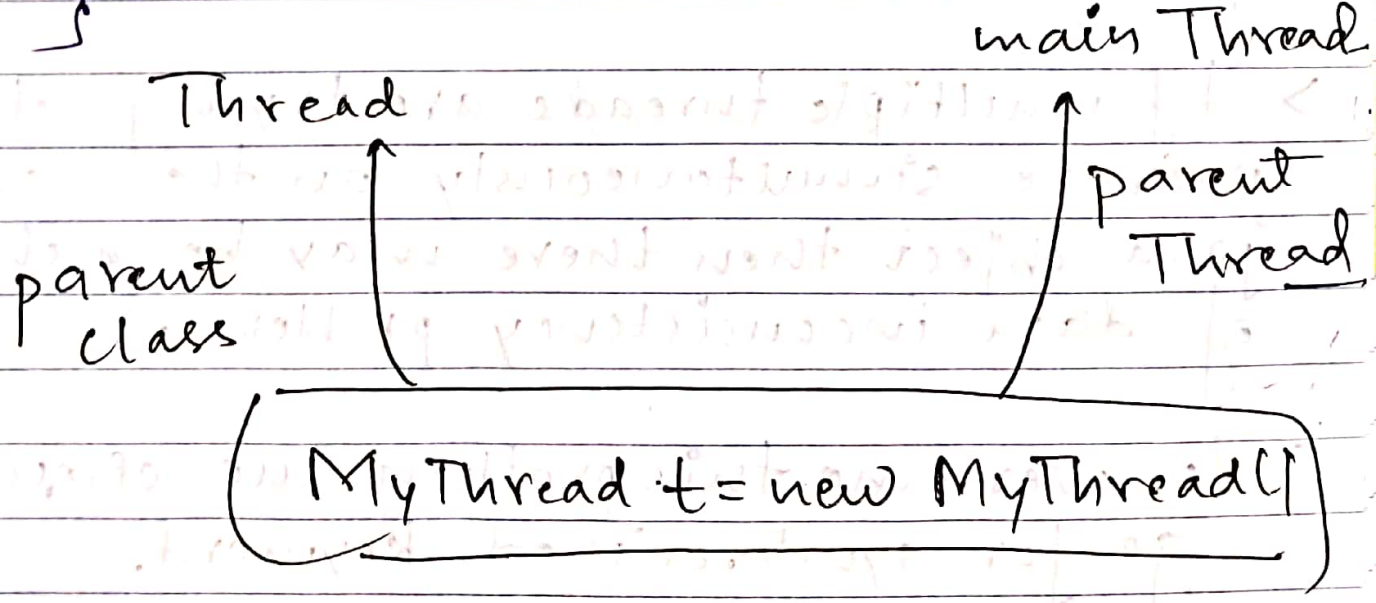
Default Priority:

> The default priority for ~~ev~~ only main thread is 5.

> For other threads default priority will be inherited from parent to child i.e whatever priority has the same priority will be there for the child thread.

→ whatever priority parent thread.

```
class MyThread extends Thread
{
}
class Test
{
    p v s m (String args[])
    {
    SOPn(Thread.currentThread().getPriority());
    // 5
    Thread.currentThread().setPriority(7);
    MyThread t = new MyThread();
    SOPn(t.getPriority());
    //ans: 7
    }
}
```



Synchronization

synchronization in java is the capability to control the access of multiple threads to any shared resource.

It is mainly used to
> prevent thread interference
> prevent consistency problem.

> synchronized is the modifier applicable only for methods and blocks but not for classes and variables.

> If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem.

> To overcome this problem we should go for synchronized keyword.

> 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 so that data inconsistency problem will be resolved.

> The main advantage of synchronised keyword is we can resolve data inconsistency problem but the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance specific problem.

Hence, If there is no specific requirement then it is not recommended to use synchronized keyword.

> Public Telephone booth is best example of synchronized keyword.

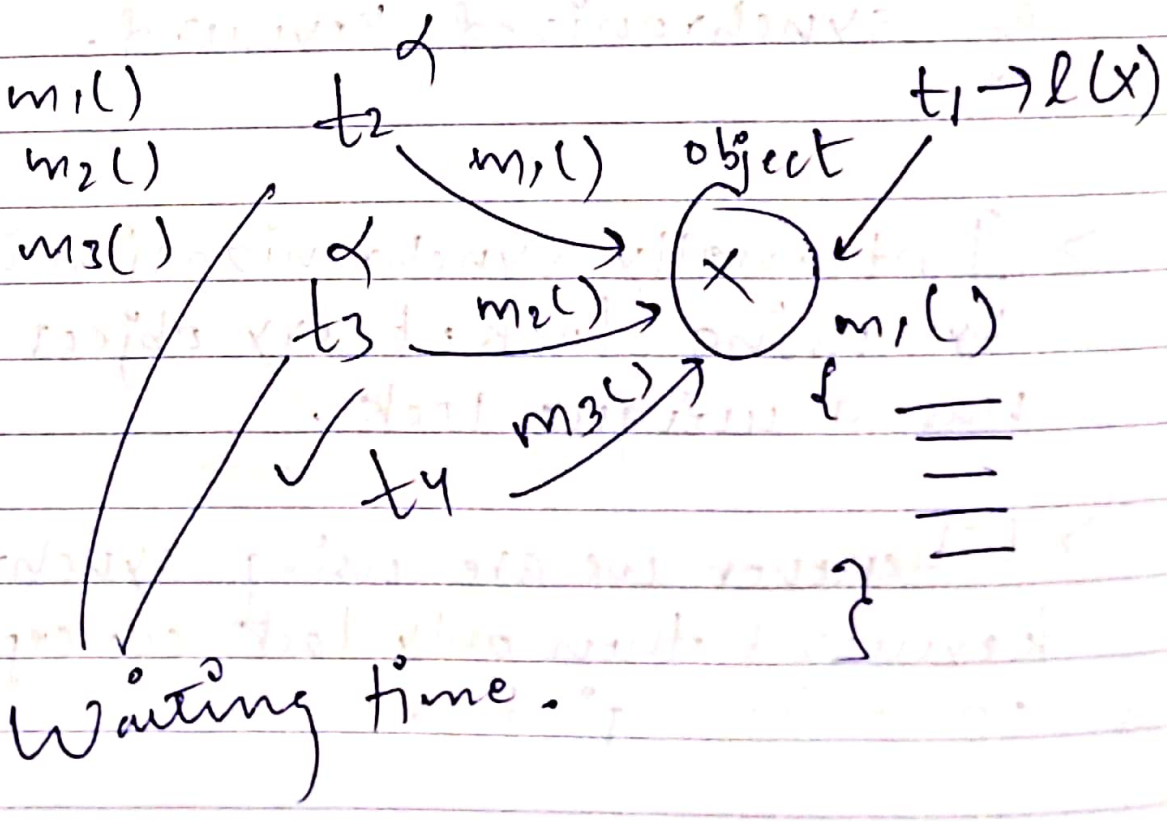
> Internally synchronization is implemented by using lock. Every object in java has a unique lock.

> Whenever we are using synchronized keyword then only lock concept will come into picture.

> If a thread wants to execute synchronized method on the given object first it has to get the lock of that object. Once thread got the lock then it is allow to execute any synchronized method on that object.
Once the method execution completed automatically thread releases the lock.

Acquiring and releasing lock internally takes care by JVM and the programmer is not responsible for this activity.

```
class X
{
  synch m1()
  synch m2()
  normal m3()
}
```

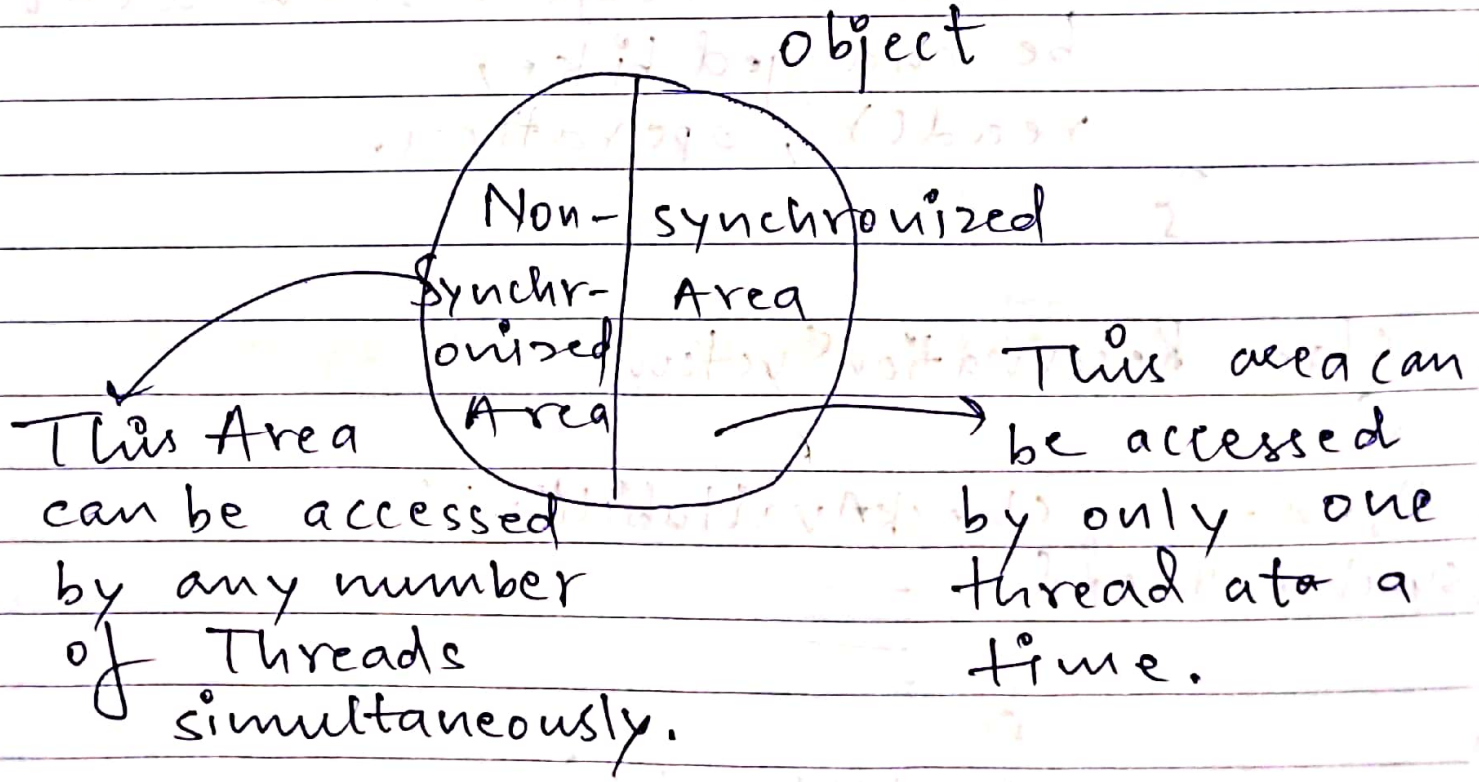


Imp.

> While a thread executing synchronized method and the given object, the remaining threads are not allowed to execute any synchronized method simultaneously on the same object.

> But the remaining threads are allowed to execute non-synchronized methods simultaneously.

*** Lock concept is implemented based on object but not based on method.



Imp.

> class X
{
synchronized Area
{ wherever we are performing update operation (Add/remove/delete/replace)

i.e where state of object is changing
}

non-synchronized
{
whereever, object state won't be changed like, read(), operation.
}

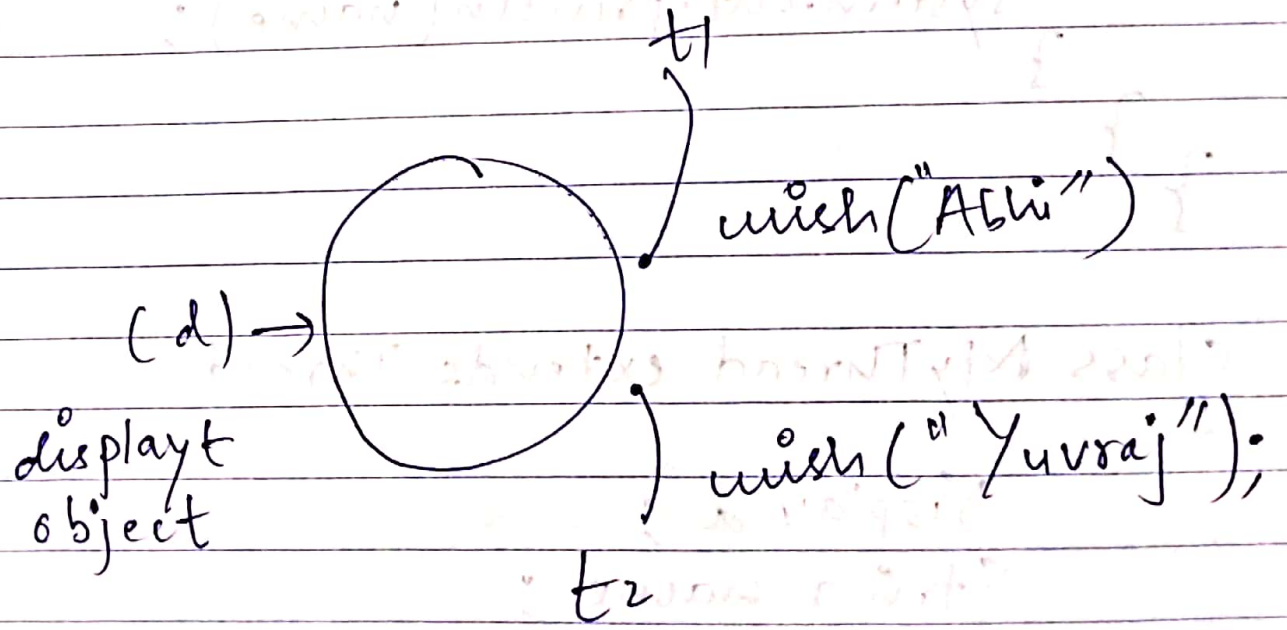
class ReservationSystem
{
Non-synchronized checkAvailability()
{
}

synchronized bookTicket()
{
}
}

Code (Important)

```
class Display
{
    public synchronized void wish(String name)
    {
        for(int i=0; i<10 ; i++)
        {
            System.out.println("Good Morning:");
            try
            {   Thread.sleep(2000);
            }
            catch(InterruptedException e){ }
            System.out.println(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);
    }
}

public class SynchronizedDemo
{
    Display d = new Display();
    MyThread t1 = new MyThread(d, "Abhi");
    MyThread t2 = new MyThread(d, "Vai");
    t1.start();
    t2.start();
}
```

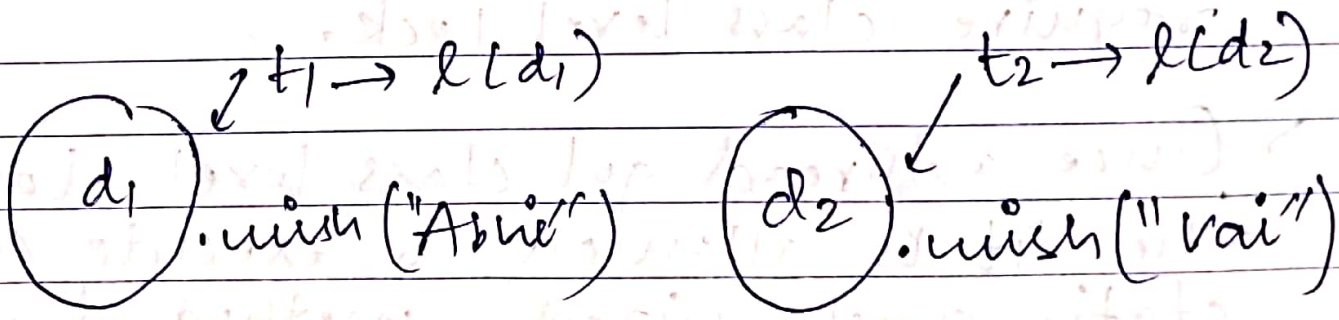


> If we are not declaring wish method as synchronized then both threads will be executed simultaneously and hence we will get irregular output.

> If we declare wish method as synchronized then at a time only one ~~me~~thread is allowed to execute wish method on the given display object.
Hence, we will get irregular output.

Case study:

```
Display d1 = new Display();
Display d2 = new Display();
MyThredd t1 = new MyThread(d1,"Abhi");
My Thread t2 = new MyThread(d2,"rai");
t1.start();
t2.start();
```



→ Even though which method is synchronized we will get irregular output because threads are operating on different java objects.

Conclusion:

If multiple threads are operating on same java object then synchronisation is required.

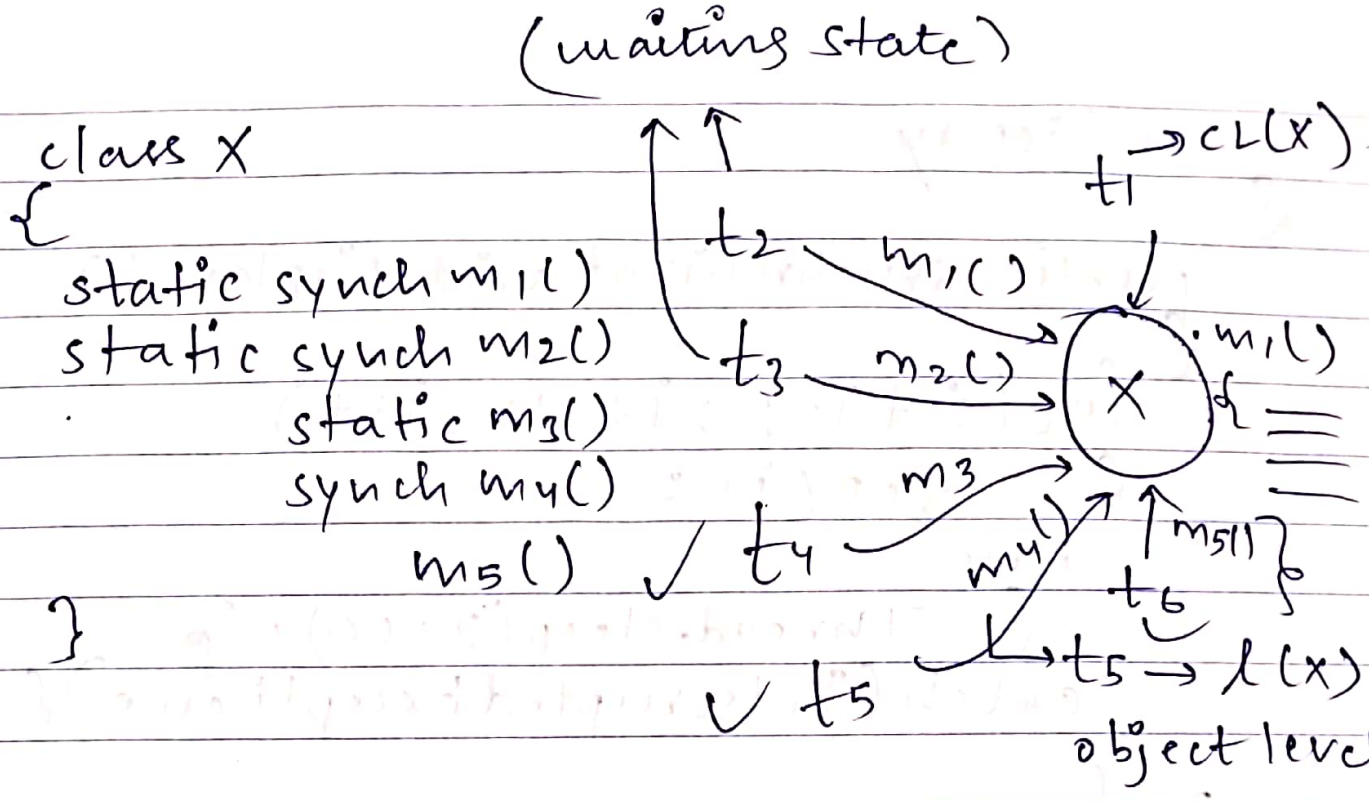
If multiple threads are operating on multiple java objects then synchronization is not required.

> Every class in java has a unique lock which is also known as class level lock.

> If a thread wants to execute a static synchronized method then thread require class level lock.

> Once a thread got class level lock then it is allowed to execute any static synchronized method of that class.

> Once the method execution gets complete automatically thread releases the lock.



> While a thread executing static synchronized method the remaining threads are not allowed to execute static synchronized method of that class simultaneously.

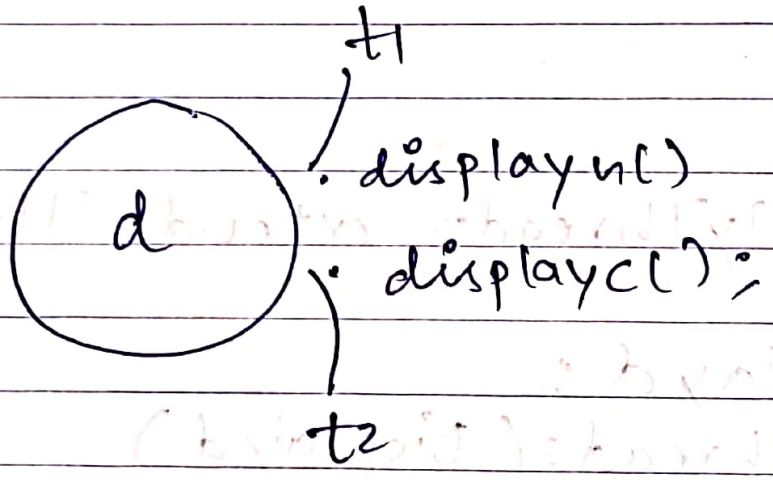
> But remaining threads are allowed to execute the following methods simultaneously:

- > normal static methods
- > synchronized instance methods
- > normal instance method.

```
class Display
{
   public synchronized void displayn()
   {
     for(int i=1 ; i<=10 ; i++)
     {   SOP(i);
       try
        { Thread.sleep(2000); }
       catch(InterruptedException e){ }
     }
   }

   public synchronized void displayc()
   {
      for(int i=65 ; i<=75 ; i++)
      {
         SOP((char)i);
        try
        { Thread.sleep(2000); }
        catch (InterruptedException e){ }
      }
   }
}
```

```
class MyThread1 extends Thread
{
    Display d;
    MyThread1(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayn();
    }
}

class MyThread2 extends Thread
{
    Display d;
    MyThread2(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayc();
    }
}
```

```
class SynchronizedDemo1
{
    p v s m (String[] args)
    {
        Display d = new Display();
        MyThread1 t1 = new MyThread1(d);
        MyThread2 t2 = new MyThread2(d);
        t1.start();
        t2.start();
    }
}
```



Synchronized Block :

> If very few lines of the code require synchronisation, then it is not recommended to declare entire method as synchronised. We have to enclose those few lines of the code by using synchoronised block.

> The main advantage of synchronised block over synchronised method is it reduces waiting time of threads and improves performance of the system or application.

> We can declare synchronised block as follows -

① To get lock of current object :

```
synchronised (this)
{
    ===
}
```

If a thread got lock of current object then only it is allowed to execute this Area.

(2) To get lock of particular object :

```
synchronised (b)
{
    ========
}
```

If a thread get lock of particular object 'b' then only it is allowed to execute this Area.

(3) To get class level lock :

```
synchronised (Display.class)
{
    ========
}
```

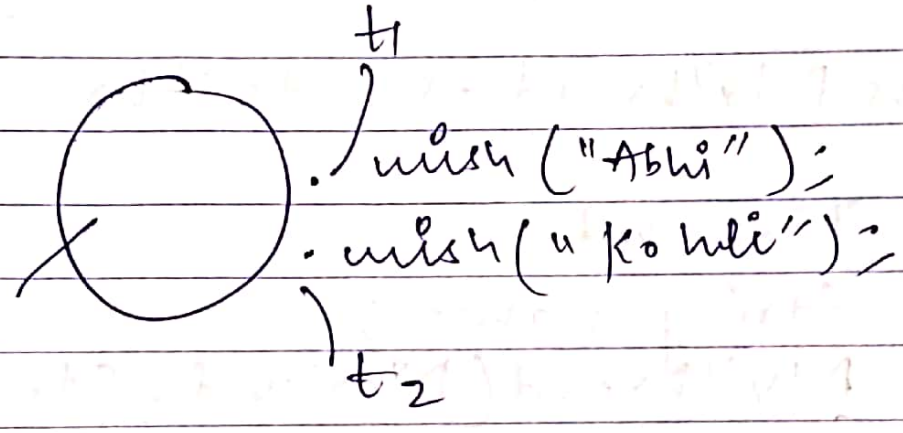
If a thread got class level lock of Display class, then only it is allowed to execute the Area.

Code :

```
class Display{

public void wish (String name)
{
    ;;;; // 1 lakh lines of code
```

```
synchronized (this){
for(int i=0 ; i<4 ; i++)
{
    Sopn("Hello:");
    try{ Thread.sleep(2000); }
    catch(InterruptedException e){}
    Sopn(name);
}
}
}
    ;;;// 1 lakh lines of code
}
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);
    }
}
```

```
public class Demo{

    Display d = new Display();
    MyThread t1 = new MyThread(d,"Abhi");
    MyThread t2 = new MyThread(d,"Kohli");
    t1.start();
    t2.start();

}
```



Imp:

> Conclusion

Lock concept applicable for object types and class types but not for primities.

Hence, we can't pass primitive type as argument to synchronized block otherwise we will get compile time Error Saying

CE: Unexpected type
found: int
required: reference

```
int x=10
synchronized(x)
{
    ==
}
```

CE : unexpected type
found : int
required : reference.

FAQs :

Q-1 : What is synchronized keyword? where we can apply.

Q-2 : Explain advantage of synchronized keyword.

Q-3 : Explain disadvantage of synchronized keyword.

Q-4. What is race condition.

Ans. If multiple threads are operating simultaneously on same java object then, there may be a chance of data inconsistency problem. This is called race condition.
We can overcome this problem by using synchronized keyword.

Q-5. What is object lock and when it is required?

Q-6 What is class level lock and when it is required?

Q-7. What is difference b/w class level lock and object lock.

Q-8 While a thread executing synchronized on the given object is remaining threads are allowed to execute any other synchronized method simultaneously on the same object.

Ans No.

Q-9. What is synchronized block.

Q-10. How to declare synchronized block to get ~~lock of~~ class level lock.

Q-11. What is the advantage of synchronized block over synchronized method.

Q-12. Is a thread can acquire multiple locks simultaneously.

→ ~~No~~ Yes.

```
class X
{   public synchronized void m1()
    {                          → here, thread has lock of
        Y y = new Y();           'X' object

        synchronized(y)
        {                      → here, thread has
            Z z = new Z();       lock of 'X'
                                 & 'y'.
            synchronized(z)
            {  ========  → here, thread has
            }  ========    locks of X,
        }                    Y, & Z.
    }
}
```

```
X x = new X();
x.m1();
```

Ans - Yes, ofcourse from difference objects.

Ques-13 : What is synchronized statement?

Ans: The statements present inside the synchronized method or block. The statement which is executed by 1 thread at a time.

Inter-Thread Communication :

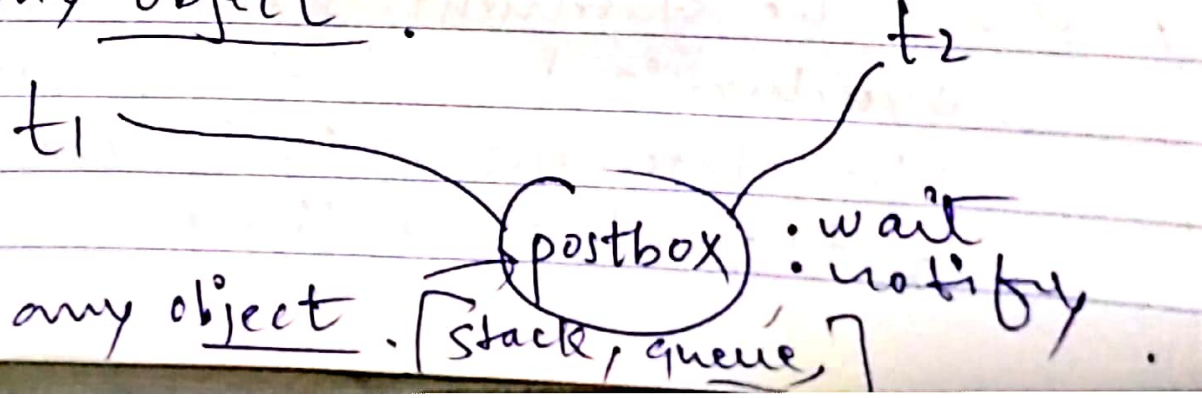
> Two threads can communicate with each other with wait(), notify() and notifyall() method.

> The thread which is expecting updation is responsible to call wait() method then immediately the thread will enter into waiting state.

> The thread which is responsible for performing updation, after performing updation it is responsible to call notify method. Then waiting thread will get that notification and continue its execution with those updated items.

#**

> Wait(), notify() & notifyAll() method present in object class but not in Thread class.
Because thread can call these methods on any object.

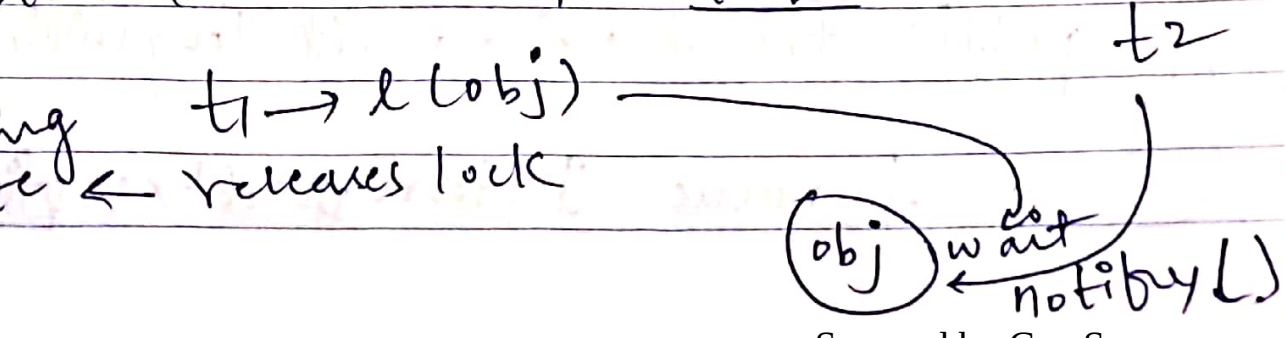


> To call wait(), notify() or notifyAll() methods on any object, Thread should be owner of that object i.e Thread should have lock of that i.e the Thread should be inside synchronized Area.

> Hence we can call wait(), notify() & notifyAll() methods from only synchronized area otherwise we will get RE: IllegalMonitorStateException.

> If a thread calls wait method on any object it immediately releases the lock of that particular object and enter into waiting state.

> If a thread calls notify() method on any object it releases lock of that object but may not be immediately. Except wait(), notify() and notifyAll() there is no other method where thread releases the lock.



Ques. Which of the following is valid -

(1) > If a thread calls wait() method immediately it will enter into waiting state without releasing any lock.

(2) > If a thread calls wait(), it releases the lock of the object but not immediately.

(3) > If a thread calls wait() method on any object it releases all locks acquired by the thread immediately enter into waiting state. ✓

(4) > If a thread calls wait() method on any object it immediately releases lock of that particular ~~state~~ object & enters into waiting state. ✓ valid

Method signature -

public final void wait()

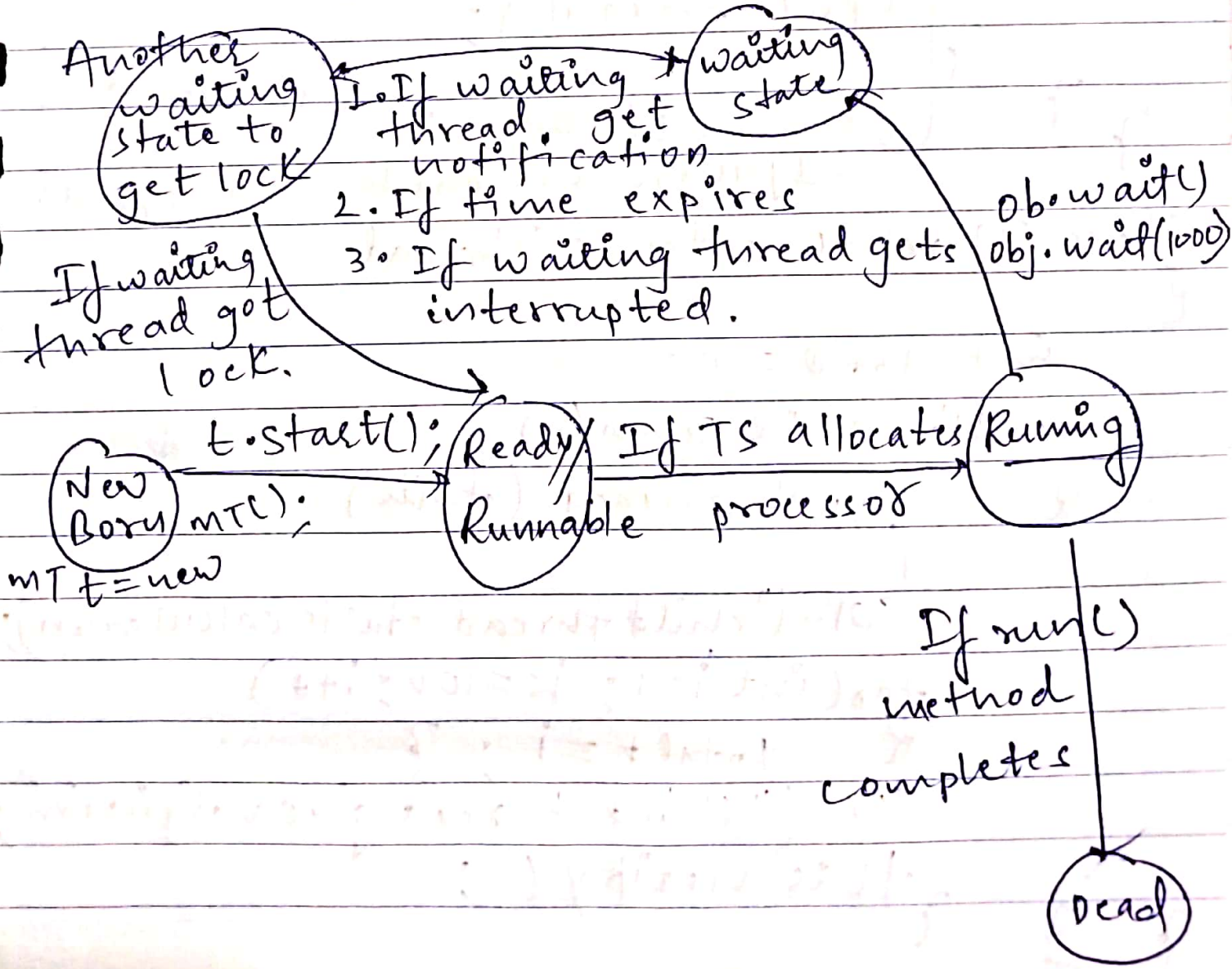
public final native void wait(long milli)

public final void wait(long milli/long nano) throws InterruptedException.

> public final native void notify();

> public final native void notifyAll();

Note: Every wait method throws InterruptedException which is checked exception, hence whenever we are using wait method compulsory we should handle this InterruptedException either by try-catch or throws otherwise we can get compile-time Error.



Code

```
Class ThreadA
{
    public static void main (String args[])
    {
        ThreadB b = new ThreadB();
        b.start();
        synchronized(b)
        {
            SOPn("main thread call wait()");
            b.wait();
            SOPn("main thread got notification");
            SOPn(b.total);
        }
    }
}
```

* b.wait(10000); //child executes first.
//If there is no one to give notification

```
Class ThreadB extends Thread
{
    int total = 0;
    public void run()
    {
        synchronized(this)
        {
            SOPn("child thread starts calculation");
            for(int i=1; i<=100; i++)
            { total += i; }
            SOPn("child thread give notification");
            this.notify();
        }
    }
}
```

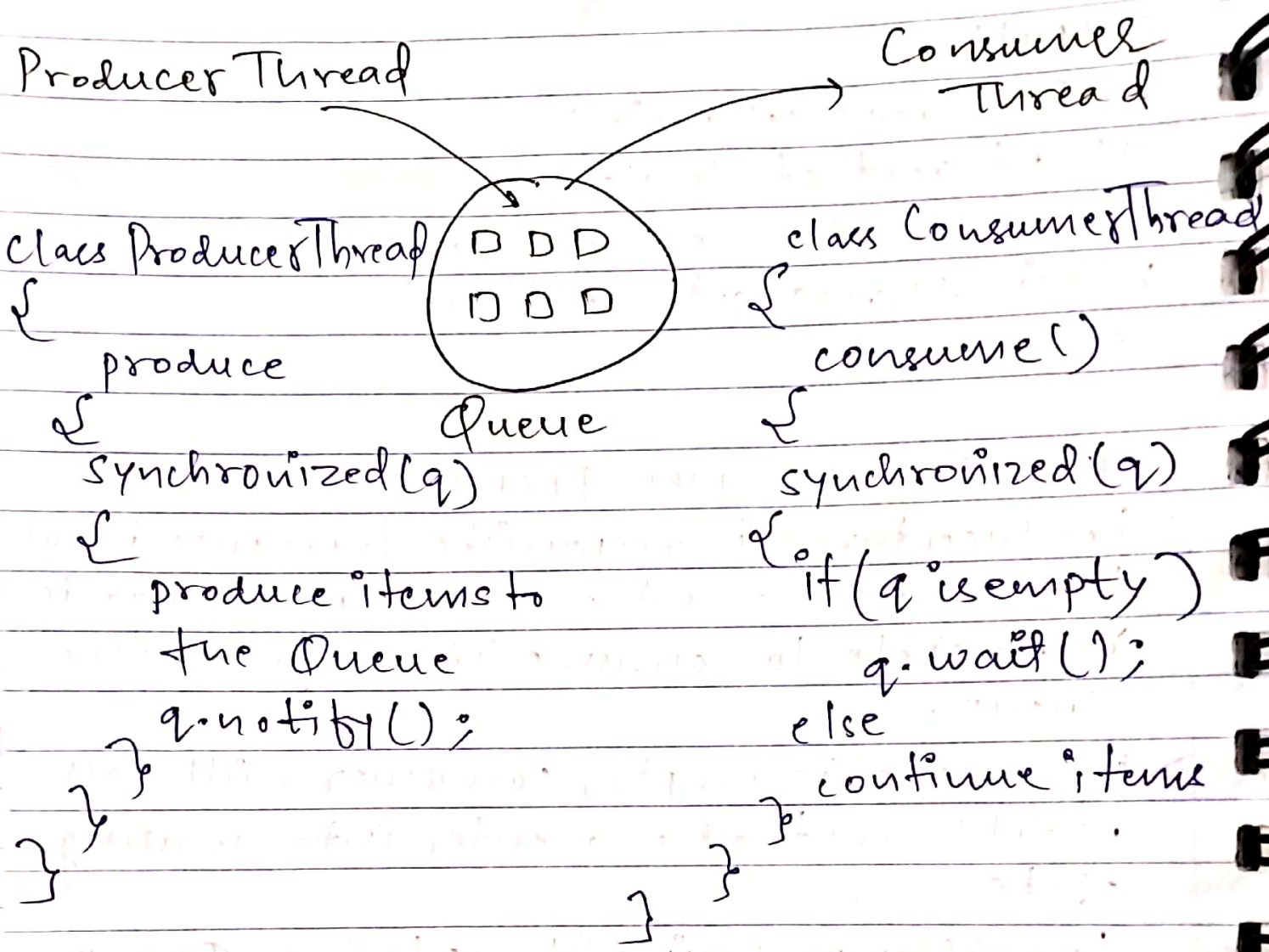
output is :
main thread call wait()
child thread starts calculation
child thread give notification
main thread got notification
5050

PRODUCER- CONSUMER PROBLEM —

> Producer thred is responsibe to produce items to the queue and the consumer thread is responsible to consume items from the queue.

> If queue is empty, consumer will call wait() method and enter into waiting state.

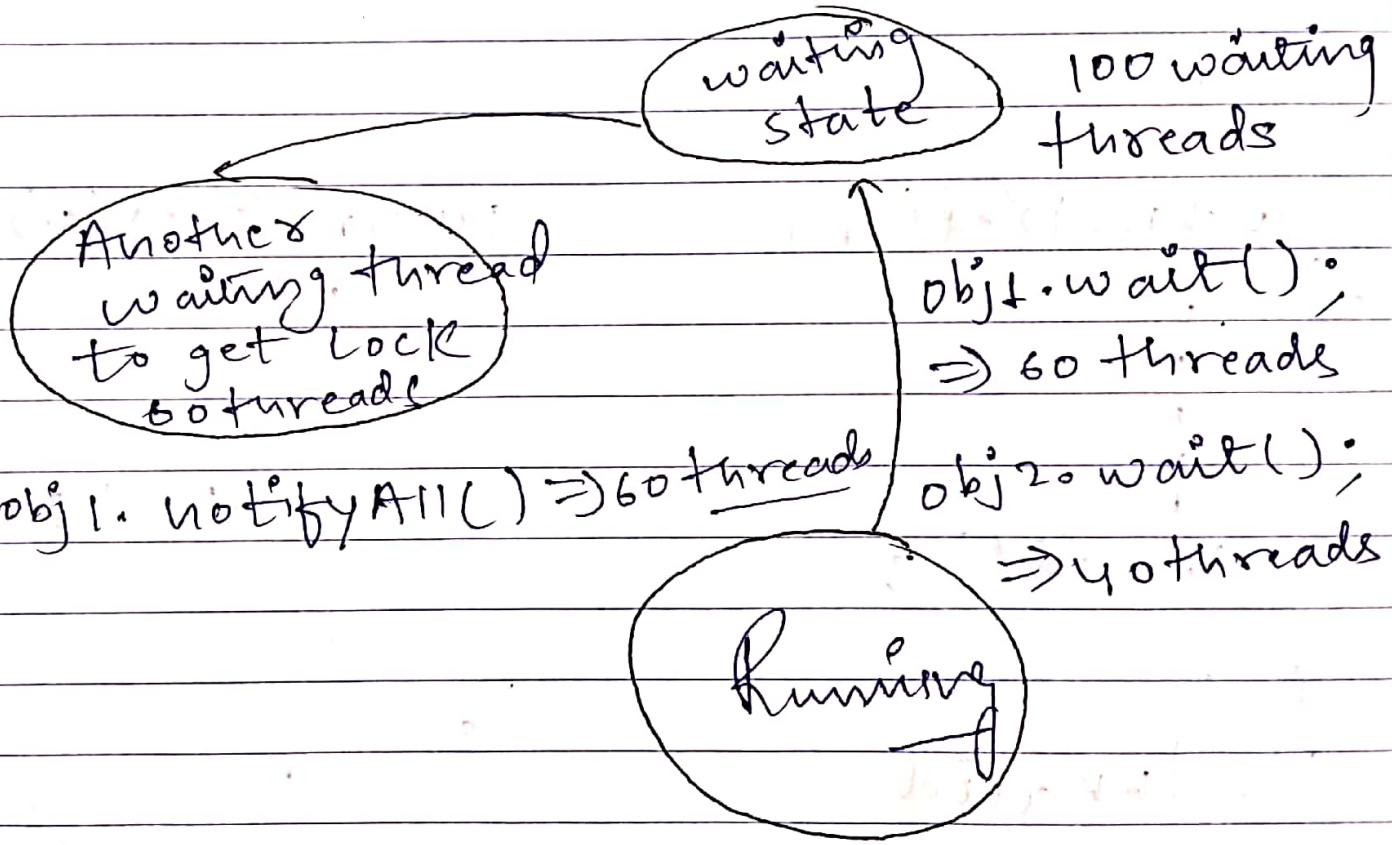
> ~~Prod~~After producing items to the Queue producer thread is responsible to call notify method then waiting consumer will get that notification and continue its execution with updated items.



Difference b/w notify() and notifyAll()

- We can use notify() to give the notification for only one waiting Threads; if multiple threads are waiting then only one thread will get notification and other threads have to wait to get further notifications.
- Which thread will be notified, we can't expect. it depends on JVM.

> We can use notifyAll() to give the notifications for all waiting threads of a particular object.
> Even though multiple threads are notified but execution will be performed one by one because threads required lock and only one lock is available.



> On which object we are calling wait method, thread required lock of that particular object.

Eg. stack s1 = new stack();
stack s2 = new stack();

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

Invalid

RE: IllegalMonitorException

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

Valid

DEADLOCK :

> If two threads are waiting for each other forever such type of infinite waiting is called deadlock.

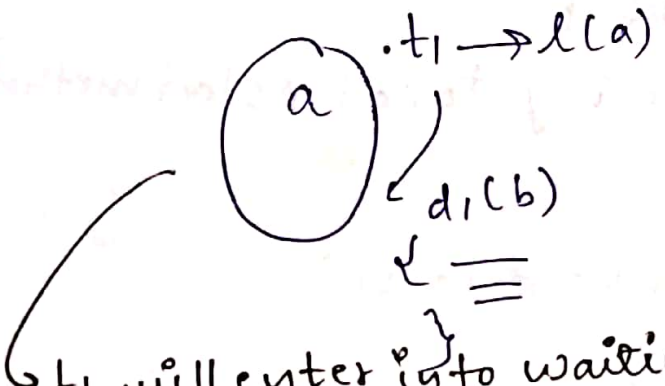
> Synchronized keyword is the only reason for deadlock situation, hence while using synchronized keyword we have to take special care.

> There are no resolution techniques for deadlocks but several prevention techniques are available.

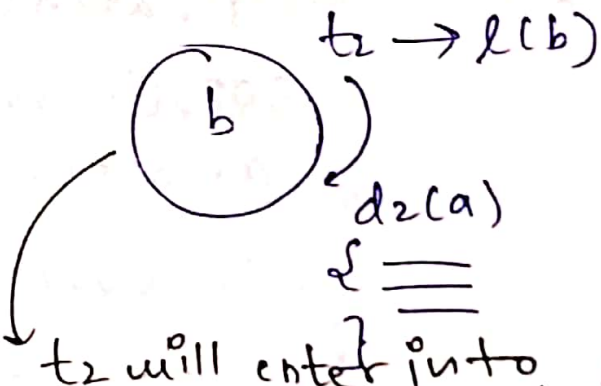
Scenario :

```
class A
{
  synch d1(B b)
  { b.last()
    ====
  }
  synch last()
  { ====
  }
}
```

```
class B
{
  synch d2(A a)
  { a.last
    ====
  }
  synch last()
  { ====
  }
}
```



t1 will enter into waiting state to get b's lock and currently t1 holds a's lock



t2 will enter into waiting state to get a's lock and currently holds b's lock.

Code -Very Imp.

```
class A
{
    public synchronized void d1(B b)
    {
        SOPn("Thread1 starts execution d1() method");
        try
        { Thread.sleep(2000);
        }catch(IE e){}
        SOPn("Thread1 trying to call B's last method");
        b.last();
    }
    public synchronized void last()
    {
        SOPn("Inside A, last() method");
    }
}
class B
{
    public synchronized void d2(A a)
    {
        SOPn("Thread2 starts execution of d2() method");
        try
        { Thread.sleep(5000);
        } catch(IE e){}
        SOPn("Thread2 trying to call A's last method");
        a.last();
    }
    public synchronized void last()
    {
        SOPn("Inside B, last() method");
    }
}
```

```
class Deadlock extends Thread
{
    A a=new A();
    B b=new B();
    public void m1()
    { this.start();
      a.d1(b);   ⟹ main Thread.
    }
    public void run()
    {
      b.d2(a);   ⟹ child Thread
    }
public static void main(String args[])
{
    Deadlock t=new Deadlock();
    t.m1();
    // t.start();  C.T.E
                   reference can't be
                   accessed directly in the
                   static class.
}
}
```

output -

Thread 1 starts execution of d1() method
Thread 2 starts execution of d2() method
Thread 2 trying to call A's last()
Thread 1 trying to call B's last()

In the above, if we remove one synchronized method, then program will not enter into Deadlock.

Hence, synchronized keyword is the only reason for deadlock situation.

Due to this while using synchronized keyword we have to take special care.

Deadlock Vs Starvation :

Long waiting of a thread where waiting never ends, such type of thing is deadlock.

Long waiting of a thread where waiting ends at certain point, is called Starvation.

Eg. 1 crore threads

1 Thread ⇒ priority -1

remaining Threads ⇒ priority ⇒ 10

low priority thread has to wait until completing all high priority, it may be long waiting but ends at certain point which is nothing but starvation.