Multi-Threading

1.) Introduction
2.) Ways to define a Thread
    i.) By extending Thread class
    ii.) By implementing Runnable(I)
3.) Getting & Setting Name of Thread
4.) Thread priorities
5.) The methods to prevent Thread execution
    i.) yield()
    ii.) join()
    iii.) sleep()
6.) Synchronization
7.) Inter-Thread Communication
8.) Deadlock
9.) Daemon Threads
10.) Multi-threading Enhancements.

(*) Introduction

≈> Multi-tasking :-

Executing several tasks simultaneously is a concept of Multi-tasking

There are two types of multi-tasking:-
1) Process Based Multitasking
2) Thread Based Multitasking

↝ Executing several tasks simultaneously, where each task is a separate independent program (process) is called "Process Based Multi-tasking"

Eg: while typing a Java program in the editor, we can listen audio songs from system at the same time we can download a file from Net. All this tasks will be executed simultaneously & independently from each other. Hence it is Process-Based Multi-tasking

- process based multi-tasking is best suitable at OS level.

↝ Executing several tasks simultaneously where each task is a separate independent part of same program is called "Thread Based Multitasking" and each independent part is called a thread.

- Thread Based multi-tasking is best suitable at Programmatic level.

↝ Whether it is Process Based (or) Thread Based, the main objective of multi-tasking is to reduce response time of system & to improve performance.

↝ The main important Areas of multi-threading are :-

1) To develop multi-media graphics
2) To develop Animations
3) To develop video games
4) To develop Web servers & Application servers etc.

↝ When compared with old languages developing multi-threaded Apps in Java is very Easy, Bez, Java provides in-built support for multi-threading with Rich API (Thread, Runnable, ThreadGroup...)

✳ Ways to define a Thread

↝ We can define a thread in the following two ways :-

1) By extending Thread class
2) By implementing Runnable (I)

(1) ⇒ By extending Thread class

Ex.

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            s.o.p("child Thread");
        }
    }
}
```

defines a Thread

Job of Thread

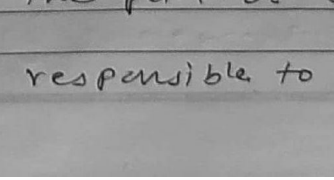
Executed by child Thread

```
class ThreadDemo
{
    p s v main(SA)
    {
        // Here Only main Thread Exist.
        MyThread t = new MyThread();
        t.start();
        for(int i=0; i<10; i++)
        {
            s.o.p("Main Thread");
        }
    }
}
```

main

instantiating child Thread by main Thread

Starting a Thread



Executed by Main Thread

★ Each Independent path / separate flow of a program is called a Thread

Note :- O/P of the above program is mixed, means it is not predictable, so when our program has two independent executable lines of code, then only we should go for Multithread.

If our program has dependency then we should not go for Multithreading

Case-1 :- Thread scheduler

⇝ It is the part of JVM.

⇝ It is responsible to schedule threads

i.e If multiple threads are waiting to get a chance of execution, then in which order threads will be executed is decided by Thread Scheduler.

→ we can't expect exact ~~behaviour~~ algorithm followed by Thread Scheduler, it is varied from JVM to JVM, hence we can't Thread execution order & exact o/p. Hence whenever situation comes to Multithreading there is no gaurantee for exact o/p, but we can provide several possible o/ps.

→ The following are various possible o/p for above program:-

P-1	P-2	P-3	.....
Main Thread Main Thread : : Child Thread Child Thread :	Child Th Child Th : : Main Th Main Th :	Main Th Child Th : :	

Case-4 :- Difference blw t.start() & t.run().

→ t.start() :-

In case of t.start(), a New Thread will be created which is responsible for the execution of run() method.

→ t.run() :-

In the case of t.run(), a New Thread won't be created & run() method will be executed just like a normal method call by Main Thread.

Hence in the above program if we replace t.start() with t.run(), then the o/p will be fixed only, as it is executed by main Thread only.

o/p :- Child Thread
Child Thread
:
Main Thread
Main Thread
:

} produced by only main Thread

Case 3 :- Importance of Thread class start() method.

⇒ Thread class start() method is responsible to Register the Thread with Thread scheduler & All other mandatory activities.

Hence, without executing Thread class start() method there is no chance of starting a new Thread in java, due to this Thread class start() method is considered as Heart of Multithreading.

```
start()
{
    (1.) Register this thread with Thread-scheduler
    (2.) Perform all other mandatory activities
    (3.) Invoke run();
}
```

Case-4 :- Overloading of run() method

⇒ Overloading of run() method is always possible but, Thread class start() method can invoke no-arg run() method, The other overloaded method we have to call explicitly like a normal method call.

Ex.

```
class MyThread extends Thread
{
    Overloading of run()
    public void run()
    {
        S.O.P("No-arg run");
    }
    public void run(int i)
    {
        S.O.P("int-arg run");
    }
}
class ThreadDemo
{
    p s v main(Sr)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

o/p :- No-arg run.

Case-5 :- If we are Not overriding run()

⇒ 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 o/p.

Ex.

```
class MyThread extends Thread
{
}
class Test
{
    p s v m (Str
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

No o/p

Note :- It is Highly recommended to override run() method, otherwise don't go for multi-threading concept

Case-6 :- Overriding of start() method.

⇒ If we override start() method, then our start() method will be executed just like normal method call & New thread won't be created

Ex.

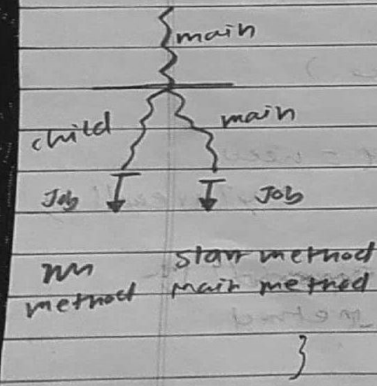
```
class MyThread extends Thread
{
    public void start()
    {
        s.o.p ("start Method");
    }
    public void run()
    {
        s.o.p ("run method");
    }
}
class Test
{
    p s v m (Str
    {
        MyThread t = new
                     MyThread();
        t.start();
        S.o.P("main method");
    }
}
```

O/P :- start Method
main Method
} Produced only by Main Thread.

⇝ It is Not recommanded to override start() method, otherwise don't go for Multi-threading concept.

Ex.2

```
class MyThread extends Thread
{
    public void start()
    {
        super.start();
        s.o.p("start method");
    }
    public void run()
    {
        s.o.p("run method");
    }
}
class Test
{
    p s v m(Str m)
    {
        MyThread t = new
                     MyThread();
        t.start();
        s.o.p("main method");
    }
}
```

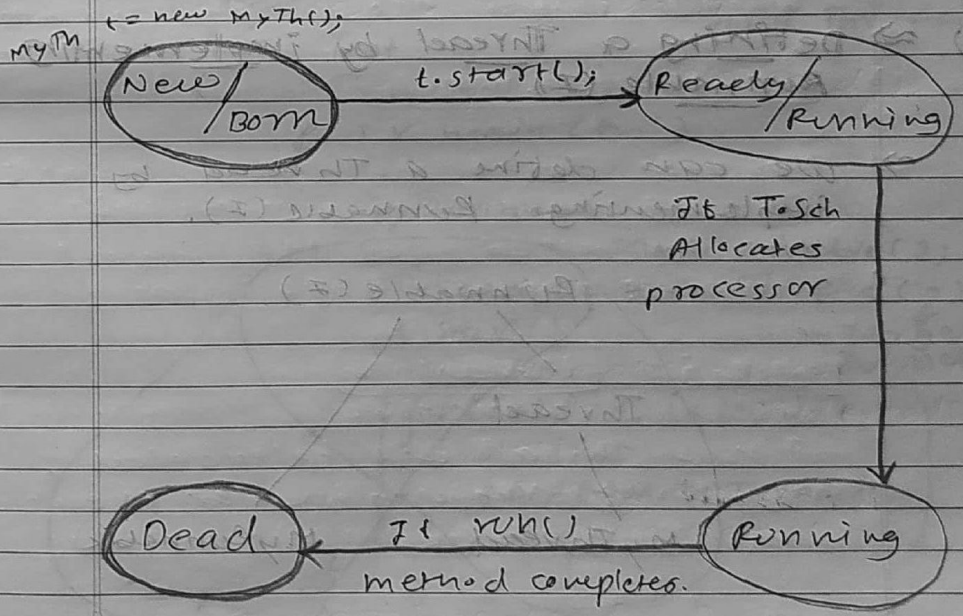


O/P :-

P-1	P-2	P-3
run method	start method	start method
start method	main method	run method
main method	run method	main method

Case-7 Thread Life-Cycle

⇝ simple life-cycle (Partial) of Thread is as shown below :-



Case-8 After starting a Thread if we are trying to restart the same Thread, then we will get (R.E:)

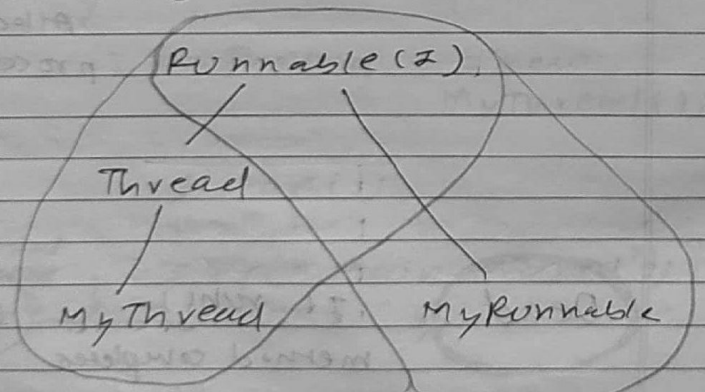
Ex:

```
MyThread t = new MyThread();
t.start();
.....
t.start();
```

R.E: IllegalThreadStateException

(2) ⇒ Defining a Thread by implementing Runnable (I)

→ we can define a Thread by implementing Runnable (I).



Approach-1 Approach-2

~ Runnable (I) present in java.lang package & it contains one method run().

Ex:

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

Define a Thread

Job of Thread

executed by child Thread

```
class Test
{
    psvm(String[] args)
    {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);
        t.start();
        for (int i=0; i<10; i++)
        {
            s.o.p("Main Thread");
        }
    }
}
```

Target Runnable

executed by Main Thread

⇒ we will get Mixed o/p, we can't tell exact o/p.

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 which is responsible for the execution of Thread class run() method, which has Empty Implementation.

case-2 :- `t1.run();`

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

case-3 :- `t2.start();`

A new Thread will be created which is responsible for execution of MyRunnable run() method.

case-4 :- `t2.run();`

New Thread won't be created & MyRunnable run() method will be executed just like Normal method.

case-5 :- `r.start();`

we will get C.E: MyRunnable class doesn't have start capability.

(C.E: cannot find symbol
s :- method start()
l :- class MyRunnable.)

case-6 :- `r.run();`

No new Thread will be created & MyRunnable run() method will be executed like Normal method call.

Q. Which Approach is best to define a Thread?

Ans:- Among two ways of defining a Thread, implements Runnable Approach is recommended.

In 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 2nd Approach, while implementing Runnable(I), we can extend any other class hence we won't miss any inheritance benefit

~ Bcz of above reason, implementing Runnable (I) approach is Recommanded than extending Thread class.

✳ 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, Runnable r);

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

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 Recommanded to use)

```
class MyThread extends Thread
{
    public void run()
    {
        S.O.P("child Th");
    }
}
class Test
{
    P S V main(Str[] )
    {
        MyThread t = new
                     MyThread();
```

Just another way to start a Thread. (Hybrid Approach)

```
        Thread t1 = new Thread(t);
        t1.start();
```

```
        S.O.p("main Thread");
    }
}
```

O/P :

child Th	main Th
main Th	child Th

Getting and Setting Name of 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 & set name of a thread by using the following two methods of Thread class.

1) public final String getName()
2) public final void setName(String name)

Ex:

```
class MyThread extends Thread
{
}
```

```
class Test
{
    public static void main(str...)
    {
        S.O.p(Thread.currentThread().
                getName());

        MyThread t = new MyThread();

        S.O.p(t.getName());

        Thread.currentThread().
                setName("Abc");

        S.O.p(Thread.currentThread().
                getName());

        S.O.p(10/0);
    }
}
```

O/P: main
Thread-0
Abc
Exception in Thread "Abc": divide by zero
j.l.ArithmeticException.

Note: we can get current executing Thread object by using Thread.currentThread() method.

Ex:

```
class MyThread extends Thread
{
    public void run()
    {
        S.O.P("run method executed by
                Thread: " +
            Thread.currentThread().getName());
    }
}
class Test
{
    p s v m(Str)
    {
        MyThread t = new MyThread();
        t.start();

        S.O.P("main method executed by
                Thread: " +
            Thread.currentThread().getName());
    }
}
```

O/P:

run method executed by Thread: Thread-0
main method executed by Thread: main

(one of the possibility)

✱ Thread Priorities

→ Every thread in java has some priority, it may be default priority generated by JVM (or) customized priority provided by programmer

→ The valid range of Thread priorities is 1 to 10.

where

1 = Min Priority and

10 = Max Priority

→ Thread class defines the following constants to represents some standard priorities

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

Q. Which are the valid priorities ?

A.) 0 ✗
B.) 1 ✓
C.) 10 ✓
D.) Thread.LOW_PRIORITY ✗
E.) Thread.HIGH_PRIORITY ✗
F.) Thread.MIN_PRIORITY ✓
G.) Thread.NORM_PRIORITY ✓

~> Thread scheduler will use priorities while Allocating processors.

~> The Thread which is having highest priority will get the chance first.

~> If Two Threads having same priority then we can't expect exact execution order, it depends on Thread scheduler

~> Thread class defines the following methods to get & set priority of a Thread.

① public final int getPriority()

② public final void setPriority(int p)

Allowed values Range : 1 to 20

Otherwise we will get R.E.

Ex:

t.setPriority(7); ✓

t.setPriority(17); ✗

R.E : IllegalArgumentException

Default Priority

~> The default priority only for the "main Thread" is 5.

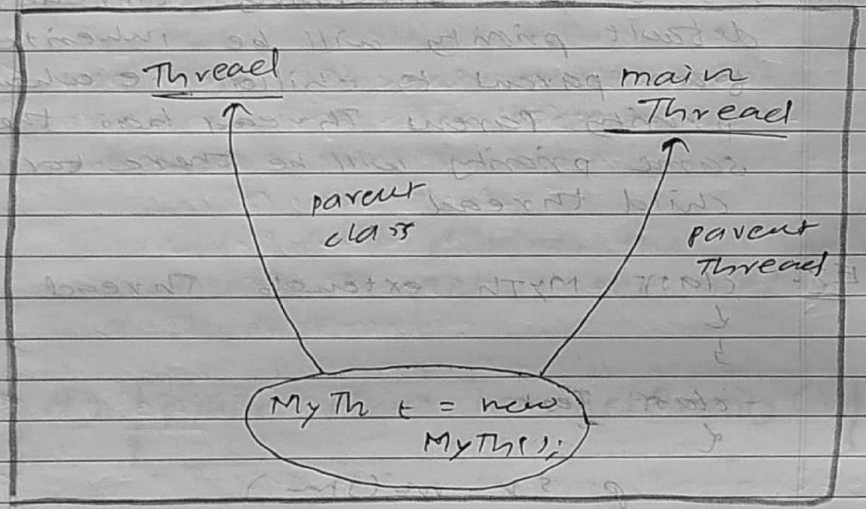
~> But for all remaining threads default priority will be inherited from parent to child i.e whatever priority parent Thread has the same priority will be there for the child thread.

Ex.

```
class MyTh extends Thread
{
}
class Test
{
    p s v m(Str)
    {
        S.O.p(Thread.currentThread().
              getPriority()); //5

        //Thread.currentThread().
              setPriority(15); R.E. IIAE

        Thread.currentThread().setPriority(8);

        MyTh t = new MyTh();
```

```
        S.O.P(t.getPriority()); //8
    }
}
```

↳ If we comment line-① , then the child Thread priority will become (5).



Ex:

```
class MyTh extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            S.O.P("child Thread");
        }
    }
}
```

```
class Test
{
    psvm(String[] args)
    {
        MyTh t = new MyTh();
        t.setPriority(10);  ---- line①
        t.start();

        for(int i=0; i<10; i++)
        {
            S.O.P("main Thread");
        }
    }
}
```

O/P :-

```
child Thread
child Thread
    :
 (10 times)
main Thread
main Thread
    :
 (10 times)
```

only ① possible O/P.

(If your underlying OS support priority)

Note :- If we comment line-① we can't predict O/P bcz, both thread will have same priorities (5).

Note: Some platforms won't provide proper support for Thread priorities

(*) The methods to prevent Thread Execution

→ we can prevent a thread execution by using the following methods:-

1) yield()
2) join()
3) sleep()

(1) ⇒ yield()

→ yield() method causes to pause current executing thread to give the chance for waiting threads of same priority. if there is no waiting thread (or) All waiting threads have low priority then same thread can continue its execution.

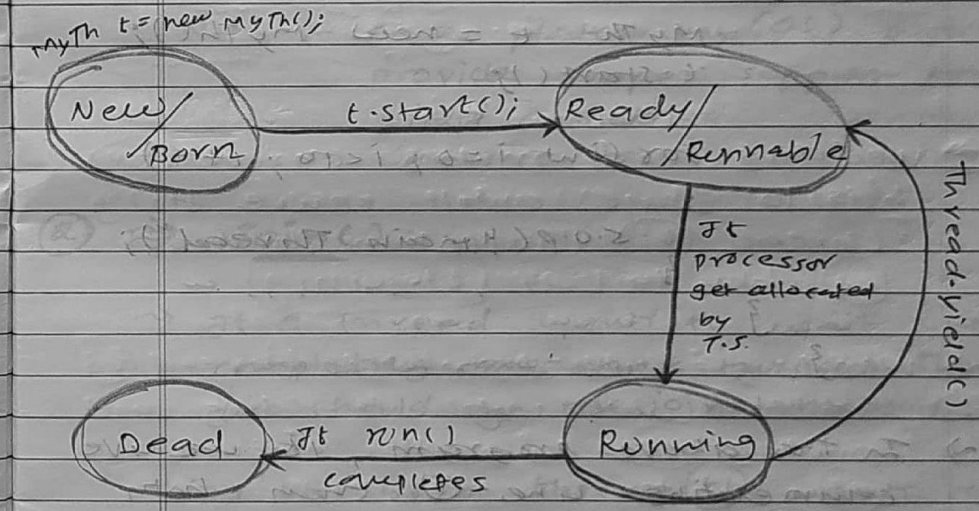
→ If multiple threads are waiting with same priority then which waiting thread will chance, we can't expect that ? it depends on Thread scheduler.

→ The Thread which is yielded, when it will get chance once again?, it depends on Thread-scheduler & we can't expect exactly.

→

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

→ Effect of yield() on life-cycle of Thread.



→ Ex:

```
class MyTh extends Thread
{
```

```
public void run()
{
    for(int i=0; i<10; i++)
    {
        S.O.P("Child Thread");
        Thread.yield(); — line①
    }
}

class ThreadYieldDemo
{
    p s v m(Str)
    {
        MyTh t = new MyTh();
        t.start();
        for(int i=0; i<10; i++)
        {
            S.O.P("main Thread");
        }
    }
}
```

⇒ In the above program, if we are commenting line-① then both threads will be executed simultaneously & we can't expect which thread will complete first.

⇒ If we are not commenting line ① then child Thread always calls yield() method bcz of that main Thread will get chance more no. of times and the chance of completing main Thread first is high.

⇒ The thread which requires more processing time, in the middle that thread is recommended to call yield() method

> Note :- some platforms (OS) won't provide proper support for yield() method.

② ⇒ join()

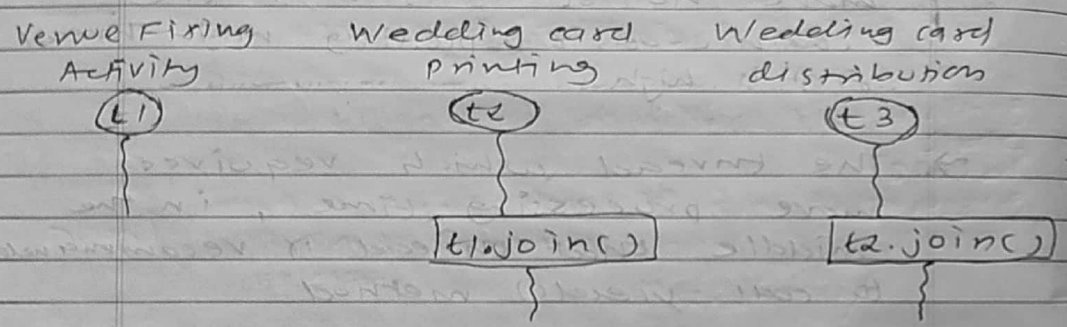
⇒ If a thread wants to wait until completing some other thread then we should go for join() method.

⇒ For ex:- If a thread (T1) wants to wait until completing (T2) then (T1) has to call T2.join()

⇒ If (T1) executes T2.join() then immediately (T1) will be entered into waiting state until (T2)

completes, once t2 completes then t1 can continue its execution

Ex:-



→ Wedding card printing thread (t2) has to wait until venue fixing thread (t1) completion, hence (t2) has to call t1.join()

→ Wedding card distribution thread (t3) has to wait until Wedding card printing (t2) completion, hence (t3) has to call t2.join()

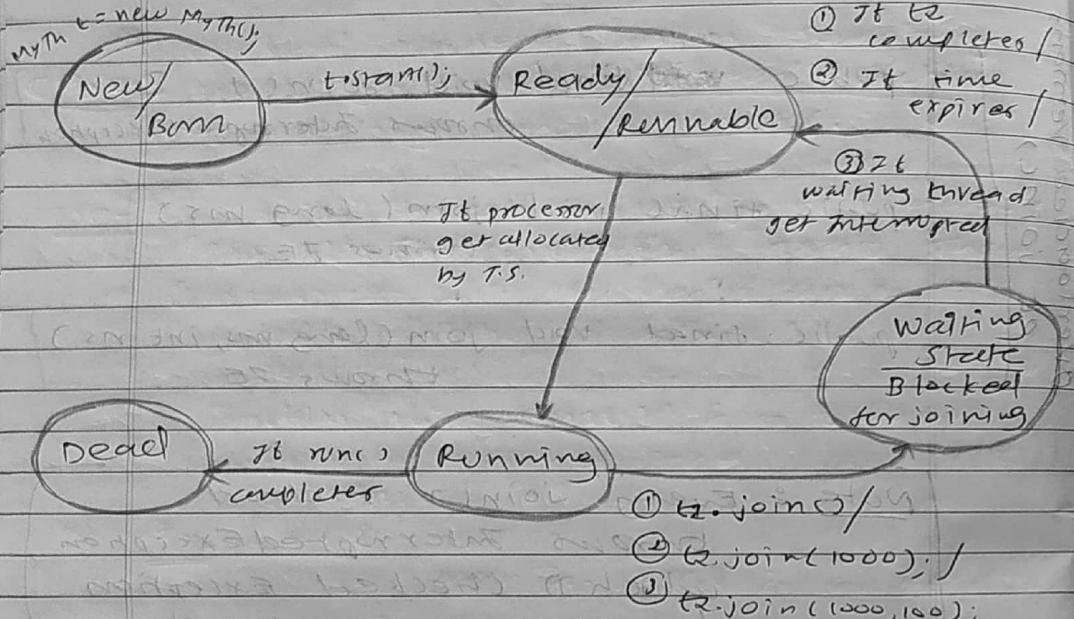
→ join() methods has overloaded versions as below:-

Overloaded versions of join()

- public final void join() throws InterruptedException
- public final void join(long ms) throws IE
- public final void join(long ms, int ns) throws IE

Note: Every join() method throws InterruptedException, which is checked Exception hence, compulsory we should handle this exception either by using try-catch (or) by throws keyword otherwise we will get C.E.

→ The impact of join() method in the life-cycle of thread is as following:-



Ex.

```
class MyTh extends Thread
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            s.o.p("seetha Th");
            try
            {
                Thread.sleep(2000);
            }
            catch(InterruptedException e)
            {
            }
        }
    }
}
```

```
class ThreadJoinDemo
{
    p s v m(str) throws IE
    {
        MyTh t = new MyTh();
        t.start();
        t.join();          — Line ①
        for(int i=0;i<10;i++)
        {
            s.o.p("Rama Th");
        }
    }
}
```

⇒ If we comment Line ① then both main & child Th will be executed simultaneously & we can't expect exact o/p.

⇒ If we not comment Line ① then main Thread will call join() on child Th object hence main Th will wait until completing child Th. In this case o/p :-

```
seetha Th.
(2 sec)
seetha Th.
(2 sec)
seetha Th
:
10 times
Rama Th.
Rama Th
:
10 times
Rama Th.
```

case 2 waiting of child Thread until completing main Thread.

```
class MyTh extends Thread
{
    static Thread mainTh;

    public void run()
    {
        try
        {
            mainTh.join();
        }
        catch(IE e)
        {
        }
        for(int i=0; i<10; i++)
        {
            s.o.p("child Th");
        }
    }
}
class Test
{
    p s v m (Str) throws IE
    {
        MyTh.mainTh = Thread.
                      currentThread();
        MyTh t = new MyTh();
        t.start();

        for(int i=0; i<10; i++)
        {
            s.o.p("main Th");
            Thread.sleep(2000);
        }
    }
}
```

O/P :-

```
main Th
main Th
    : (10 times)
    :
child Th
child Th
    : (10 times)
```

→ In the above example, child Thread calls join on main Thread object hence, child Thread has to wait until main Thread completion.

case 3 :- If main Thread calls join() method on child Thread object & child Thread calls join() method on main Thread Object then both threads will wait forever, so program will be

stucked / paused (This is something like deadlock).

case-4 If a thread calls join() method on the same thread itself, then program will stucked (This is something like deadlock).

In this case thread have to wait Infinite amount of time.

Ex.

```
class Test
{
    p s v m(Str..) throws IE
    {
        Thread.currentThread().join();
    }
}
```

(3) ⇒ sleep()

⇝ If a thread don't want to perform any operation for particular amount of time then we should go for sleep method.

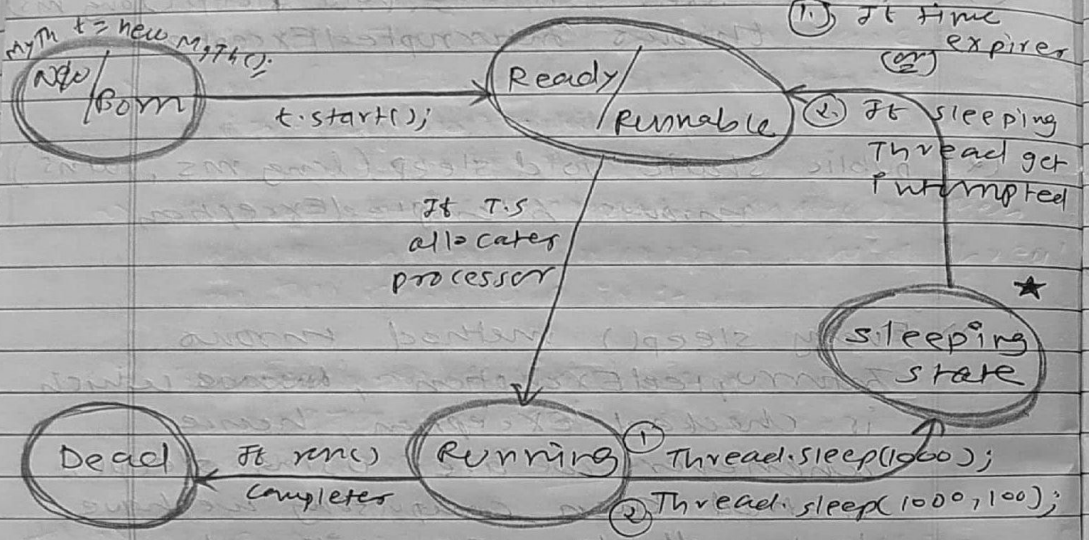
(1) `public static native void sleep(long ms) throws InterruptedException`

(2) `public static void sleep(long ms, int ns) throws InterruptedException`

⇝ Every sleep() method throws InterruptedException, which is checked Exception, hence whenever we are using sleep() method then compulsory we have to handle the IE either by try-catch or by throws keyword, otherwise we will get C.E.

⇝ Effect of sleep() method in the Thread life-cycle is as shown below:-

Ex.
```
class SlideRotator
{
    p s v m(Str..)
        throws IE
    {
        for(int i=0; i<10; i++)
        {
            s.o.p("slide-"+i);
            Thread.sleep(2000);
        }
    }
}
```



O/P:-

slide-1
(After 5 sec)
slide-2
⋮
slide-10

Q: How a thread can interrupt another thread?

Ans:→ A thread can interrupt a sleeping thread (or) waiting thread by using interrupt() method of Thread class.

→ `public void interrupt()`

Ex:

```
class MyTh extends Thread
{
    public void run()
    {
        try
        {
            for(int i=0; i<10; i++)
            {
                s.o.p("Lazy Thread");
                Thread.sleep(2000);
            }
        }
        catch(InterruptedException e)
        {
            s.o.p("Interrupted");
        }
    }
}
class Test
{
    p s v m(str)
    {
        MyTh t = new MyTh();
        t.start();
        t.interrupt();   → Line-①
        s.o.p("End of main");
    }
}
```

⇝ If we comment line-① then main Thread won't interrupt child Thread. In this case child Thread will execute for-loop to timer

⇝ If we not comment line-① then main Thread interrupts child Thread, In this case o/p is :-

End of main
Lazy Thread
Interrupted

★ Note :- Whenever we are calling interrupt() method, If the target thread is not in sleeping state/waiting state then there is no impact of interrupt() call immediately, interrupt() call will be waited until target thread entered into sleeping/waiting state.

⇝ If the target thread entered into sleeping/waiting state, then immediately interrupt() call will interrupt the target thread.

⇝ If target thread never entered into sleeping/waiting state in its lifetime then there is no impact of interrupt() call. This is only case where interrupt call will be wasted.

Ex.

```
class MyTh extends Thread
{
    public void run()
    {
        for(int i=0; i<10000; i++)
        {
            s.o.p("Lazy Th-" +i);
        }
        s.o.p("I want sleeping state");
        try
        {
            Thread.sleep(10000);
        }
        catch(Exception e)
        {
            s.o.p("Interrupted");
        }
    }
}
```

```
class Test
{
    p s v m (Str[] )
    {
        MyTh t = new MyTh();
        t.start();
        t.interrupt();
        S.o.p("End of main Th");
    }
}
```

⇒ In above example, interrupt() call waited until child Th completes for-loop 10000 times.

⇒

property	yield()	join()	sleep()
① purpose	If a thread want to pause its execution to give the chance for remaining threads of same priority then we should go for yield() method	If a thread wants to wait until completing some other thread, then we should go for join() method.	If a thread don't want to perform any operation for some amount of time, then we should go for sleep() method.

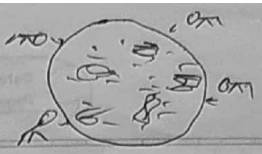
② Is it overloaded?	NO	Yes	Yes
③ Is it final?	NO	Yes	NO
④ Is it throws IE?	NO	Yes	Yes
⑤ Is it native?	Yes	NO	sleep(long) (Yes) sleep() (NO)
⑥ Is it static?	Yes	No	Yes

✱ Synchronization

⇒ Synchronized is the modifier applicable only for methods & blocks but not for classes & variables.

⇒ If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of Data Inconsistency problem.

⇒ For synchronization, Real-time example can be Biryani 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 synchronized keyword is we can resolve D.I. problems but the main Disadvantage of synchronized keyword is it increases the waiting time of thread & creates performance problems. Hence if there is no specific requirement then it's not recommended to use synchronized keyword.

⇒ Internally Synchronization concept is implemented by using lock every object in java has unique lock, whenever we are using synchronized keyword then only lock concept will come into picture.

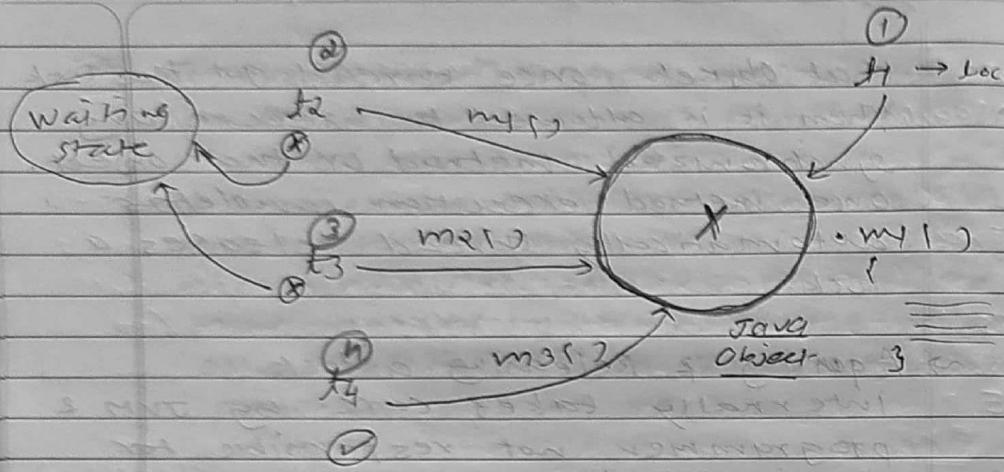
⇒ If a thread wants to execute synchronized method on the given Object, first it has to get lock of that Object, once thread got the lock then it is allowed to execute any synchronized method on that object. once method execution completes automatically thread releases a lock.

⇒ Acquiring & Releasing a lock internally taken care by JVM & programmer not responsible for this activity.

⇒ while a thread executing synchronized method on a given object, the remaining threads are not allowed to execute any synchronized method simultaneously on the same object, but remaining threads are allowed to execute non-synchronized methods simultaneously.

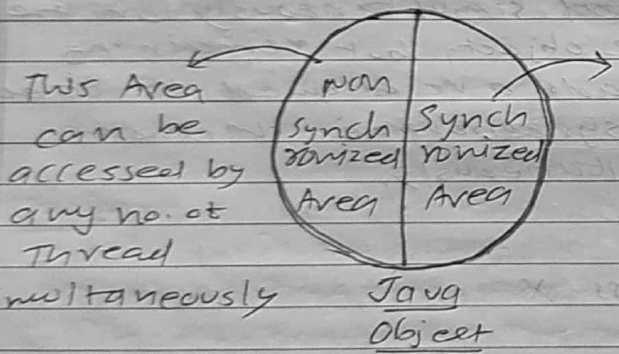
Ex.

```
class X
{
    sync m1()

    synch m2()

    m3()
}
```



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

⇒



Ex. ⇒ class X
{

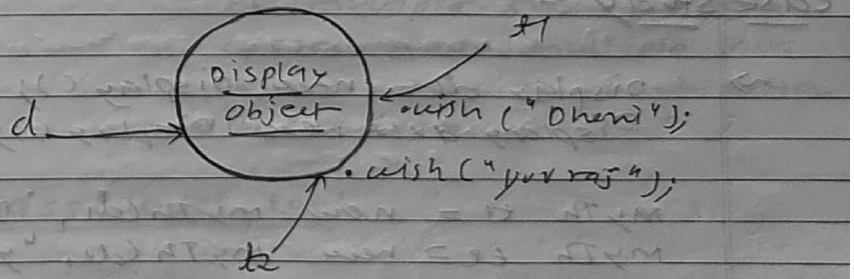
Synchronized Area
{
where ever we are performing update operation [Add/delete/replay]
i.e where state of object is changing
}
Non-synchronized Area
{
where ever object state won't be changed
i.e like read operation
}
}

Ex.

class ReservationSystem
{
Non-synchronized checkAvailability()
{
== (Just Read operation)
}
Synchronized bookTicket()
{
== (Update)
}
}

Ex. class Display
{
public synchronized void
wish (String name)
{
for(int i=0; i<10; i++)
{
s.o.p("Good Morning:");
try
{
Thread.sleep(2000);
}
catch(IE e)
{
}
s.o.p(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 SynchronizationDemo
{
p.s.v m(Str[])
{
Display d = new Display();
MyThread t1 = new
MyThread(d, "Dhoni");
MyThread t2 = new
MyThread(d, "yuvraj");
t1.start();
t2.start();
}
}



⇝ If we declare wish() method as synchronized then both thread will be executed simultaneously & hence we will get irregular o/p.

⇝ If we declare wish() method as synchronized then at a time only one thread is allowed to execute wish() method on the given display object, hence we will get regular o/p.

o/p :-

Good Morning : Dhoni
10 times

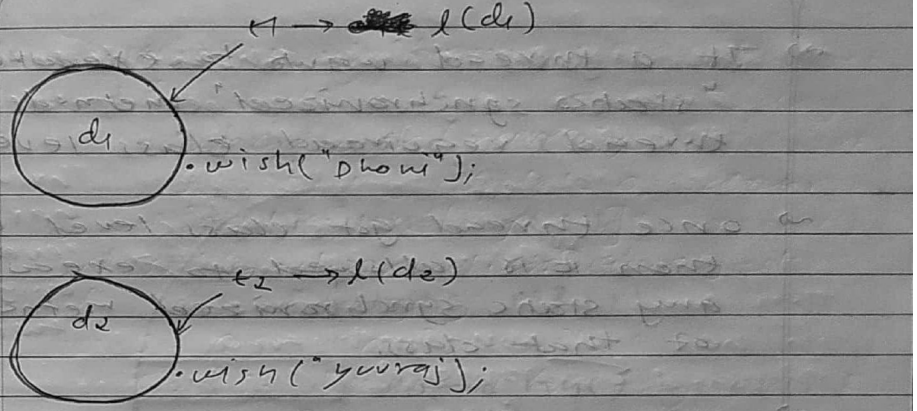
Good morning : yuvraj
10 times

Case study :-

⇝ Display d1 = new Display();
Display d2 = new Display();

MyTh t1 = new MyTh(d1, "Dhoni");
MyTh t2 = new MyTh(d2, "yuvraj");

t1.start();
t2.start();

⇝ 

⇝ Even though wish() method is synchronized, we will get irregular o/p, bcz threads are operating on different java objects.

⇝ Conclusion :- If multiple threads are operating on same java object then synchronization is required.

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

★ ⇝ class-level lock

⇝ Every class in java has a unique lock, which is nothing but class-level lock.

⇝ If a thread wants to execute "static synchronized" method then thread required class level lock.

⇝ once thread got class level lock then it is allowed to execute any static synchronized method of that class.

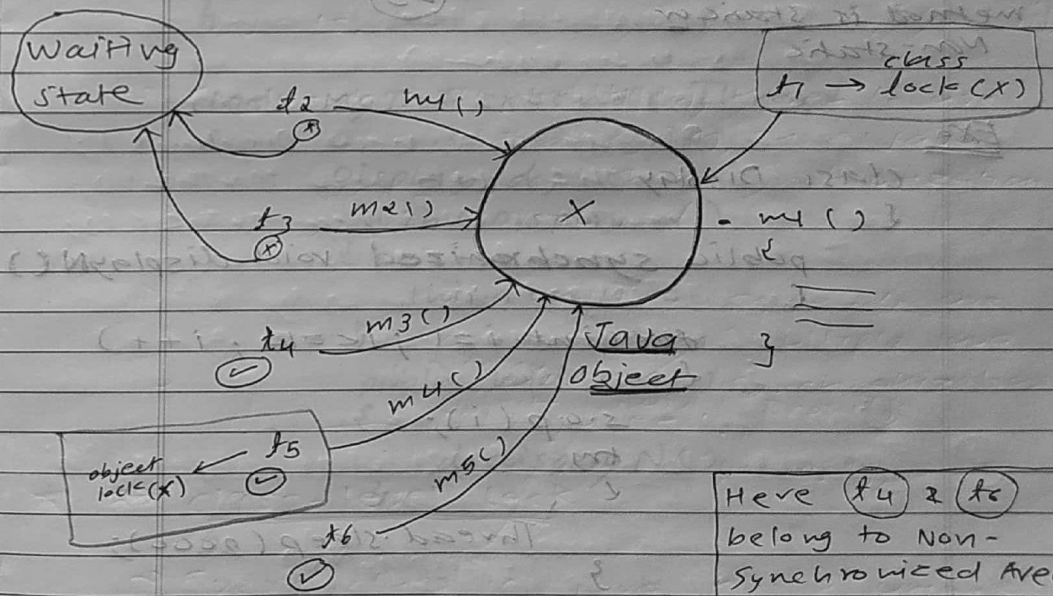
⇝ once method execution completes automatically thread releases a lock.

⇝ while a thread executing static synchronized method, 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) Synchronized Instance methods
3) ~~Synchronized~~ Normal Instance methods

EX.

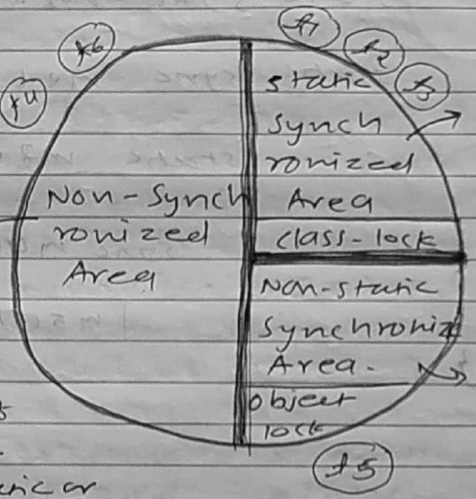
```
class X
{
    static sync m1()
    {
    }
    static sync m2()
    {
    }
    static m3()
    {
    }
    sync m4()
    {
    }
    m5()
    {
    }
}
```



Here t4 & t6 belong to Non-Synchronized Area so can be Accessed by multiple threads simultaneously

t5 only need Object level lock which is available, so it can be execute as well

⇒ The figure which shows different area for object can be updated like ...



This Area can be Accessed by multiple threads simultaneously irrespective of whether the method is static or Non-static

Thread which has class-level lock can only execute this area

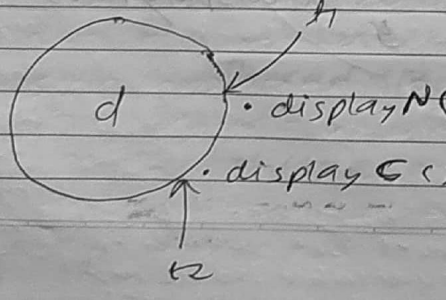
Thread which has object-level lock can only execute this area.

Ex-

```
class Display
{
    public synchronized void displayN()
    {
        for(int i=1; i<=10; i++)
        {
            s.o.p(i);
            try
            {
                Thread.sleep(2000);
            }
            catch(IE e)
            {
            }
        }
    }
```

```
    public synchronized void displayC()
    {
        for(int i=65; i<75; i++)
        {
            s.o.p((char)i);
            try
            {
                Thread.sleep(2000);
            }
            catch(IE e)
            {
            }
        }
    }
}

class MyThI extends Thread
{
    Display d;
    MyTh(Display d)
    {
        this.d=d;
    }
    public void run()
    {
        d.displayN()
    }
}
```

```
class MyTh2 extends Thread
{
    Display d;
    MyTh2(Display d)
    {
        this.d = d;
    }
    public void run()
    {
        d.displayC();
    }
}

class Test
{
    p s v m(sa)
    {
        Display d = new Display();
        MyTh1 t1 = new MyTh1(d);
        MyTh2 t2 = new MyTh2(d);

        t1.start();
        t2.start();
    }
}
```



⇝ The Thread which get lock first will execute first as a whole.

O/P :

possibility1	possibility2
1 2 3 4 5 6 7 8 9 10 A B C D E F G H I J	A B C D E F G H I J 1 2 3 4 5 6 7 8 9 10

⇛ Synchronized Block

⇝ If very few lines of the code requires synchronization then it is not recommanded to declare entire method as synchronized, we have to enclose those few lines of the code by using synchronized block.

⇝ The main Advantage of synchronized block over synchronized method is it reduces waiting time of threads & improves performance of system.

Real-time example :

Bridge in-between can have only 1 vehicle at a time

so, blocking all other vehicles from starting is not good practice.

we have to block only that bridge

⇝ we can declare synchronized block as follows :-

① To get lock of current object

```
Synchronized (this)
{
    ~~~
    ~~~
}
```

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

② To get lock of particular object 'b'

```
synchronized (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

```
synchronized (Display.class)
{
    ~~~
    ~~~
}
```

If a thread get class level lock of 'Display' class, then only it is allowed to execute this Area.

Ex.

```
class Display
{
    public void wish(String name)
    {
        ;;;;;;; // 1 lakh line of code

        synchronized (this)
        {
            for(int i=0; i<10; i++)
            {
                S.O.P("Good morning:");
                try
                {
                    Thread.sleep(2000);
                }
                catch(IE e)
                {}
```

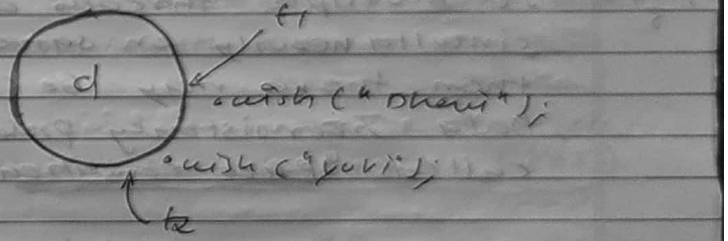
```
        s.o.p(name);
    }
}
;;;;;;;;;; // 2 lakh line of code
    }
}

class MyTh extends Thread
{
    Display d;
    String name;

    MyTh (Display d, String name)
    {
        this.d = d;
        this.name = name
    }

    public void run()
    {
        d.wish(name);
    }
}

class Test
{
    p s v m(strm)
    {
        Display d = new Display();
        MyTh t1 = new MyTh(d, "Dhoni");
        MyTh t2 = new MyTh(d, "yuvi");
        t1.start();
        t2.start();
    }
}
```

o/p :- Regular o/p

```
Good Morning: Dhoni
} to timer
Good Morning : yuvi
} to timer
```

one of the possibility (yuvi can come first also)

~> so, these 2 lakh lines of code doesn't affect performance as they can be executed by multiple threads simultaneously.



~> lock concept applicable only for object type & class types but Not for primitives, hence we can't primitive type as argument to synchronized block otherwise we will get C.E.

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

Unexpected type
found : int
required : reference

⇒ FAQs :-

1) What is synchronized keyword, where we can Apply?

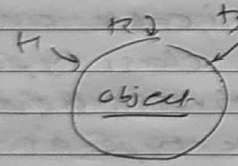
2) Explain Advantage of synchronized keyword.

3) Explain DisAdvantage of synchronized keyword.

4) What is Race condition?

Ans: If multiple threads are operating simultaneously on same java object then there may be chance of Data Inconsistency problem, this is called "Race condition".

We can overcome this problem by synchronized keyword.



5) What is object lock, when it is required?

6) What is class level lock, when it is required?

7) What is the difference blw class level lock & object level lock?

8) while a thread executing a synchronized method on the given object is the remaining threads are allowed to execute any other synchronized method simultaneously on the same object? NO

9) What is synchronized block?

10) How to declare sync block, to get lock of current object, to get class level lock?

11) What is Advantage of sync block over sync method?

★ 12) If a thread can aquire multiple locks simultaneously? (Yes)

Ans: yes, of course from different objects.

Ex.

```
class X
{
    public synchronized void m1()
    {
        // Here thread has lock
        of 'X' object

        Y y = new Y();
```

```
synchronized (Y)
{
  // Here thread has locks of
     "X" and "Y"

  Z z = new Z();

  synchronized(Z)
  {
     // Here thread has locks of
        "X", "Y" and "Z"
     ≃
  }
 }≃
}
```

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

★ 13) What is synchronized statement? (Interview people created terminology)

Ans: The statements present in synchronized method or in synchronized block are called synchronized statements.

⊛ Inter-thread Communication

⇝ Two threads can communicate with each other by using wait(), notify() and notifyAll() methods.

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

⇝ The Thread which is responsible to perform updation. After performing updation it is responsible to call notify() method, then waiting thread will get notification & continue its execution with those updated items.

★ ⇝ wait(), notify(), notifyAll() methods present in Object class but not in Thread class, bcz thread can call this methods on any java object.

Other methods like start(), join() are in Thread class bcz we can call them only on Thread object, Any other java object can't call there methods.

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

⇝ Hence, we can call wait(), notify() & notifyAll() methods only from synchronized Area, otherwise we will get R.E.

IllegalMonitorStateException.

⇝ If a thread calls wait() method on any object it immediately releases a lock of that object & enter into waiting state.

⇝ If a thread calls notify() method on any object it releases the lock of that object but may not immediately.

⇝ Except wait(), notify() and notifyAll() methods, there is no other method where thread releases a lock.

Method	Is thread releases lock?
yield()	NO
join()	NO
sleep()	NO
wait()	Yes
notify()	Yes
notifyAll()	Yes

Q. Which of the following is valid?

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

(2) If a thread calls wait() method it releases lock of that object but may not immediately. (X)

(3) If a thread calls wait() method on any object it releases All locks aquired by that thread & immediately enter into waiting thread. (X)

(4) If a thread calls wait() method on

any object it immediately releases the lock of that particular object & enter into waiting state. ✓

(5) If a thread calls notify() method on any object, it immediately releases the lock of that particular object. ✗

(6) If a thread calls notify() method on any object, it releases the lock of that object but may not immediately. ✓

⇒ signatures:-

(1) public final void wait() throws InterruptedException

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

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

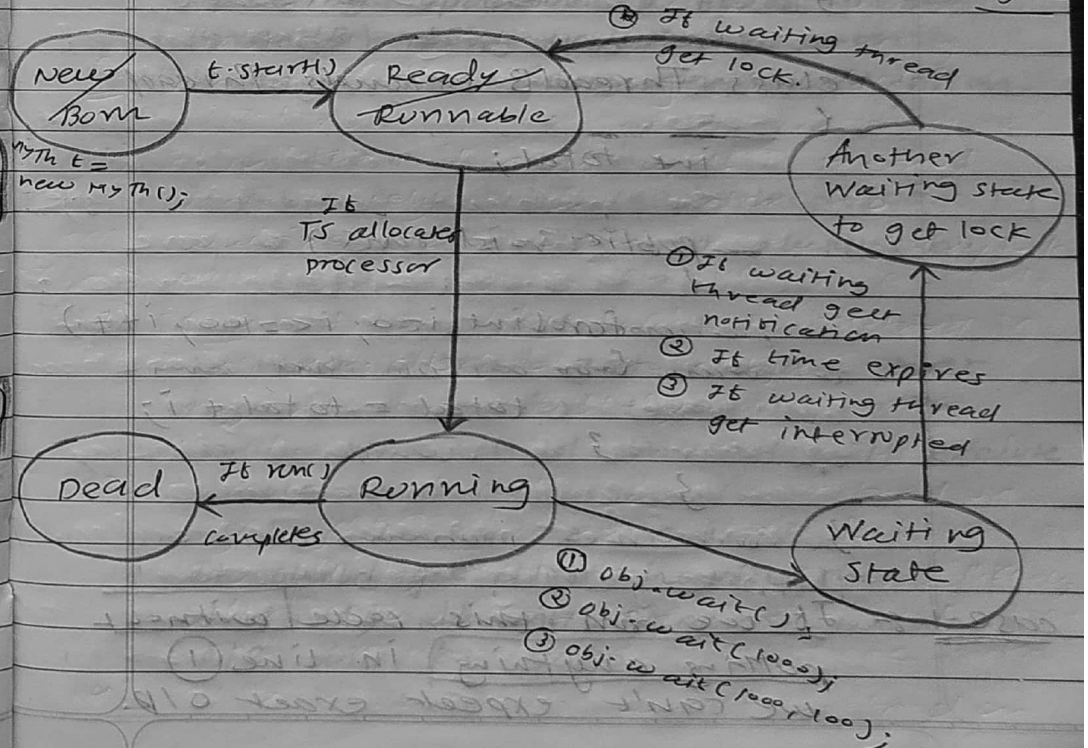
(4) public final native void notify()

(5) 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 IE either by try-catch or by throws keyword otherwise we will get C.E

⇒ one wait() method is native

⇒ Impact of these methods on Thread Life Cycle



Ex.

```
class ThreadA
{
    s v m(String[] args) throws IE
    {
        ThreadB b = new ThreadB();
        b.start();

        [          ]  ⟶ Line (1)

        S.O.P(b.total);
    }
}

class ThreadB extends Thread
{
    int total;

    public void run()
    {
        for(int i=0; i<=100; i++)
        {
            total = total + i;
        }
    }
}
```

case-1 :- If we run this code without putting anything in line (1) we can't expect exact O/P.

Possibility-1 ⇒ 0 (If main thread got chance first)

possibility-2 ⇒ 5050 (If child thread got chance first)

possibility-3 ⇒ 0<ans<5050 (If child got chance first but after half execution main got chance)

(1) -so, here main thread (ThreadA) expecting updation & child thread (ThreadB) doing updation & both threads are doing communication.

case-2 :- If we put [Thread.sleep(10000)] at Line (1), O/P will always be 5050, bcz

If main thread get chance first it will sleep for 10000 ms & in that time child thread will execute whole &

If child got first chance then obviously it will execute whole & O/P will be 5050.

Though we are fulfilling our requirement with this case, the disadvantage of this approach is that, In whatever time child thread get executed (for example in 10 ms) still main thread sleep for 10000 ms & we will get o/p after that. which is not good practice, performance wise.

case-3 :- If we put `b.join()` at line ① main thread will wait until completing child thread & we will always get o/p 5050.

But the problem with this case is, If there are 1 crore lines of code after that for loop completion, our main thread have to wait for all of that completion but here it requires only updation of b, which is done by 1 for loop only, so here also performance problems will occur, despite getting results.

case-4 :- so the best suitable methods for this case will be wait() & notify()

At line ① we will put b.wait() method & Immediately after for-loop we will put notify() method

so after completing for-loop immediately child thread will notify the main thread that update is done, now you can do anything with that variable.

↳ But wait() & notify() only works inside synchronized Area so, we have to call them inside otherwise we will get RE

`IllegalMonitorStateException.`

↳ Example of this case with flow of program is as shown below:-

Ex.

```
class ThreadDemoForTest
{
    p s v m(String[] args)
    {
        ThreadB b = new ThreadB();
        b.start();                          ← line-①
        synchronized(b)
        {
            ① S.O.P("Main Thread trying to call wait() method");
            ② b.wait();                     ← line-②
            ⑦ S.O.P("Main Thread got notification");
            ⑧ S.O.P(b.total);
        }
    }
}
```

```
class ThreadB extends Thread
{
    int total = 0;
    public void run()
    {
        synchronized(this)
        {
            ③ S.O.P("Child Thread starts calculation");
            ④ for(int i=1; i<=100; i++)
              {
                  total = total + i;
              }
            ⑤ S.O.P("Child Thread trying to give notification");
            ⑥ this.notify();
        }
    }
}
```

O/P :- Main Thread trying to call wait() method
Child Thread starts calculation
Child Thread trying to give notification
Main Thread got notification
5050

} case when Main Thread get chance first

(case-1)

Case-2 :- This case will happens if child Thread got chance first.

So, If we Add `Thread.sleep(10000)` at Line-① then after getting first chance also main thread will go to sleep for 10000 ms & child thread will get chance & execute whole & give notification also but as main thread is sleeping it will not take that notification

After 10000ms main thread waker up & get chance to execute by TS & execute Line-② & enter into waiting state & wait for threadB to give notification but as child thread already get executed it will never respond & main thread will be in waiting state forever.

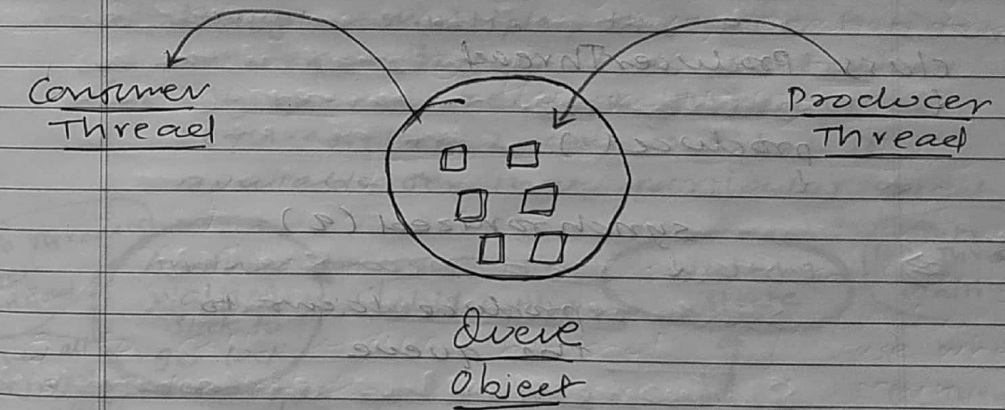
~ So, as a solution of this case main thread will decide to call wait method with "some time" as argument.

So, If we replace wait() with wait(10000) at Line-② after waiting 10000ms main will continue its execution irrespective of it get notified or not.

⇒ Producer - Consumer Problem

~ Producer Thread is responsible to produce items to the Queue and Consumer Thread is responsible to consume items from the Queue.

~ If Queue is Empty then consumer will call wait() method & enter into waiting state, After producing items to the Queue, producer thread is responsible to call notify() method then waiting consumer will get that notification & continue its execution with updated items.



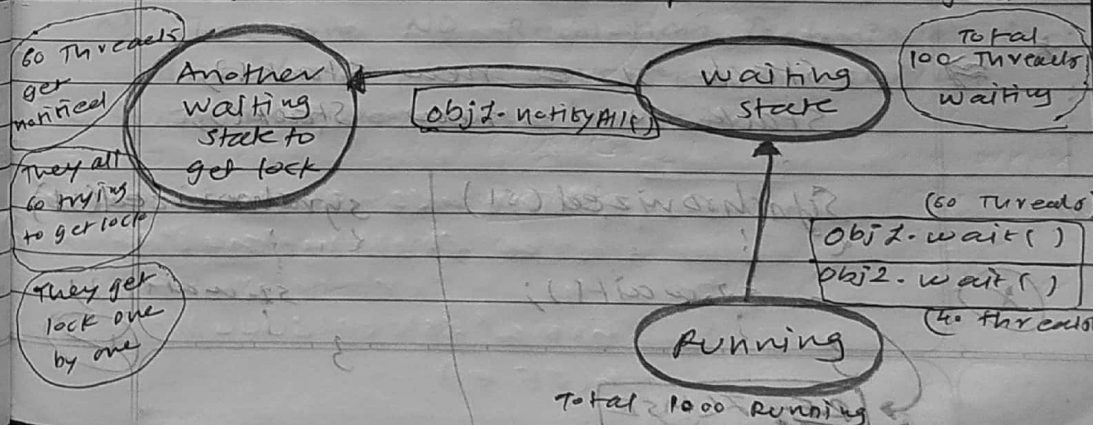
```
class ConsumerThread
{
    consume()
    {
        synchronized(q)
        {
            if(q is Empty)
            {
                q.wait();
            }
            else
            {
                consume items;
            }
        }
    }
}
```

```
class ProducerThread
{
    produce()
    {
        synchronized(q)
        {
            produce item and to
            the queue
            q.notify();
        }
    }
}
```

⇒ Difference b/w notify() & notifyAll()

⇒ we can use notify() method to give the notification for only one waiting thread, If multiple threads are waiting, then only one thread will be notified & the remaining threads have to wait for further notifications, which thread will be notified we can't expect, It depends on JVM.

⇒ we can use notifyAll() method to give the notification for all waiting thread of a particular object, Even though multiple thread is notified but execution will be performed one by one because threads requires lock & only one lock is available for that particular object



⇝ Real-time example :- Bus stand situation.

Total 200 People on Bus stand,
50 people for Destination - Delhi
70 people for Destination - Mumbai
+ others

Bus Arrived for Mumbai,

- Information is Announced that Mumbai Bus came (notification)
- All passengers waiting for Mumbai bus (70) get notified & run to pickup a bus. (Another waiting state)
- Bus has only one door from which only one person can enter, so they all have to be on the bus one by one. (so thread get lock one by one)

★ ⇝

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

synchronized(s1)	synchronized(s1)
{	{
;	;
s2.wait(); (X)	s1.notify(); (Y)
;	;
}	}
RELEASE	

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

For ex. If we are calling wait() method on s1, then we have to ~~[illegible]~~ get lock of s1 object but not s2 object.

⊛ Deadlock

⇝ If two threads are waiting for each other forever, such type 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 deadlock, but several Prevention Techniques are Available.

Ex. class A
{
public synchronized void d1(B b)
{
s.o.p("Thread 1 starts execution of d1() method");
try
{
Thread.sleep(6000);
}
catch(InterruptedException e)
{}
s.o.p("Thread 1 trying to call B's last()");
b.last();
}
public synchronized void last()
{
s.o.p("Inside A, this is last() method");
}
}

class B
{
public synchronized void d2(A a)
{
s.o.p("Thread 2 starts execution of d2() method");
try
{ Thread.sleep(6000); }
catch(IE e)
{}
s.o.p("Thread 2 trying to call A's last()");
a.last();
}

public synchronized void last()
{
s.o.p("Inside B, this is last() method");
}
}

class DeadLockDemo extends Thread
{
A a = new A();
B b = new B();

public void m()
{
this.start();
a.d1(b); // This line executed by main Thread
}
public void run()
{
b.d2(a); // This line executed by child Thread.
}
p s v m (sm)
{
DeadLockDemo d = new DeadLockDemo();
d.m();
}
}

O/P :-

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

[Now Nothing displays]

~ In the above problem if we remove atleast one synchronized keyword then the program won't enter into deadlock, hence synchronized keyword is the only reason for deadlock situation, so due to this while synchronized keyword we have to take special care.

Deadlock vs starvation

⇒ Long waiting of a thread where waiting never ends is called Deadlock.

Where as

~ Long waiting of a Thread where waiting ends at certain point is called starvation.

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

⇒ In the Deadlock example, If any of the thread don't sleep means if we don't write Thread.sleep(5000), then Deadlock will not happen because thread will not take much time to execute d1() method, so after executing d1() method one thread releases the lock which can be easily use by others to execute.

Daemon Threads

⇒ The threads which are executing in the background are called daemon threads.

Ex. ① Garbage Collector
② Signal Dispatcher
③ Attach Listener
⋮
etc.

→ The main objective of daemon threads is to provide support for Non-daemon thread (main thread)

for example, if main thread runs with the low memory, then JVM runs GC to destroy used objects, so that No. of bytes of free memory will be improved, with this free memory main thread can continue its execution.

→ Usually, daemon threads having low priority but based on our requirement daemon threads can run with high priority also.

→ we can check daemon nature of the thread by using isDaemon() method of Thread class.

```
public boolean isDaemon()
```

→ we can change Daemon nature of a thread by using setDaemon() method

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

→ But changing Daemon nature is possible before starting of a thread only, After starting a thread if we try to change Daemon nature then we will get R.E.

IllegalThreadStateException.

→ Default nature :-

→ By default, main thread is always Non-Daemon and for all remaining threads Daemon nature will be inherited from Parent to child, i.e if the parent thread is Daemon then automatically child thread is also daemon & if the parent thread is non-daemon then automatically child thread is also Non-Daemon

Note : It is impossible to change Daemon nature of main thread, because it is already started by JVM at begining.

class myThread extends Thread { }

Ex. class Test
{
p s v m(sta)
{
S.O.p(Thread.currentThread().
isDaemon); // false

Thread.currentThread().
setDaemon(true);
// R.E. ILTSE

MyThread t = new MyThread();

S.O.p(t.isDaemon()); // false

t.setDaemon(true);

S.O.p(t.isDaemon()); // true

}
}

→ Whenever last Non-Daemon thread terminates automatically all Daemon threads will be terminated, irrespective of their position.

Ex. class MyTh extends Thread
{
public void run()
{
for(int i=0; i<10; i++)
{
S.O.P("child Thread");

try
{
Thread.sleep(2000);
}
catch(IE e) { }
}
}
}

class DaemonThreadDemo
{
p s v m(sta)
{
MyTh t = new MyTh();
t.setDaemon(true); → line ①

t.start();

S.O.P("End of main
Thread");
}
}

⇒ If we are commenting line-① both main & child threads are Non-Daemon and hence both threads will be executed until their completion if we are not commenting line-① then main thread is Non-Daemon and child thread is Daemon hence whenever main thread terminates automatically child thread will be terminated In this case o/p :-

End of main Thread
Child Thread

(or)

End of main Thread

(or)

Child Thread
End of main Thread

✱ Interview FAQ's

① ⇒ Java Multi-threading concept is implemented by using the following two models.

(1) Green Thread model
(2) Native OS model

Green Thread model

⇒ The thread which is managed completely by JVM without taking underlying OS support is called Green Thread.

⇒ Very few OS like SUN Solaris provide support for Green Thread model, Anyways Green Thread model is deprecated & NOT recommended to use.

Native OS Model

⇒ The thread which is managed by the JVM with the help of underlying OS, is called Native OS model.

⇒ All windows based OS provide support for Native OS model.

(2) How to stop a Thread ?

→ We can stop a thread execution by using stop() method of Thread class.

```
public void stop()
```
(deprecated)

→ If we call stop() method then immediately the thread will enter into dead state, Anyways stop() method is deprecated & not recommanded to use.

(3) How to suspend & Resume a Thread?

→ We can suspend a thread by using suspend() method of Thread class, then immediately thread will be enter into suspended state

→ we can resume a suspended thread by using resume() method of Thread class, then suspended thread can continue its execution.

→
```
public void suspend()
```
→
```
public void resume()
```
(Deprecated)

→ Anyways this methods are deprecated & not recommanded to use.

→ Complete life cycle of Thread execution as shown in the figure in the next page

* Thread * Life cycle