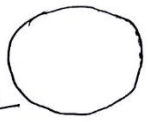
SYNCHRONIZATION

Synchronized is a modifier applicable only for methods & blocks but not for classes & variables

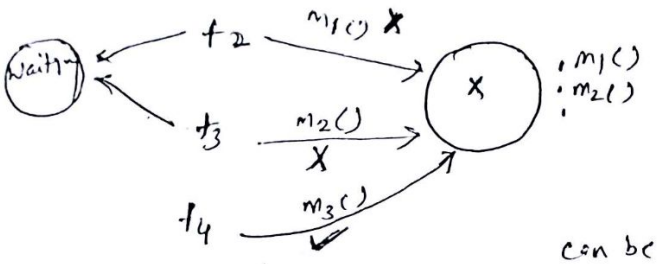
- If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem.
- To overcome this problem we should use synchronized keyword.
- If a method / block declared as synchronized then at a time only one thread is allowed to execute that method/block on the given object so that data inconsistency problem will be resolved.
- The main advantage of synchronized keyword is we can resolve data inconsistency problems but the main disadvantage of synchronized keyword is, it increases waiting time of threads & creates performance problems. Hence if there is no specific requirement then it is not recommended to use synchronized keyword.

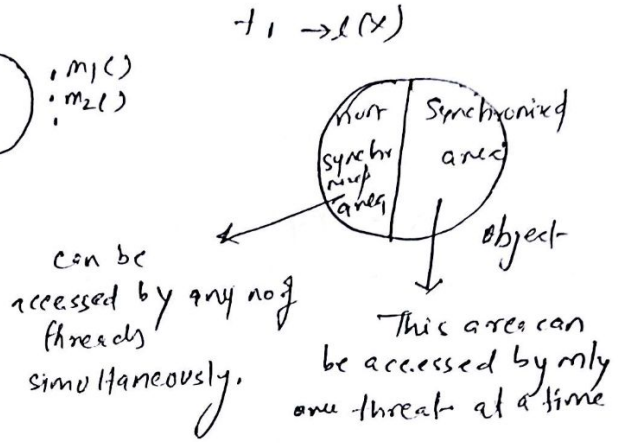


- Internally synchronization concept is implemented by using lock. Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.
- If a thread wants to execute synchronized method on the given object first it has to get lock of that object. Once thread gets the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases lock.
- Acquiring & releasing lock internally takes care by JVM. Programmer is not responsible for this activity.

class X
{
synchronized m1() {}
synchronized m2() {}
m3() {}
}

→ While a thread executing synchronized method on the 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.



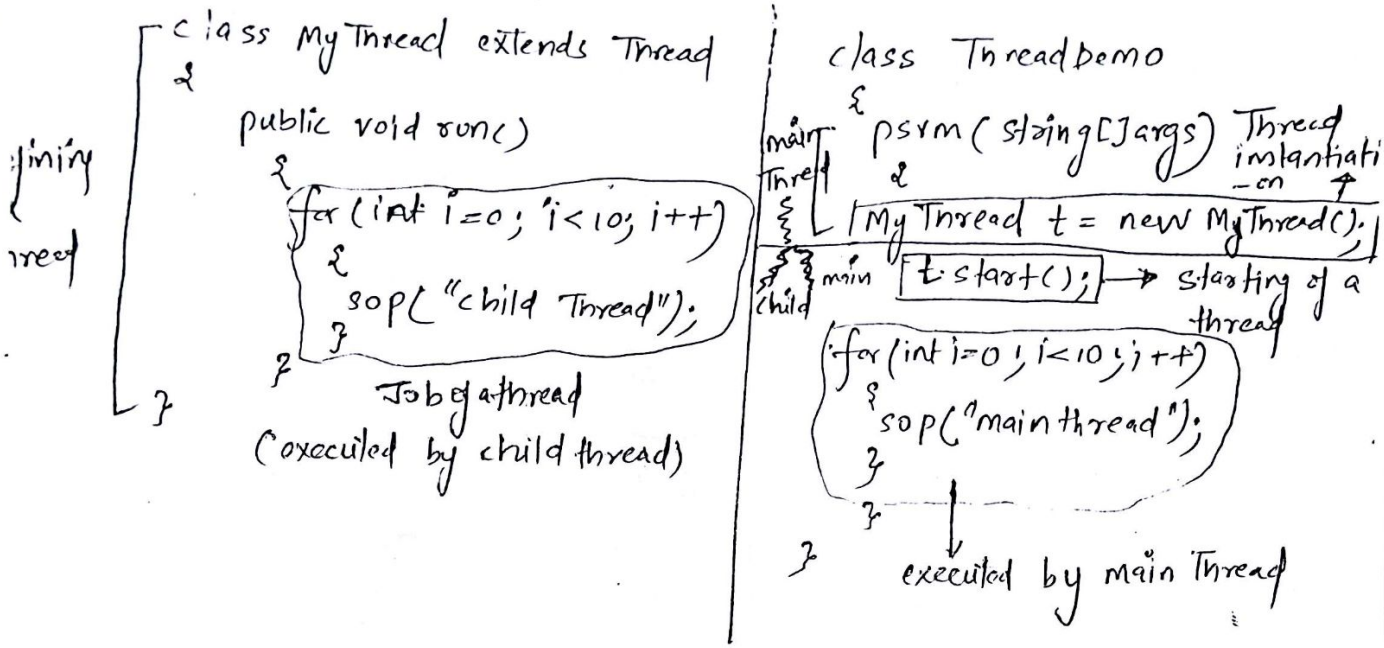


class X
{
synchronization {
where ever we are performing update operation [add/delete/replace/remove] where state of object changing
}
non-synchronized
{
Whenever object state won't be changed [read] operation]
}

ex class ReservationSystem
{
checkAvailability()
{
// read operation
}
synchronized bookTicket()
{ update operation
}
}

1) By extending Thread class

t.start() → calls start() method of Thread class, which internally calls run() method of MyThread class.



output → mixed output (no dependency)
<any order threads can follow>

Case1 Thread schedular

1) It is the part of JVM
2) It is responsible to schedule threads i.e if multiple threads are waiting to get chance of execution then in which order threads will be executed is decided by Thread schedular.

We can't expect exact algorithm followed by thread schedular, it is varied from JVM to JVM. Hence we can't expect threads execution order & exact output.

Hence whenever situation comes to multithreading, there is no guarantee for exact output but we can provide several possible outputs

possible outputs:

1) main Thread
main Thread
:
10 times
child Thread
child Thread
:
10 times

2) child Thread
child Thread
:
10 times
main Thread
main Thread
:
10 times

3) main Thread
child Thread
main Thread
:
:

4) child Thr
main Thre
child Thread
:
:

***

case 2 Difference between t.start() & t.run();

t.start() → Thread class method

t.run() → ↓

```
class MyThread extends Thread
{
public void run()
{
sop (for(int i=0;i<10;i++)
{
sopC"child Thread");
}
}
}
```

```
class ThreadDemo
{
psvm (String[] args)
{
MyThread t = new MyThread();
t.run();
for (int i=0; i<10; i++)
{ sop("main Thread");
}
}
}
```

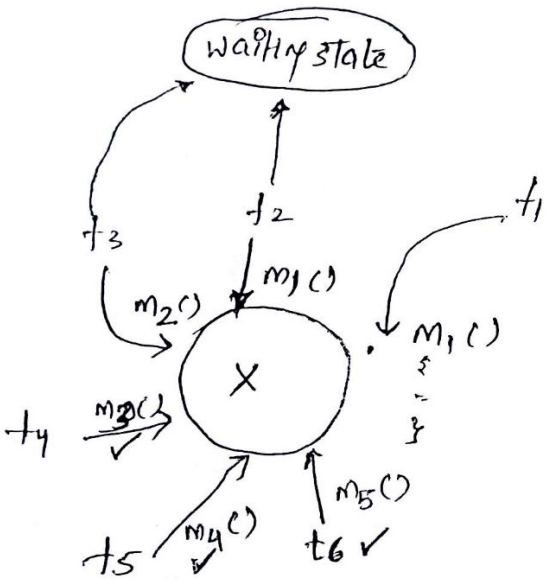
Child Thread
child Thread
:10 times
main Thred
:10 times

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

But, in the case of t.run(), a new thread wont be created & run() method will be executed just like a normal method call. by main thread. Hence in the above program, output is fixed child Thread (10 times) followed by main Thread (10 times) This total output produced by only main Thread

Once method execution completes, automatically thread releases the lock.

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



$t_1 \rightarrow CL(X)$
$t_5 \rightarrow OL(X)$

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 :-

a) normal static method
b) instance synchronized method
c) normal instance method

```
class Display
{
public void displayn()
  {
   for(int i=1; i<=10; i++)
     {
      sop(i);
     try{
        Thread.sleep(2000);
        }
     catch(InterruptedException e)
       {
       }
     }
  }
public void displayc()
  {
   for(int i=65; i<=75; i++)
     {
      sop((char)i);
     try{ Thread.sleep(2000);
        }
     catch(IE e)
       {
       }
     }
  }
}
```

```
class Test
{
 psvm(String[] args)
  {
Display d = new Display();
MyThread1 t1 = new MyThread1(d);
MyThread2 t2 = new MyThread2(d);
     t1.start();
     t2.start();
  }
}
```

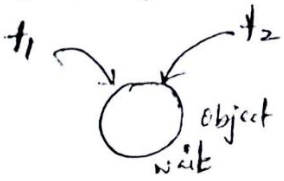

```
class MyThread1 extends Thread
 {
  Display d;
  MyThread1(Display d)
    {
     this.d=d;
    }
 public void run()
   {
    d.displayn();
   }
}
```

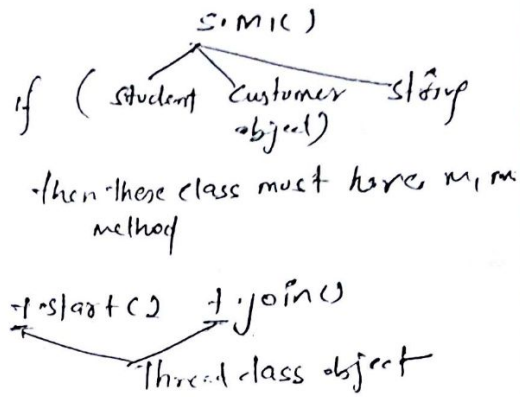
```
class MyThread2 extends Thread
 {
  Display d;
 MyThread2(Display d)
   {
    this.d=d;
   }
 public void run()
   {
    d.displayc();
   }
}
```

Inter Thread Communication

- Two threads can communicate with each other by using wait, notify & notifyAll methods.
- 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 to perform updation, after performing updation, it is responsible to call notify method then waiting thread will get that notification & continue its execution with those updated items

- wait, notify & notifyAll methods present in Object class but not in Thread class because Thread can call these methods on any Java object



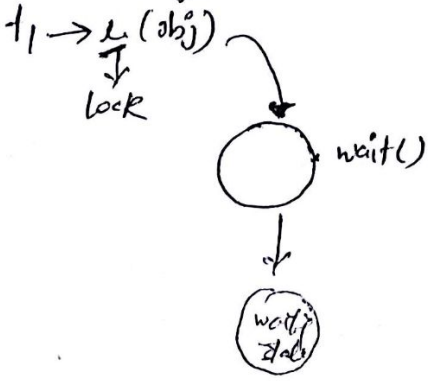


start & join applicable only for Threads / Thread object

To call wait, notify & notifyAll methods on any object, thread should be owner of that object (must acquire lock on that object) ie thread should be inside synchronized area.

Hence we can call wait, notify & notifyAll methods only from synchronized area, otherwise we will get RuntimeException "IllegalMonitorStateException"

- If a thread calls wait method on any object, it immediately releases the lock of that particular 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 (instant)

* Except wait, notify & notifyAll methods, there is no other method where thread releases the lock.

1) wait() method →

public final void wait() throws InterruptedException

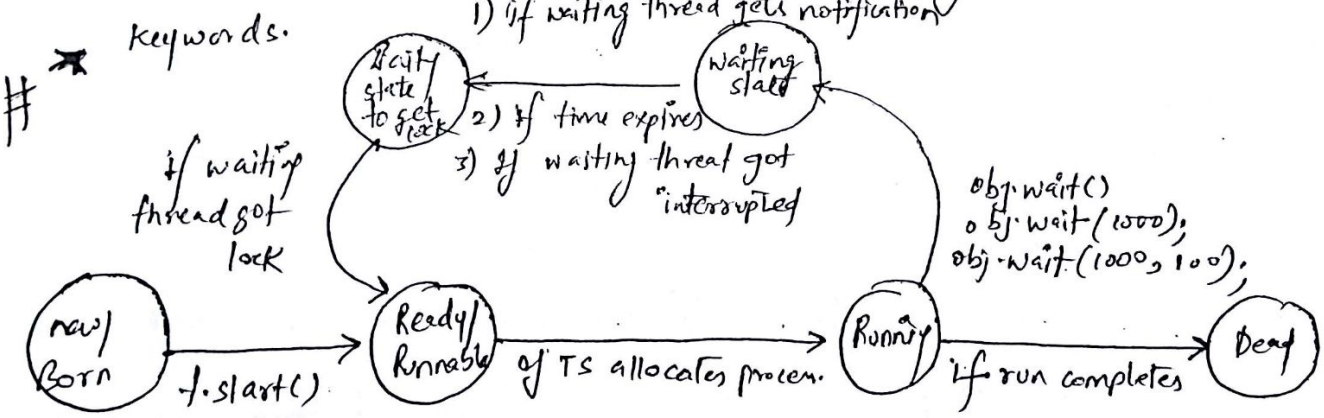
public final native void wait(long ms) "

public final void wait(long ms, int ns) "

2) 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, compulsary we should handle the exception either by using try-catch or throws keywords.



CHANGES IN TRASITION STATE DIAGRAM

= notification
– get the lock

```
class ThreadA
{
  psvm(String[]args)
  {
  ThreadB t = new ThreadB();
  t.start();
  sop("hi");
  sop(t.total);
  }
}
```

```
class ThreadB extends Thread
{
  int total=0;
  public void run()
  {
  for(int i=0; i<1000; i++)
  {
  total = total+i;
  }
  }
}
```

500500

* Using wait() & notify()

```
class ThreadA
{
  psvm(String[]args) throws InterruptedException
  {
  ThreadB b = new ThreadB();
  b.start();
  synchronized(b){ t.wait(); }
  sop(t.total);
  }
}
```

```
class ThreadB
{
  int total=0;
  public void run()
  {
  synchronized(this)
  {
  for(int i=0; i<=1000; i++)
  {
  total = total+i;
  this.notify();
  }
  }
}
```

3) Program (file)

Producer-Consumer Problem

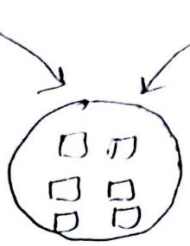
→ Producer thread is responsible to produce items to the Queue & consumer thread is responsible to consume items from the Queue

→ If Queue is empty then consumer thread 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 ProducerT
{
produce()
{
 synchronized(q)
 {
  produce items to the
   Queue
  q.notify();
  }
 }
}
```

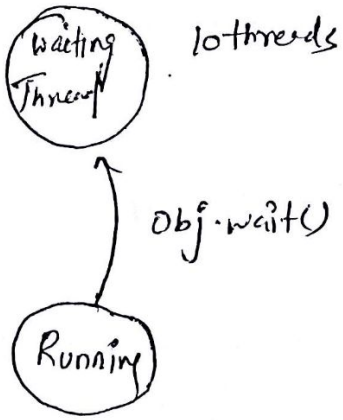



```
class ConsumerT
{
 consume()
 {
   synchronized(q)
   {
    if( q is empty)
      q.wait();
    else
     consume items
   }
 }
}
```

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 notify & the remaining threads have to wait for further notifications.

→ Which thread will be notify we can't expect, it depends on JVM



We can use notifyAll to give the notification for all waiting threads of a particular object. Then only one thread

Even though multiple threads notified, execution will be performed one by one because threads require lock & only one lock is available.

DEADLOCK

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

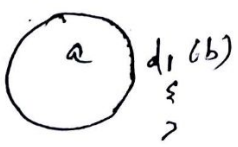
synchronized keyword is the only reason for deadlock situation Hence, while using synchronized keyword we have to take special care

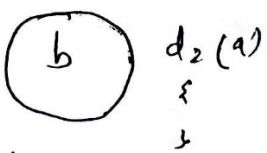
There are no resolution techniques for deadlock but several prevention techniques are available

ex:

```
class A
{
synchronized void d1(B b)
        {
        :
        b.last();
        }
synchronized last()
        {
        :
        }
}
```

```
classB
{
synchronized void d2(A a)
        {
        :
        a.last();
        }
synchronized last()
        {
        :
        }
}
```





t1 will enter into waiting state to gets b's lock & currently t1 holds a's lock

t2 will enter into waiting state to gets a's lock & t2 currently holds b's lock.

```
class A
{
  public synchronized void d1(B b)
  {
    sop("Thread1 starts execution of d1() method");
    try { Thread.sleep(5000); }
    catch(IE e) { }
    sop("Thread-1 trying to call b's last method");
    b.last();   // m2
  }
  public synchronized void last()   // m1
  {
    sop("Inside A, last() method");
  }
}
class B
{
  public synchronized void d2(A a)
  {
    sop("Thread-2 starts execution of d2() method");
    try { Thread.sleep(5000); }
    catch(IE e) { }
    sop("Thread-2 trying to call A's last() method");
    a.last();   // m1
  }
  public synchronized void last()   // m2
  {
    sop("Inside B, last() method");
  }
}
```

```
class MyThread extends Thread
{
    A a = new A();
    B b = new B();
    public void m1()
    {
        this.start();   // main starts child thread
        a.d1(b);        ----> main thread
    }
    public void run()
    {
        b.d2(a);        ----> child thread
    }
    psvm(String[] args)
    {
        MyThread t = new MyThread();
        t.m1();
    }
}
```

In the above program 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. Due to this, while using synchronized keyword we have to take special care.

a.d1(b) { } run() { } ?

Starvation

Deadlock Vs

long waiting of a thread where waiting never ends is called deadlock

whereas, long waiting of a thread where waiting ends at certain point is called starvation.

Ex low priority thread has to wait until completing all high priority threads. This waiting can be long waiting but ends at certain point, is called starvation.

1 thread - P(1) ← wait
1000 threads P(10)