7- Multithreading

1. Introduction
2. The ways to define, instantiate & start a new Thread.
3. Getting & Setting name of Thread.
4. Thread priorities
5. The methods to prevent Thread Execution
   i) yield()
   ii) sleep()
   iii) join()
6. Synchronization
7. Interthread Communication
8. Deadlock
9. Daemon Threads
10. Conclusions

1) Introduction :—

Multi tasking :—

→ Executing several tasks simultaneously is the concept of Multitasking.

→ There are 2 types of multitasking.

1. Process based Multitasking
2. Thread based Multitasking.

1) Process based Multitasking:—

→ Executing several tasks simultaneously where each task is a separate independent process, such type of multitasking is called process based multitasking.

Ex: While typing a Java program in the editor we can able to listen

MP3 audio songs from the same system. At the same time we can download a file from the internet.

All these tasks are independent of each other and will execute simultaneously. and hence it is Process based Multi-tasking.

→ Process based multitasking is best suitable at OS level.

2) Thread based Multitasking :—

→ Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based Multitasking and each independent part is called a Thread.

→ Thread based Multitasking is best suitable at Programmatic level.

→ Whether it process based or thread based, the main purpose of multitasking is to reduce response time & to improve performance of the system.

→ The main important application areas of Multithreading are

1) To develop multi media graphics.
2) To develop video games.
3) To develop Animations.
4) To develop web & application servers

→ When compared with old languages Java provides in-built support for multithreading (by providing a rich API: Thread, ThreadGroup, ThreadLocal, Runnable etc).

→ Hence developing multithreading examples in Java very easy when compared with old languages.

2) Defining, Instantiating and Starting a new Thread:-

→ We can define thread in the following 2 ways.

1) By extending Thread class
2) By implementing Runnable interface.

1) Defining a Thread by extending Thread class:-

Ex:

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

Defining a Thread

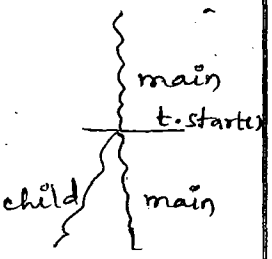
→ Job of Thread

```
class ThreadDemo
{
  P s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
    for(int i=0; i<10; i++)
    {
      S.o.p("main thread");
    }
  }
}
```

→ Instantiation of Thread

→ Starting of a Thread

→ main thread



*** Note:- start() method is not a normal method call, it will start a new flow of execution (i.e., new thread).

Case(i): Thread Schedular :—

→ If multiple threads are waiting then in which order threads will be executed is decided by Thread Schedular.

→ Thread Schedular is the part of JVM and we can't expect exact behaviour of Thread Schedular. It is JVM vendor dependent.

→ Due to this we can't expect the order in which threads will be executed & hence we can't expect exact o/p, but several possible o/p's we can define.

→ The following are various possible o/p's for the above program.

possibility-1	possibility-2	possibility-3	possibility-4
main thread	child thread	main thread	child thread
main thread	child thread	child thread	main thread
⋮	⋮	main thread	child thread
(10 times)	(10 times)	child thread	main thread
child thread	main thread	⋮	⋮
child thread	main thread	(10 times)	(10 times)
⋮	⋮		
(10 times)	(10 times)		

Case(ii): Difference blw t.start() and t.run() :—

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

But in case of t.run(), no new Thread will be created & run() method will be executed just like a normal method call by main thread.

→ In the above program, if we replace t.start() with t.run() then the o/p is

child thread
child thread
⋮ (10 times)
main thread
main thread
⋮ (10 times)

} Total o/p produced by only main thread.

Case(iii): Important of Thread class start() method :-

→ Thread class start() method is responsible to perform all required activities for thread like Registering the thread with Thread Schedular etc. After completing all required activities it will invoke run() method.

```
start()
{
  1. Register this thread with Thread Schedular
  2. Perform all other required activities
  3. Invoke run() method.
}
```

→ Hence without executing Thread class start() method there is no chance of starting a new Thread in Java.

→ Hence Thread class start() is considered as heart of Multithreading.

***

Case(iv): Overloading of run() method :-

→ We can overload run() method, but Thread class start() method will always call no-argument run() method only.

→ The other overloaded method we have to call explicitly then it will be executed just like a normal method call.

Ex:

```
class MyThread extends Thread
{
  public void run()
  {
    S.o.p ("no-arg run");
  }
  public void run(int i)
  {
    S.o.p ("int-arg run");
  }
}
```

(Overloaded methods)

```
class ThreadDemo
{
  p s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
  }
}
```

o/p : no-arg run

Case(v): If we are not overriding run() method :—

→ If we are not overriding run() method then Thread class run() method will be executed which has empty implementation. Hence we won't get any o/p.

Ex:
```
class MyThread extends Thread
{
}
```

```
class ThreadDemo
{
  p s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
  }
}
```

o/p : no o/p.

Note :- It is highly recommended to override run() method otherwise don't go for Multithreading concept.

Case(vi): If we override start() method :—

→ If we are overriding start() method then it will be executed just like a normal method call by main thread and 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 ThreadDemo
{
  p s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
    S.o.p("main method");
  }
}
```

main
main
start method
main method

o/p : start method
main method
} Total o/p produced by only main thread.

Note :- It is never recommended to override start() method in our class.

Case(vii):

Ex:
```
class MyThread extends Thread
{
    public void start()
    {
        super.start();
        S.o.p("start method");
    }
    public void run()
    {
        S.o.p("run method");
    }
}
```

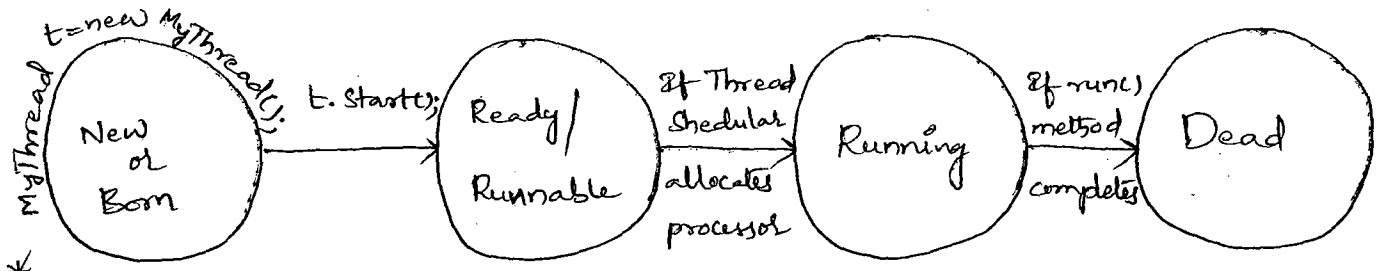
```
class ThreadDemo
{
    p s v m(-)
    {
        MyThread t=new MyThread();
        t.start();
        S.o.p("main method");
    }
}
```

main → super.start(); → S.o.p("start method");
main → t.start(); → S.o.p("main method");
child → run method
s.m.m → S.o.p("run method");

o/p: possibility-1	possibility-2	possibility-3
start method	run method	start method
main method	start method	run method
run method	main method	main method.

***

Case(viii): Life Cycle of Thread:-



***

Case(ix):

→ After starting a thread if we are trying to restart same thread once again we will get runtime Exception saying, IllegalThreadStateException.

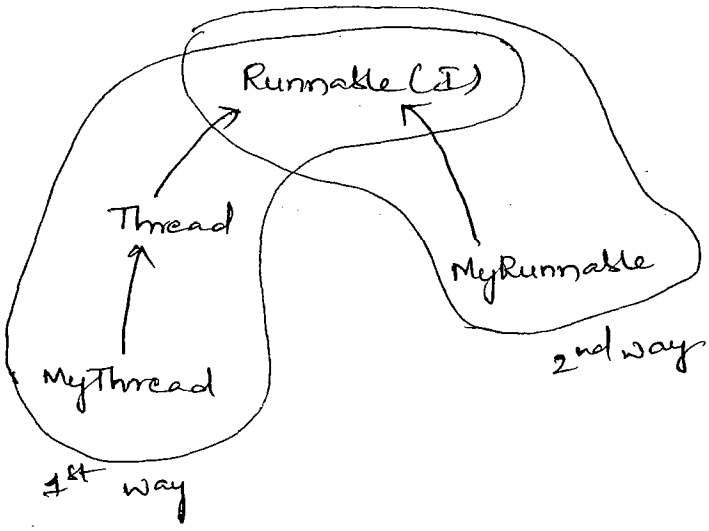
Ex:
```
Thread t=new Thread();
t.start();
:
t.start(); → RE: IllegalThreadStateException
```

2) Defining a Thread by implementing Runnable interface :—

→ We can define a thread even by implementing Runnable interface directly.

→ Runnable interface present in java.lang package & it contains only one method.

```
public void run()
```



Ex:

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

Defining Thread

→ JOB of Thread

```
class ThreadDemo
{
    p s v m(-)
    {
        MyRunnable r=new MyRunnable();
        Thread t=new Thread(r);
        t.start();
        for(int i=0; i<10; i++)
        {
            S.o.p("main Thread");
        }
    }
}
```

→ We can't expect exact o/p for the above program, but several possible outputs we can provide.

Case Study:-

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

Case(i): t1.start();

→ A new Thread will be created which is responsible for the execution of Thread class run() method, which has empty implementation.

Case(ii): t1.run();

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

Case(iii): t2.start();

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

Case(iv): t2.run();

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

Case(v): r.start();

→ We will get compile time error saying, MyRunnable class doesn't contain start ability.

```
CE: cannot find symbol
symbol: method start()
location: class MyRunnable
```

Case(vi): r.run();

→ MyRunnable run() method will be executed just like a normal method call & new Thread won't be created.

Q1: In which of above cases a new Thread will be created?

Ans:- t1.start(); t2.start();

Q2: In which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method?

Ans:- t2.start();

Q3: In which of the above cases MyRunnable run() method will be executed?

Ans:- t2.start(); t2.run(); r.run();

Recommended way to define Thread :—

→ Among 2 ways of defining a Thread, implements Runnable approach is recommended to use.

→ In 1st approach, our [illegible] class always extending Thread class & hence there is no chance of extending any other class. So that we will miss Inheritance benefit.

→ But in 2nd approach, while implementing Runnable interface we can extend any other class & hence we won't miss any Inheritance benefit.

→ Becoz of above reason implements Runnable approach is recommended to define a Thread.

Thread class constructors :—

① Thread t=new Thread();

② Thread t=new Thread(Runnable r);

③ Thread t=new Thread(String name);

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

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

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

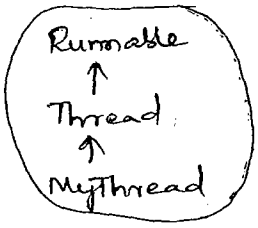
⑦ Thread t=new Thread(ThreadGroup g, Runnable r, String name);

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

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

Ex:
```
class MyThread extends Thread
{
  public void run()
  {
    S.o.p("Child Thread");
  }
}
```

```
class ThreadDemo
{
  P S V m(-)
  {
    MyThread t=new MyThread();
    Thread t1=new Thread(t);
    t1.start();
  }
}
```



3) Getting & Setting name of a Thread :—

→ Every Thread in Java has some name it may be explicitly provided by programmer or default name generated by JVM.

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

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

Ex:
```
class MyThread extends Thread
{
}
class Test
{
  P S V m(-)
  {
    S.o.p(Thread.currentThread().getName()); ⇒ o/p : main
    MyThread t=new MyThread();
    S.o.p(t.getName()); ⇒ o/p : Thread-0
    Thread.currentThread().setName("Pawan Kalyan");
```

```
        S.o.p (Thread.currentThread().getName()); ⇒ o/p : Pawan Kalyan
        S.o.p (10/0); → RE : Exception in thread "Pawan Kalyan" : j.l.AE :
    }                                                        / by zero
}
```

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

4) Thread Priorities :—

→ Every thread in Java has some priority it may be explicitly provided by programmer or default priority generated by JVM.

→ The valid range of Thread priorities is 1 to 10 (but not 0 to 10), where 1 is least and 10 highest.

→ Thread class defines the following constants to represent some standard priorities.

Thread.MIN_PRIORITY ⟶ 1

Thread.MAX_PRIORITY ⟶ 10

Thread.NORM_PRIORITY ⟶ 5.

Q: Which of the following are valid priorities in Java?

✗ ① Thread.HIGH_PRIORITY

✗ ② Thread.LOW_PRIORITY

✗ ③ 0

✓ ④ 1

✓ ⑤ Thread.MIN_PRIORITY

✓ ⑥ Thread.MAX_PRIORITY

→ Thread Schedular will use Thread priorities while allocating processor.

→ The thread which is having highest priority will get chance first.

→ If two threads having the same priority then we can't expect in which order threads will be executed & it depends on Thread Schedular.

→ Thread class defines the following methods to get & set priority of a thread.

```
public final int getPriority();
public final void setPriority(int priority);
```

→ the allowed values are 1 to 10 o.w. we will get RE: IllegalArgumentException

Ex: Thread t=new Thread();
t.setPriority(10); ✓
t.setPriority(0); } → RE: IllegalArgumentException
*** t.setPriority(100); }

Default Priority :—

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

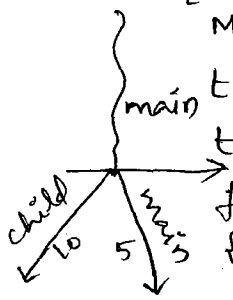
Ex:
```
class MyThread extends Thread
{
}
class Test
{
  p s v m(—)
  {
    S.o.p(Thread.currentThread().getPriority()); ⇒ o/p : 5
    Thread.currentThread().setPriority(9);
    MyThread t=new MyThread();
    S.o.p(t.getPriority()); ⇒ o/p : 9
  }
}
```

MyThread → parent class is Thread
MyThread → parent thread is main thread.

→ parent class & parent thread are different.

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

```
class ThreadPriorityDemo
{
  p s v m(-)
  {
    MyThread t=new MyThread();
    t.setPriority(10); ---> ①
    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)
```

→ If we are commenting line ① then both child & main threads have same priority i.e., 5 and hence we can't expect exact execution order & exact o/p.

Note:- Some OS's won't provide proper support for Thread priorities.

5) The methods to prevent Thread Execution :—

→ We can prevent Thread execution by using the following 3 methods.

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

1) yield():—

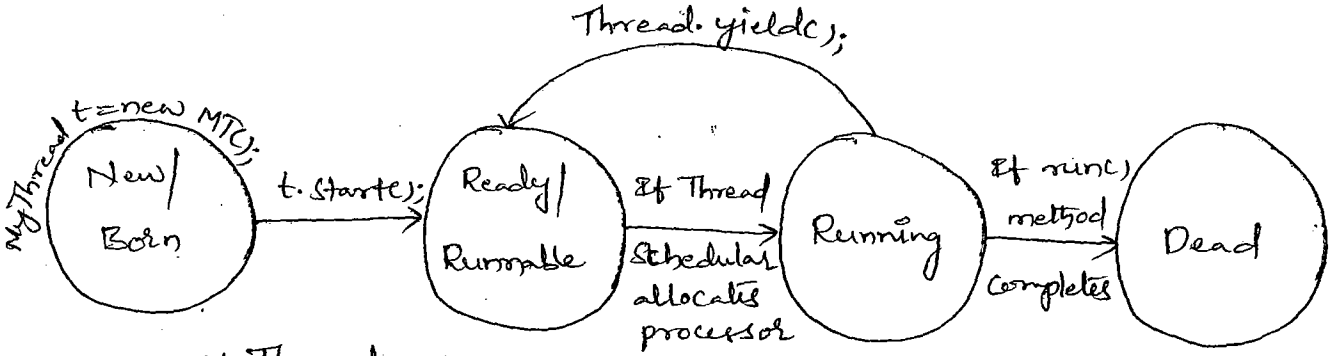
→ yield() method causes to pause current executing thread to give the chance to remaining waiting threads of same priority.

→ If there is no waiting thread or all waiting threads having low priority then the same thread will continue its execution.

→ If several waiting threads having the same priority then we can't expect which thread will get chance & it depends on Thread Schedular.

→ The thread which is yielded, when it will get the chance once again we can't expect, it depends on Thread Schedular.

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



Ex:
```
class MyThread extends Thread
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      Thread.yield();
      S.o.p("Child Thread");
    }
  }
}
```

```
class ThreadYieldDemo
{
  p s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
    for(int i=0; i<10; i++)
    {
      S.o.p("Main Thread");
    }
  }
}
```

→ In the above example, the chance of completing main thread first is high becoz child thread always calls yield() method.

Note:- Some os's won't provide proper support for yield() method.

2) join() :—

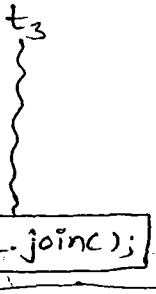
→ If a thread wants to wait until completing some other thread then we should go for join() method.

For example, If a thread t_1 wants to wait until completing thread t_2 then t_1 thread has to call t_2.join() then immediately t_1 thread will be entered into waiting state.

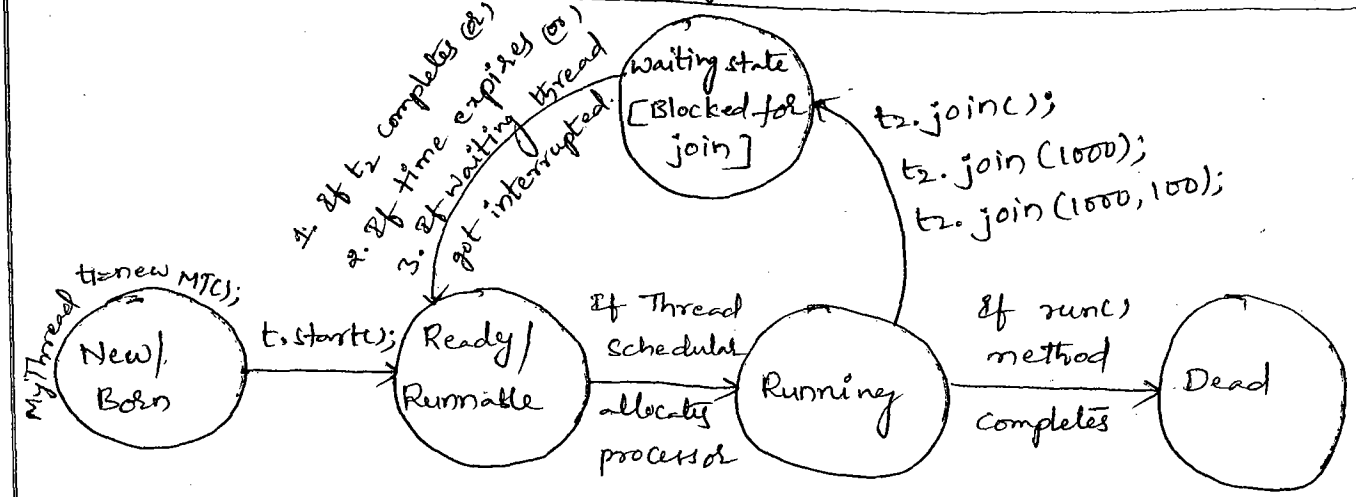
Ex: Venue Fixing

Wedding cards Printing

Wedding cards distribution



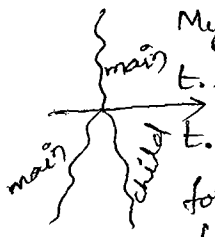

```
public final void join() throws InterruptedException
public final void join(long ms) throws InterruptedException
public final void join(long ms, int ns) throws InterruptedException
```



Case(i): Waiting of main Thread Until child Thread completion:-

Ex:
```
class MyThread extends Thread
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      S.o.p("Seetha Thread");
      try
      {
        Thread.sleep(2000);
      }
      catch(IE e)
      {
      }
    }
  }
```

```
class ThreadJoinDemo
{
  p s v m(-)throws IE
  {
    MyThread t=new MyThread();
    t.start();
    t.join(); ---> ①
    for(int i=0; i<10; i++)
    {
      S.o.p("Rama Thread");
    }
  }
}
```



→ If we are not commenting line ① then main thread executes join() method on child thread object.

→ Hence main thread has to wait until completing child thread object.

→ In this case, the o/p is

```
Seetha Thread
Seetha Thread
  :
(10 times)
Rama Thread
Rama Thread
  :
(10 times)
```

→ If we are commenting line ① then we can't expect exact execution order & exact o/p.

Note:- Every join() method throws IE which is checked Exception and hence whenever we are using join() method compulsory we should handle IE either by try-catch or throws keyword, o.w. we will get CE.

Case(ii): Waiting of Child Thread Until Completing Main Thread:-

Ex:
```
class MyThread extends Thread
{
  static Thread mt;
  public void run()
  {
    try
    {
      mt.join();
    }
    catch(IE e)
    {
    }
    for(int i=0; i<10; i++)
    {
      S.o.p("child Thread");
    }
  }
}
```

```
class ThreadJoinDemo1
{
  p s v m(-) throws IE
  {
    MyThread.mt=Thread.currentThread();
    MyThread t=new MyThread();
    t.start();
    for(int i=0; i<10; i++)
    {
      S.o.p("Main Thread");
      Thread.sleep(2000);
    }
  }
}
```

O/P:
```
Main Thread
Main Thread
  |
(10 times)
Child Thread
Child Thread
  |
(10 times)
```

Case(iii):

→ If main thread calls join() method on child thread object & child thread calls join() method on main thread then both threads will wait for each other. So the program will be hanged like Dead lock situation.

Case(iv):

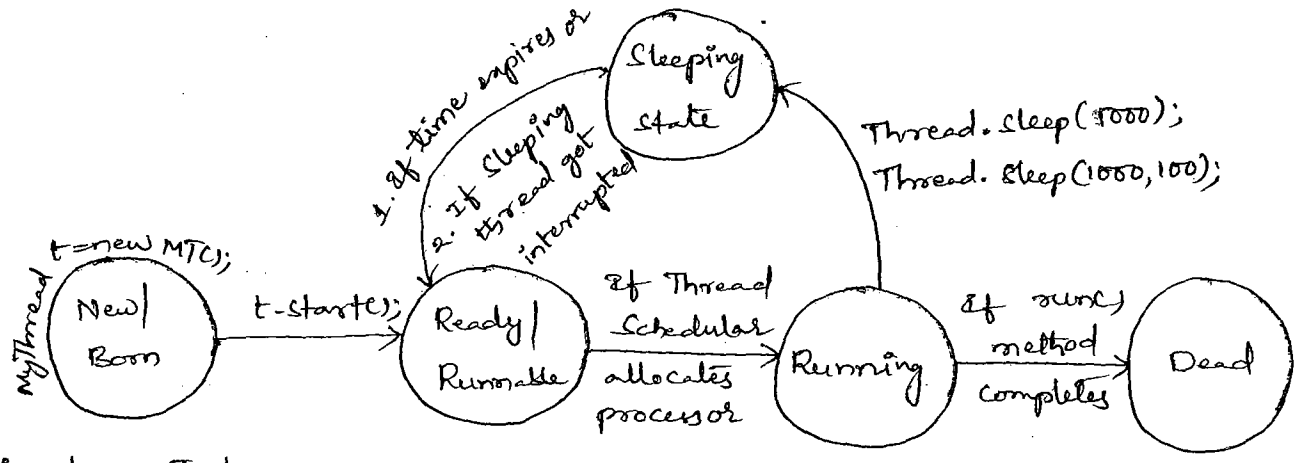
→ If a thread calls join() method on the same thread object then the program will be hanged like Deadlock.

Ex: Thread.currentThread().join();

3) sleep() :—

→ If a thread don't want to perform any operation for a particular amount of time i.e., just pausing is required then we should go for sleep() method.

```
public static native void sleep(long ms) throws IE
public static void sleep(long ms, int ns) throws IE
```



Ex: class Test

```
{
  p s v m() throws IE
  {
    for(int i=1; i<10; i++)
    {
      S.o.p("Slide-"+i);
      Thread.sleep(3000);
    }
  }
}
```

o/p : Slide - 1
Slide - 2
⋮
(9 times)

How a Thread can interrupt another Thread :—

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

```
public void interrupt();
```

Ex:
```
class MyThread extends Thread
{
  public void run()
  {
    try
    {
      for(int i=0; i<10; i++)
      {
        S.o.p("I am Lazy Thread");
        Thread.sleep(2000);
      }
    }
    catch(IE e)
    {
      S.o.p("I got Interrupted");
    }
  }
}
```

```
class ThreadInterruptDemo
{
  P s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
    t.interrupt();
    S.o.p("End of main Thread");
  }
}
```

main → t.start(); → t.interrupt(); → S.o.p("End of main Thread");

child → run()

→ In the above program, main thread interrupts child thread then the o/p is:

```
End of main Thread
I am Lazy Thread
I got interrupted
```

***

Note:— Whenever we are calling interrupt() method we may not see impact immediately. If the target thread is in sleeping or waiting state then immediately the thread will be interrupted.

If the target thread not in sleeping or waiting state then interrupt call will wait until target thread entered into sleeping or waiting state.

Once target thread entered into sleeping or waiting state then immediately it will be interrupted.

There is only one situation where interrupt call will be wasted i.e., if the target thread never entered into sleeping or waiting state in its life time.

Ex:
```
class MyThread extends Thread
{
  public void run()
  {
    for(int i=1; i<=10000; i++)
```

```
    {
      S.o.p("I m Lazy Thread -" +i);
    }
    S.o.p("I m entering into sleeping state");
    try
    {
      Thread.sleep(10000);
    }
    catch(IE e)
    {
      S.o.p("I got interrupted");
    }
  }
}
class ThreadSleepDemo1
{
  P s v m(-)
  {
    MyThread t=new MyThread();
    t.start();
    t.interrupt();
    S.o.p("End of Main Thread");
  }
}
```

→ In the above program, interrupt call waited until executing for loop 10000 times.

Comparison table of yield(), join() & sleep() method :—

Property	yield()	join()	sleep()
1. purpose	It causes to pause current executing thread to give the chance for remaining waiting threads of same priority.	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 a particular amount of time i.e., just pausing is required then we should go for sleep() method.
2. Is it static?	Yes	No	Yes
3. Is it final?	No	Yes	No
4. Is it overloaded?	No	Yes	Yes
5. Is it throws IE?	No	Yes	Yes
6. Is it native?	Yes	No	sleep(long ms) → native sleep(long ms, int ns) ↓ non-native

9. Daemon Threads :—

→ The threads which are executing in the background are called Daemon Threads.

Ex: Garbage Collector.

→ The main purpose of Daemon Threads is to provide support for non-daemon threads (main threads).

Ex: Whenever main thread running with less memory then JVM runs Garbage Collector to destroy useless objects. So that free memory will be provided.

→ We can check whether the thread is Daemon or not by using isDaemon() method of Thread class.

```
public final boolean isDaemon();
```

→ We can change daemon nature of a thread by using setDaemon() method.

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

→ We can change daemon nature before starting of a thread, by mistake if we are trying to change daemon nature after starting of a thread then we will get RE saying, IllegalThreadStateException.

Ex:
```
MyThread t = new MyThread();
t.setDaemon(true); ✓
t.start();
t.setDaemon(false); → RE : IllegalThreadStateException
```

*** Default Nature :-

→ By default main thread is always non-daemon and for all remaining threads daemon nature will be inheriting from parent to child.

*** → We can't change daemon nature of main thread because it is already started by JVM at very begining.

Ex:
```
class MyThread extends Thread
{
}
```

```
class Test
{
  p s v m(-)
  {
    S.o.p (Thread.currentThread().isDamon()); ⟹ O/P: false
   //Thread.currentThread().setDaemon(true); → RE: ITSE
    MyThread t=new MyThread();
    S.o.p (t.isDaemon()); ⟹ O/P: false
    t.setDaemon(true);
    S.o.p (t.isDaemon()); ⟹ O/P: true
  }
}
```

→ Once last non-daemon thread terminates automatically all daemon threads will be terminated.

Ex:

```
class MyThread 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(-)
  {
    MyThread t=new MyThread();
    t.setDaemon(true); → ①
    t.start();
    S.o.p ("End of main");
  }
}
```

→ If we comment line ① then both main & child threads are non-daemon.

→ Hence both threads will be continued until their completion.

→ If we are not commenting line ① then main thread is non-daemon, but child thread is daemon.

→ Hence whenever main thread terminates automatically child thread will be terminated.

→ In this case, the o/p is

```
End of main
child Thread
```

Note:- Usually daemon threads run with low priority, but based on our requirement they may run with high priority also.

***

6) Synchronization:—

→ synchronized is the keyword applicable for methods & blocks, but not for classes & variables.

→ If a method or block declared as synchronized then at a time only one thread is allowed to operate on the given object so that data inconsistency problems will be resolved.

→ The main advantage of synchronized keyword is we can overcome data inconsistency problems.

→ But the main disadvantage of synchronized keyword is it increases waiting time of threads & performance will be reduced.

→ Hence if there is no specific requirement then it is never recommended to use synchronized keyword.

→ Every object in Java has a unique lock.

→ Internally synchronization concept is implemented by using lock concept.

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

→ If a thread wants to execute a 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 the given object.

→ Once synchronized method execution completes thread releases the

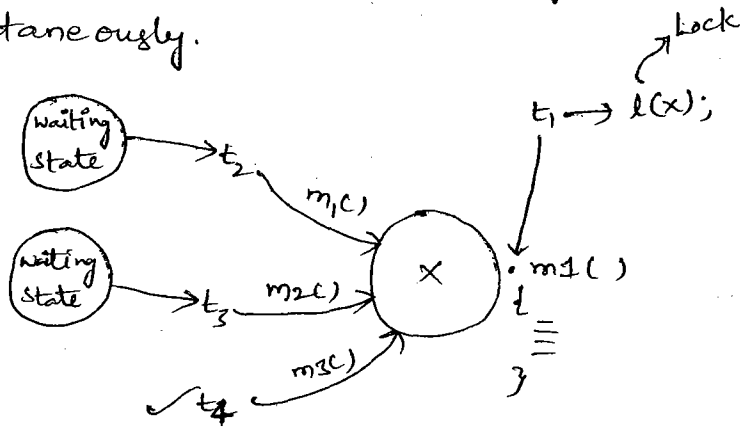
lock automatically.

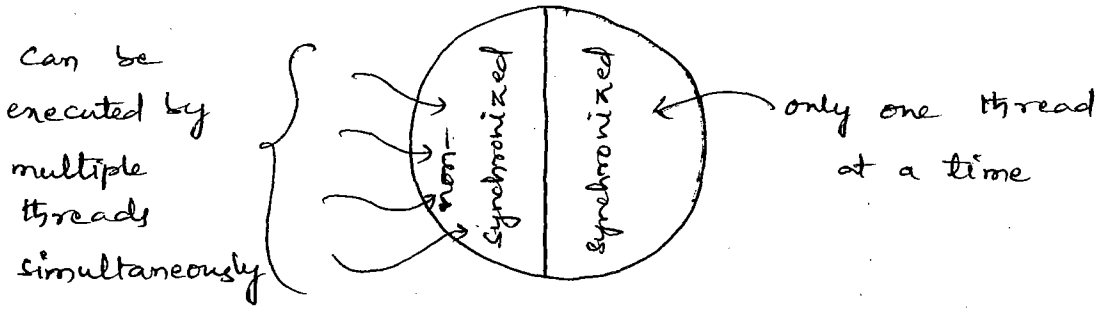
→ Acquiring & releasing the lock takes care by JVM automatically & programmer is not responsible for these things.

→ While a thread executing synchronized method on the given object the remaining threads are not allowed to execute any synchronized method on that object simultaneously.

→ But remaining threads are allowed to execute any non-synchronized methods simultaneously.

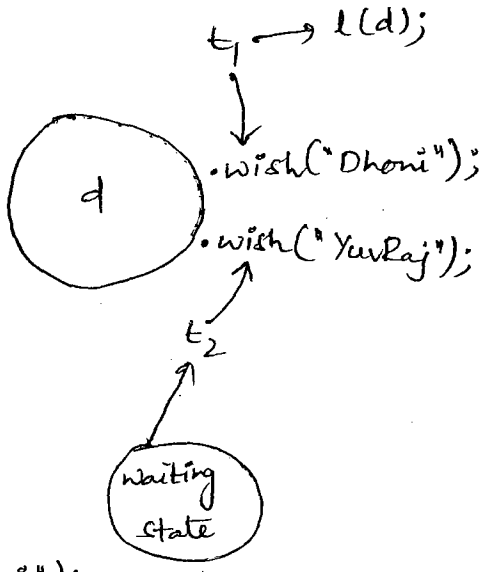
Ex:
```
class X
{
  synchronized m1()
  synchronized m2()
  m3()
}
```





Ex:
```
class Display
{
 public synchronized void wish(String name)
 {
  for(int i=0; i<10; i++)
  {
   S.o.print("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 SynchronizedDemo
{
  P s v m(-)
  {
    Display d=new Display();
    MyThread t1=new MyThread(d, "Dhoni");
    MyThread t2=new MyThread(d, "YuvRaj");
    t1.start();
    t2.start();
  }
}
```



→ If we are not declaring wish(-) method as synchronized then at a time both threads will be executed simultaneously & hence we will get irregular o/p.

```
Good Morning : Good Morning : YuvRaj
Good Morning : Dhoni
-----------
```

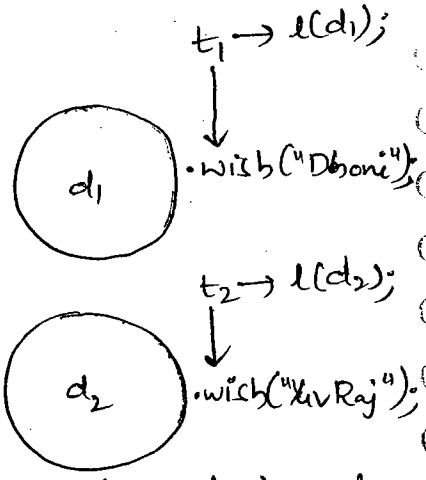
→ If we declare wish(-) method as synchronized then at a time only one thread is allowed to execute on the given Display object.

→ In this case, the o/p is

Good Morning : Dhoni
Good Morning : Dhoni
(10 times)
Good Morning : YuvRaj
Good Morning : YuvRaj
(10 times)

Case Study :-

Ex:
```
Display d1=new Display();
Display d2=new Display();
MyThread t1=new MyThread(d1,"Dhoni");
MyThread t2=new MyThread(d2,"YuvRaj");
t1.start();
t2.start();
```



→ Eventhough wish(-) method is synchronized we will get irregular o/p becoz threads are operating on different objects.

Conclusion :-

→ If multiple threads are operating on multiple objects then there is no impact of synchronized keyword.

→ But if multiple threads are operating on same Java object then there is impact of synchronized keyword.

Class level Lock :-

→ Every class in Java has a unique lock which is also known as class level lock.

→ If a thread wants to execute static synchronized method then it has to get class level lock.

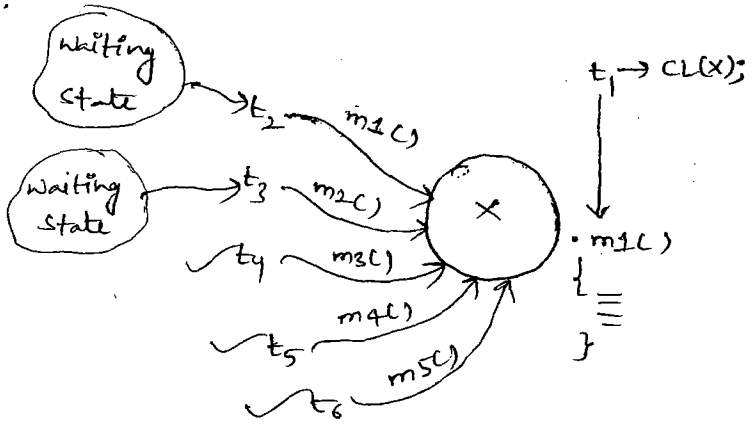
→ Once thread got class level lock then it is allowed to execute any static synchronized method.

→ While a thread executing any static synchronized method then remaining threads are not allowed to execute any static synchronized methods simultaneously.

→ But remaining threards are allowed to execute the following methods simultaneously.

1. normal static methods.
2. normal synchronized methods.
3. normal instance methods.

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



Synchronized block :—

→ If very few lines of the code requires synchronization then it is never recommend to declare entire method as synchronized. we have to enclose those few lines of the code with synchronized block.

→ The main advantage of synchronized block over synchronized method is waiting time of threads will be reduced so that performance will be improved.

→ We can declare synchronized block as follows.

1) To get lock of current object :—

```
synchronized (this)
{
  ==
  ==  → If a thread got lock of current object then only it is allowed
  ==      to execute that block.
}
```

2) To get lock of a particular object 'b' :—

```
synchronized(b)
{
  ≡≡ → If a thread got lock of b then only it is allowed
        to execute that block.
}
```

3) To get lock of Display class (Class level lock):—

```
synchronized(Display.class)
{
  ≡≡ → If a thread got class level lock of Display then only
        it is allowed to execute that block.
}
```

*** Note:— Lock concept applicable only for objects & classes but not for primitives. Hence we can pass either object reference or class name as argument to synchronized block, but not primitives otherwise we will get compile time error.

Eg:
```
int a=10;
synchronized(a)
{
  ≡≡
}
```
→ CE: unexpected type
found: int
required: reference

** FAQs:—

1. What is synchronized? and where we can apply?
2. What is the advantage & disadvantage of synchronized keyword?
3. What is synchronized method?
4. What is object lock? and when it is required?
5. While executing a synchronized method is, remaining threads are allowed to execute any other synchronized method on that object simultaneously?

Ans: No.

⑥. What is synchronized block?

⑦ Explain the advantage of synchronized block over synchronized method?

⑧ What is difference blw synchronized method & synchronized block?

⑨ Explain with an example how to declare a synchronized block to get class level lock?

⑩ What is difference blw object level lock and class level lock?

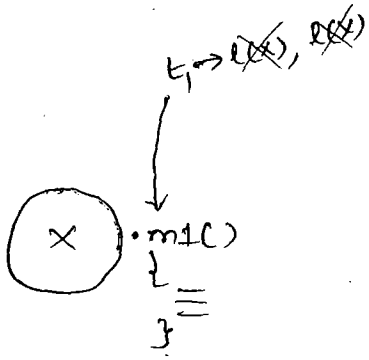
⑪ When a thread required class level lock?

⑫ Is a thread can aequire multiple locks simultaneously?

Ans:- A thread can aequire multiple locks simultaneously ofcourse from different objects.

Ex:

```
class X
{
  public synchronized void m1()
  {
    |
    |
    Y y=new Y();
    Synchronized(y)
    {
      ≡  → At this stage thread has x object lock & y object lock.
    }
    ≡
  }
  |
}
```



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

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

7) Interthread Communication :—

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

→ The thread which required updation it has to call wait() method & immediately it will entered into waiting state.

→ The thread which provides updation is responsible to call notify() method so that waiting thread will get that notification & it continue its execution with those updations.

*** → wait(), notify() & notifyAll() methods present in Object class but not Thread class becoz Thread can call these methods on any Java object.

→ To call wait(), notify() & notifyAll() methods compulsory the current thread should be owner of that object i.e., compulsory the current thread should has lock of that object i.e., compulsory the current thread should inside synchronized area.

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

→ Once a thread calls wait() method on any object it immediately releases the lock of that object & entered into waiting state. (but not all locks).

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

→ Except wait(), notify() & notifyAll() methods thread won't releases the lock anywhere else.

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 Once a thread calls wait() method immediately it releases all locks whatever it has and entered into waiting state.

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

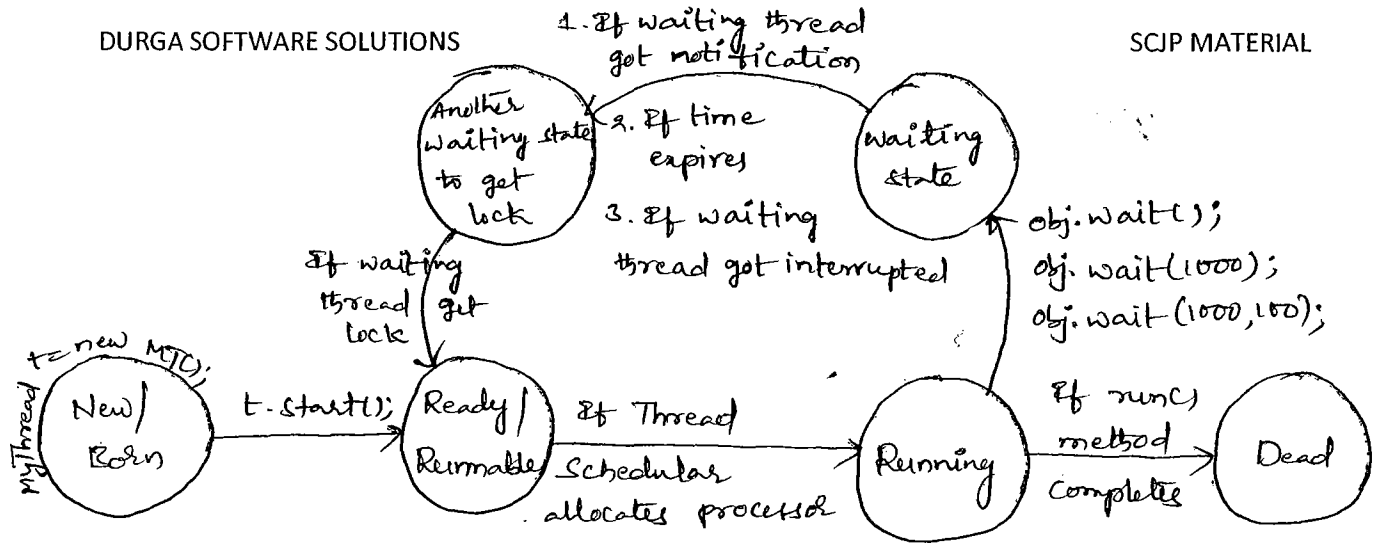
✓ ③ Once a thread calls wait() method on any object it immediately releases the lock of that object & entered into waiting state

✗ ④ Once a thread calls notify() method on any object it immediately releases the lock of that object.

✗ ⑤ Once a thread calls notify() method on any object it releases all locks acquired by that thread.

✓ ⑥ Once a thread calls notify() method on any object it releases the lock of particular object may not immediately.

```
public final void wait() throws IE
public final native void wait(long ms) throws IE
public final void wait(long ms, int ns) throws IE
public final native void notify()
public final native void notifyAll()
```



Ex: class ThreadA
{
P s v m() throws Exception
{
ThreadB b=new ThreadB();
b.start();
synchronized(b)
{
① S.o.p("Main thread calling wait method");
b.wait();
④ S.o.p("Main thread got notification call");
⑤ S.o.p(b.total);
}
}
}

Waiting state

class ThreadB extends Thread
{
int total=0;
public void run()
{
synchronized(this)
{
② S.o.p("Child thread starts calculation");
for(int i=1; i<100; i++)
{
total=total+i;
}
③ S.o.p("Child giving notification call");
this.notify();
}
}
}

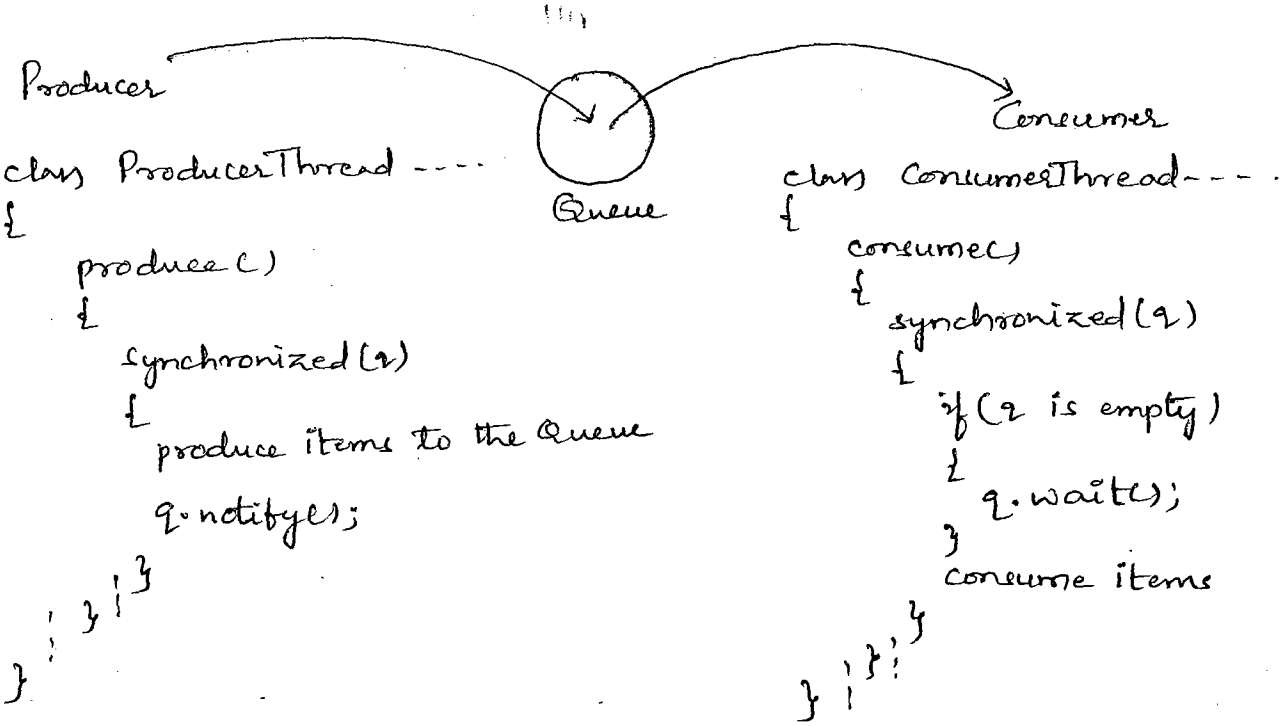
O/P: Main thread calling wait method
Child thread starts calculation
Child giving notification call
Main thread got notification call
5050.

Producer Consumer Problem:—

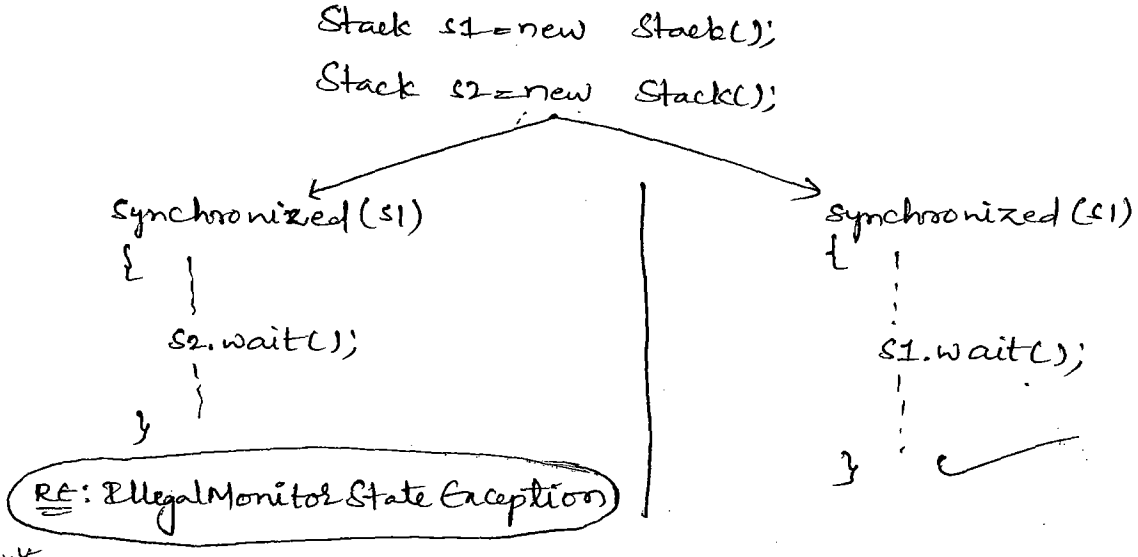
→ Producer thread will produce items to the Queue & Consumer thread will consume those items from the Queue.

→ If Queue is empty then the Consumer thread will call wait() method on the Queue object.

→ After producing items to the Queue Producer thread will call notify() method on the Queue object so that waiting Consumer will get that updation & continue its execution with those updated items.



Note:— If a thread wants to call wait() or notify() or notifyAll() on any object then thread is required to get lock of that object.



*** Difference b/w notify() and notifyAll() :—

→ We can use notify() method to give the notification for only one waiting thread. If several threads are waiting then only one thread will be notified & all remaining threads will wait for further notifications, but 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 threads of a particular object, eventhough all waiting threads got notified, but execution will happen one by one because they required lock.

8) Deadlock:—

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

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

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

Ex:

```
class A
{
  public synchronized void foo(B b)
  {
    S.o.p("Thread1 starts execution of foo() method");
    try
    {
      Thread.sleep(6000);
    }
    catch(IE e) { }
    S.o.p("Thread1 trying to call b.last()");
    b.last();
  }
  public synchronized void last()
  {
    S.o.p("Inside A, this is last() method");
  }
}
class B
{
  public synchronized void bar(A a)
  {
    S.o.p("Thread2 starts execution of bar() method");
    try
    {
      Thread.sleep(6000);
    }
    catch(IE e) { }
    S.o.p("Thread2 trying to call a.last()");
    a.last();
  }
  public synchronized void last()
  {
    S.o.p("Inside B, this is last() method");
  }
}
```

```
class DeadLock extends Thread
{
    A a = new A();
    B b = new B();
    public void m1()
    {
        this.start();
        a.foo(b); ---> Main Thread
    }
    public void run()
    {
        b.bar(a); ---> Child Thread
    }
    P S V m()
    {
      DeadLock d = new DeadLock();
      d.m1();
    }
}
```

main, child, main

o/p: Thread1 starts execution of foo() method
Thread2 starts execution of bar() method
Thread1 trying to call b.last()
Thread2 trying to call a.last()

Starvation —Vs Deadlock :—

→ A long waiting of a thread where waiting never ends is called Deadlock.

→ A 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. It's a long waiting, but that waiting ends at certain point which is nothing but Starvation.

How to stop a Thread in middle of execution:—

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

```
public void stop()
```

→ If we call stop() method on any Thread object immediately it will be entered into Dead state.

→ stop() method is deprecated & not recommended to use.

How to suspend and resume a Thread:—

→ A thread can suspend other thread by using suspend() method then immediately that thread will be entered into suspended state.

```
public void suspend()
```

→ A thread can resume a suspended thread by using resume() method then immediately suspended thread will continue its execution.

```
public void resume()
```

→ Anyway these methods are deprecated and not recommended to use.

ThreadGroup:—

→ Based on the functionality we can group threads as a single unit which is nothing but ThreadGroup.

→ ThreadGroup provides a convenient way to perform common operations for all threads belongs to a particular group.

Ex: Stop all consumer threads

For all producer threads set high priority.

→ We can create a ThreadGroup by using the following constructor of ThreadGroup class

```
ThreadGroup g = new ThreadGroup(String gname);
```

→ We can attach a thread to the ThreadGroup by using the following constructor of Thread class.

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

Ex:
```
ThreadGroup g=new ThreadGroup("printing threads");
MyThread t1=new MyThread("header printing");
MyThread t2=new MyThread("footer printing");
MyThread t3=new MyThread("body printing");
;;;;;;;;;;;;
g.stop();
```

→ Java multithreading concept is implemented by using the following 2 models.

1. Green Thread Model
2. Native OS Model

1) Green Thread Model:—

→ The threads which are managed completely by JVM without taking support from underlying OS such type of threads are called Green Threads.

2) Native OS Model:—

→ The threads which are managed with the help of underlying OS are called Native Threads.

→ Windows based OS's provide support for Native OS Model.

→ Very few operating systems like SUN Solaries provide support for Green Thread Model.

→ Anyway Green Thread Model is deprecated & not recommended to use.

ThreadLocal:—

→ We can use ThreadLocal class to define thread specific local variables like database connections, counter variables etc.