multithreading

multitasking :-

- Executing several tasks simultaneously is called as multitasking.
- There are two types of multitasking
  - (a) process based multitasking
  - (b) Thread based

(a) process based multitasking :-

Executing several tasks simultaneously where each task is seperate independent process is called as process based multitaskings

ex: while typing a java program in the editor we can able to listen mp3 audio songs at the same time we downloaded a file from the net all the these task are independent of each other & executing simultaneously & hence it is process based multitasking

- This type of multitasking is best suitable for 'os level'.

(b) Thread based multitasking :-

- Executing several task simultaneously where each task is a seperate independent part of the same program, is called as Thread based multitasking
- each independent part is called as Thread.
- This type of multitasking is best suitable for programmatic level.
- when compared with c++, developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API

imp -
- To implement multimedia graphics
- To develop animations
- To develop video game etc.
- To develop web & appl servers
- The main objective of multi-tasking is to improve performance of the system by reducing response time.

* Two ways to define instantiate & start new thread:-

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

① By extending thread class:-

Ex

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            Syso("child thread");
        }
    }
}
class ThreadDemo
{
    P S V m(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        for(int i=0; i<10; i++)
        {
            Syso("main thread");
        }
    }
}
```

Case 1) Thread scheduler:-

If multiple threads are waiting to execute then which Thread will execute 1st decided by "Thread Scheduler" which is part of JVM.

case 2) Diff bet" t.start() and t.run() methods:

- 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() no new thread will be created & run method will be executed just like a normal method by the main thread.

case3) 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 and hence we won't get any o/p.

Ex

```
class MyThread extends Thread
{
}
class ThreadDemo
{
    P S V M (String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

case4) overriding of run() method.

If we overload run method but Thread class start() method always invokes no argument run() method. the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

Ex

```
class MyThread extends Thread
{
    public void run()
    {
        SYSO("no argument method");
    }
    public vord run(int i)
    {
        SYSO("Int argument method");
    }
}
class ThreadDemo
{
    P S V M (String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

o/p→ No argument method

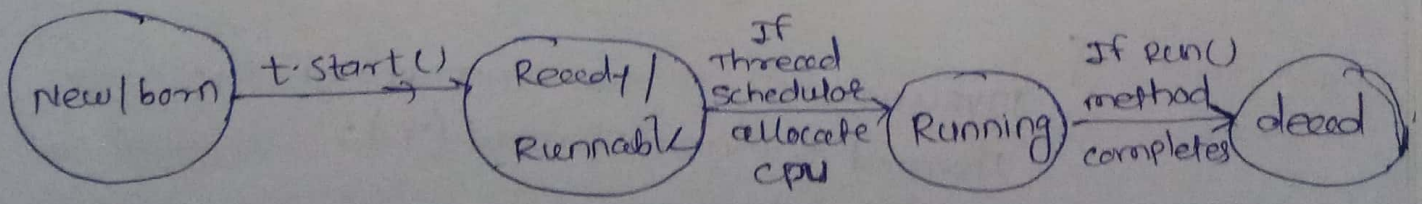
⑤ Overriding of start() method :-

If we override start method then our start() method will be executed just like normal method call and no new thread will be started.

Ex
```
class MyThread extends Thread
{
    Public void start()
    {
        SysO("start method");
    }
    Public void run()
    {
        SysO("run method");
    }
}
class ThreadDemo
{
    P S V m(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        SysO("main method");
    }
}
```
O/P → start method.
main method.

⑥ Life cycle of the thread :-

MyThread t = new MyThread();

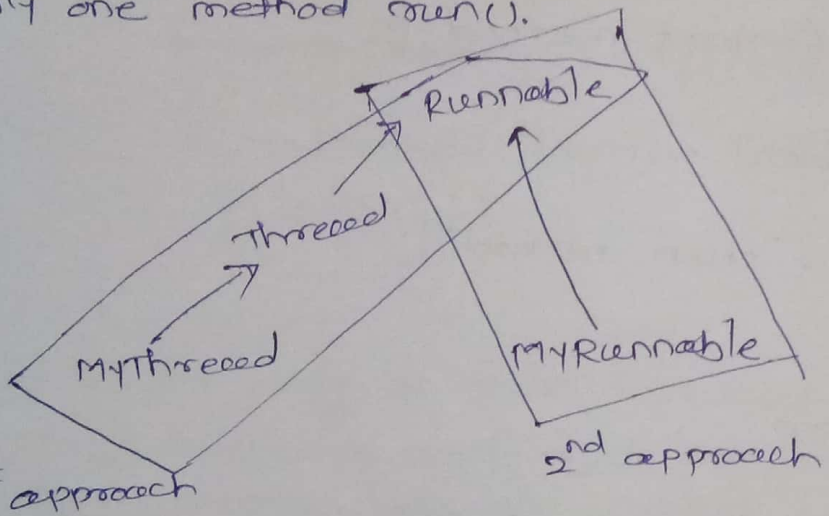


- once we created thread object then thread is said to be new/born state
- once we call start() method the thread will be entered into Ready or runnable state.

- If thread scheduler allocate CPU then the thread will be entered into running state.
- once run method completes then the thread will be entered into dead state

② By implementing Runnable interface:

- we can def^n thread even by implementing Runnable interface also.
- Runnable interface present in java.lang. pkg & contains only one method run().



Ex

```
class MyRunnable implements Runnable
{
   Public void run()
   {
      for(int i=0; i<10; i++)
       {
          Syso("child thread");
       }
   }
}
class ThreadDemo
{
   P S V m(String[] args)
   {
      MyRunnable r = new MyRunnable();
      Thread t = new Thread(r);
       t.start();
      for(int i=0; i<5; i++)
       {
          Syso("main thread");
   }   }
```

o/p → main thread (5)
child thread (10)

We can't expect exact o/p but there are several possible 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.

o/p - main thread (5)

Case 2 : t1.run();-

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

o/p → main ~~method~~ thread (5)

case 3: t2.start();

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

o/p - main ~~method~~ Thread (5)
child ~~thread~~ thread (10)

Case 4: t2.run()

no new thread will be created & MyRunnable run() method will be executed just like normal method call.

o/p- child Thread (10)
main thread (5).

Case 5:- r.start()

we will get C.E saying start() method is not available in MyRunnable class.

case 6:- r.run()

no new thread created & MyRunnable class run() method will be executed just like normal method call.

o/p - child Thread (10)
main Thread (5)

- Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.
- In 1st approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance
- But in 2nd approach while implementing Runnable interface we can extends some other class also. Hence implements Runnable mechanism is recommended to define a Thread.

Thread priorities :-

- Every Thread in java has some priority it may be default priority generated by jvm or explicitly provided by the programmer.
- The valid range of thread priority is 1 to 10. where 1 is the least priority and 10 is the highest priority
  1. Thread.MIN_PRIORITY — 1
  2. Thread.MAX_PRIORITY — 10
  3. Thread.NORM_PRIORITY — 5.
- There is no const like Thread.LOW_PRIORITY & Thread.HIGH_PRIORITY.
- Thread scheduler uses these priority while allocating CPU.
- The thread which is having highest priority will get chance for 1st execution.
- If 2nd thread having same priority then we can't expect exact execution order. it depend on thread scheduler whose behaviour is vendor dependent.
- we can get & set the priority of thread by using following methods.
  1. Public final int getPriority()
  2. public final void setPriority // allowed value are 1 to 10

The default priority only for the main thread is 5. But for all the remaining threads the default priority will be inheriting from parent to child.

Ex ①

```
class MyThread extends Thread.
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      syso("child thread");
    }
  }
}
class ThreadPriorityDemo
{
  psvm(String[] args)
  {
    MyThread t = new MyThread();
    //t.setPriority(10); ——— 1
      t.start();
      for(int i=0; i<10; i++)
      {
        syso("main thread");
      }
  }
}
```

- If we are commenting line 1 then both main & child thread will have the same priority & hence we can't exact execution order.
- If we are not commenting line 1 then ~~both threads will have the same priority & hence we can't exact execution order.~~ child thread has the priority 10 and main thread has the priority 5 hence child thread will get chance for execution & after completing child thread main thread will get the chance in this the o/p is,

o/p: child thread (10)
main thread (10);

* The method to prevent a thread from execution.

① yield(); ② join(); ③ sleep();

① yield():-

- yield() method causes to pause current executing thread for giving a chance of remaining waiting thread of same priority.
- If all waiting threads have the low priority or if there is no waiting threads then the same thread will be continued it's execution.
- If several waiting thread have same priority with available then we can't expect exact which thread will get chance for execution.

Ex

```
class MyThread extends Thread.
{
    public void run()
    {
        for (int i=0; i<10; i++)
        {
            Thread.yield();
            SYSO("child thread");
        }
    }
}
Class ThreadYieldDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        for (int i=0; i<10; i++)
        {
            SYSO("main thread");
        }
    }
}
```

o/p → main thread (10)
child thread (10).

- In above program child thread always calling yield() method & hence main thread will get chance more no of times for execution.

② Join():-

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

EX - If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.

- Every join() method throws interrupted exception, which is checked exception hence compulsory we should handle either by try catch or by throws keyword. otherwise we will get compile time error.

EX

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            Syso("child thread");
            try
            {
                Thread.sleep(2000);
            }
            catch (InterruptedException e)
            {
            }
        }
    }
}
class ThreadJoinDemo
{
    p s v m (String[] args) throws InterruptedExcepto
    {
        MyThread t = new MyThread();
        t.start();
        //t.join(); ——— ①
        for(int i=0; i<5; i++)
        {
            Syso("main Thread");
        }
    }
}
```

- If we are commenting line 1 then both threads will be executed simultaneously & we can't expect exact execution order.
- If we are not commenting line 1 then main thread will wait until completing child thread in this the o/p is child thread 10 times followed by main thread 5 times

Waiting of child thread until completing main thread;

ex

```
class MyThread extends Thread
{
    public void run()
    {
        try
        {
            mt.join();
        }
        catch (InterruptedException e) { }
        for (int i=0; i<5; i++)
        {
            SYSO("child thread");
        }
    }
}
class ThreadJoinDemo
{
    P S V m(String[] args) throws InterruptedException
    {
        MyThread t = new MyThread();
        MyThread.mt = Thread.currentThread();
        t.start();
        for (int i=0; i<5; i++)
        {
            Thread.sleep(2000);
            SYSO("main thread");
        }
    }
}
```

o/p → main thread (5)
child thread (5)

③ sleep() method :-

If a thread don't want to perform any operat" for a particular amount of time then we should go for sleep() method.

① Public static native void sleep (long ms) throws IE
② Public static void sleep (long ms, int ns) throws IE.

Ex

```
class ThreadJoinDemo
{
    P S V m (String[] args) throws IE
    {
        SysO ("M");
        Thread.sleep(2000);
        SysO ("E");
        Thread.sleep (2000);
        SysO ("G");
        Thread.sleep (2000);
        SysO ("A");
    }
}
```

o/p → M
E
G
A

Interrupting a thread :-

How a thread can interrupt another thread?

If a thread can interrupt a sleeping or waiting thread by using interrupt() (break off) method of thread class.

Ex

```
class Mythread extends Thread
{
    public void run() {
    { try{
        for (int i=0; i<10; i++)
        {
            SysO("I am lazy thread"+i);
            Thread.sleep (2000);
    } }
```

```
catch(IE e)
  {
    Syso("I got interrupted");
  }
 }
}
class ThreadInterruptDemo
 {
   P S V m(String[] args)
   {
     MyThread t = new MyThread();
     t.start();
    // t.interrupt(); ——— (1)
     Syso("end of main thread");
```

- If we are commenting line 1 then main thread won't interrupt child thread & hence child thread will be continued it's completion

o/p- ~~end of main thread~~
@I am lazy thread (10)
①end of main thread.

- If we are not commenting line 1 then main thread interrupts child thread & hence child thread won't continued untill it's completion in this case o/p is

o/p→ end of main thread.
I am lazy thread.
I got interrupted.

- whenever we are calling interrupt method we may not see the effect immediately, if target thread is sleeping or waiting state it will be interrupted immediately
- If target thread is not in sleeping or waiting state then interrupt call will exist untill target thread will enter into sleeping or waiting state. once target thread enter into sleeping —↓— it will effect immediately

Difference betn yield(), sleep() & join() method ?

① yield():

- The purpose of yield() method to pause current executing thread for giving a chance of remaining waiting threads of same priority
- yield() method is static
- yield() method is not final.
- ——→ is not overloaded
- yield() method not throws IE (Interrupted exep°)
- yield() method is native method

② join():-

- The purpose of join method, If thread wants to wait until completing some other thread then we should go for join() method.
- join method is not static
- join() ——→ final
- ——→ overloaded
- ——→ throws Interrupted exception
- ——→ has no native method.

③ sleep():-

- The purpose of sleep() method. If thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
- sleep method is a static method.
- ——→ not final method
- ——→ overloaded
- ——→ throws Interrupted exception.
- sleep has native & non-native method.
  sleep(long ms) ——→ native
  sleep(long ms, int ns) ——→ non-native

Synchronization :-

- Synchronized keyword applicable for method & blocks but not class & variables.
- If thread of block declared as a synchronized then at a time only one thread is allow to execute that method or block on the given object
- The main advantage of synchronized keyword is we can resolve ~~data~~ inconsistency problem.
- But the disadvantages of synchronized keyword is it increases waiting time of the ~~~~ thread & effect performance of the system.
- Hence if there is no specific requirement then never recommended to use synchronized keyword.
- Internally synchronization concept is implemented by using lock concept.
- Every object in java has unique lock. whenever we are using synchronized keyword then only lock concept ~~~~ will come into picture.
- If thread want't to execute any synchronized method on the given object 1st it has to get the lock of that object. once thread got the lock of that object then it's allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically thread release lock.
- while thread executing any synchronized method the remaining threads are not allowed execute any synchronized method on that object simultaneously. But remaining thread allow to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method]

Ex

```
class Display
{
    Public synchronized void wish(String name)
    {
        for(int i=0; i<5; i++)
        {
            sysO("good morning");
            {
                try
```

```
        {
            thread.sleep(2000);
        }
        catch (IE e) { }
        Sys o(name);
    }
  }
}
class MyThread extends thread.
{
    Display d;
    string name;
    MyThread (Display d, string name)
    {
        this.d = d;
        this.name = name;
    }
    public void run()
    {
        d.wish(name);
    }
}
class SynchronizedDemo
{
    p s v m(string[] args)
    {
        Display d1 = new Display();
        MyThread t1 = new MyThread(d1, "Akshay");
        MyThread t2 = new MyThread(d1, "Bhosale");
        t1.start();
        t2.start();
    }
}
```

If we declare wish() method as synchronized then the thread will be executed one by one that is until completing the 1st thread 2nd thread will wait in this case we will get regular o/p

o/p → good morning : Akshay (5)
good morning : Bhosale (5)

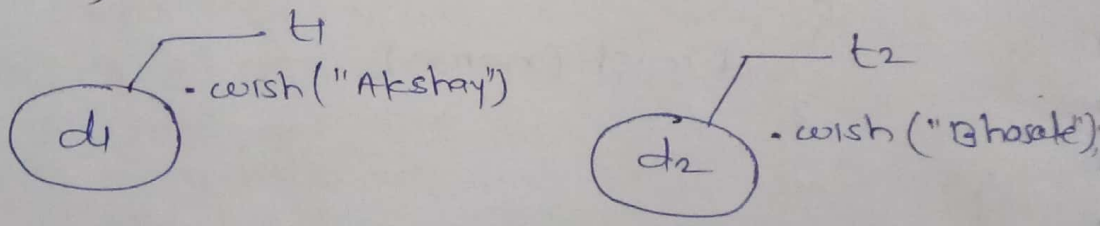
If we are not declaring wish() method as synchronized then both threads will be executed simultaneously & we will get irregular o/p.

o/p → good morning : good morning : Bhosale
good morning : Akshay
: Bhosale
⋮

Case study :

① Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Akshay");
MyThread t2 = new MyThread(d2, "Bhosale");
t1.start();
t2.start();

Diagram :-



Even though we declared wish() method as synchronized but we will get irregular o/p in this case, because both thread operating on diff objects.

Conclusion :-

- If multiple threads are operating on multiple object then there is no impact of synchronization.
- If multiple threads are operating on same java objects then ~~th~~ synchronized concept is applicable

Class level Block :-

- every class in java has unique lock. If thread wants to execute a static synchronized method then it required class level lock
- once thread got class level lock then it is allow to execute any static synchronized method of that class

- while thread executing any static synchronized method the remaining threads are not allowed to execute any static synchronized method of that class simultaneously.
- But remaining thread allow to execute normal synchronized method, normal static methods & normal instance methods simultaneously.
- class level lock & object lock both are different & there is no relationship bet^n these two.

* Synchronized Block:-

- If very few lines of the code required synchronization then its never recommended to declare entire method as synchronized we have to enclose those few lines of the code with in synchronized block.
- main advantages is it reduced the waiting time of thread & improves performance of system.

Ex 1:- To get lock of current object we can declared synchronized block
- If thread got lock of current object then only it is allowed to execute this block

```
synchronized (this){ }
```

Ex 2:- To get the block of particular object 'b' we have to declared a synchronized block as follows:
- If thread got lock of 'b' object then only it is allowed to execute this block.

```
synchronized (b){ }
```

Ex 3:- To get class level lock we have to declare synchronized block. as follows

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

- If thread got class level lock of Display then only it allow to execute this block

As the argument of synchronized block we can pass either object reference or "class file" & we can't pass primitive values as argument [bcz lock concept dependent on objects and classes but not for primitives]

Ex

```
int x = 6;
synchronized (x){}
```

o/p → compile time error

Q - what is object lock & when it is required.?

→ object lock means every object in java has unique lock is nothing but object lock.
- whenever thread wants to execute instance synchronized method

Q - what is class level lock & when it is required?

→ every class in java has unique lock which is nothing but class level lock
- whenever thread wants to execute static synchronized method.

* Inter Thread Communication (wait(), notify(), notifyAll()):
- Two threads can communicates with each other by using wait(), notify() & notifyAll() methods.
- The thread which is required updation it has to call wait() method on the required object then immediately the thread will entered into waiting state.
- The thread which is performing updation of object, it is responsible to give notification by calling notify() method.
After getting notification the waiting thread will get those updations.
- wait(), notify() & notifyAll() methods are available in object class but not in thread class because thread can call these methods on any common object
- To call wait(), notify(), notifyAll() methods compulsory the thread should be owner of that object.
i.e. current thread should has lock of that object
i.e. current thread should be in synchronized area.

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

- once thread calls wait() method on the given object it releases the lock of that object but may not immediately
- except these wait(), notify(), notifyAll() methods there is no other place where the lock release will be happen.

Ex

```
class ThreadA
{
    P S V m(String[] args) throws IE
    {
        Thread B = new ThreadB();
        b.start();
        synchronized(b)
        {
            S+so("main thread calling wait() method"); -①
            b.wait();
            S+so("main thread got notification call"); -④
            S+so(b.total);
        }
    }
}
class ThreadB extends Thread.
{
    int total = 0;
    Public void run()
    {
        synchronized (this)
        {
            S+so("child thread starts calculation"); -②
            for(int i=0; i<=100; i++);
            {
                total = total + i;
            }
            S+so("child thread giving notification call");
            this.notify(); -③
        }
    }
```

o/p → main thread calling wait() method
child thread starts calculation
——+—— giving notification call
main thread got notification call.
5050.

Notify VS NotifyAll():-

- We can use notify() method to give notification for only one thread. If multiple threads are waiting then only one thread get the chance & remaining threads has to wait for further notification.
- We can use notifyAll() method to give the notification for all waiting threads. All waiting threads will be notified & will be executed one by one, because they are required lock.

* Deadlock:-

- If two threads are waiting for each other forever (without end) such type of situation is called as deadlock.
- There are no resolution techniques for deadlock but several preventation techniques are possible.
- Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.
- If we remove atleast one synchronized keyword then we won't get Deadlock. Hence synchronized keyword in the only reason for Deadlock due to this while using synchronized keyword we have to handling carefully

* Daemon Threads:-

- The Threads which are executing in the background are called as daemon threads.
- the main objective of daemon threads is to provide support for non-daemon threads like main thread.
- A long waiting thread which never ends is called Deadlock.
- A long waiting thread which ends a certain point is called as starvation.