***Program*** *: Set of instructions*

***Application*** *: Collection of programs*

***Process*** *:*

* *Step by step execution of an application*
* *Process is the part of Operating system (OS based)*

*eg. java code > compile > .class file is created > execution/run > output*

***Thread*** *:*

* *Multiple threads run in a single process*
* *It is sub-part of a process*

***Process*** *: So process can also be defined as collection/group of threads.*

*-* ***Difference between process and thread***

***process*** *: Step by step execution of an application or collection/group of threads*

*Time consuption to execute is program is more*

*Heavy weight*

*Intercommunication between two different process is time consuming*

*Context switching*

*thread : Sub part of a process*

*Time consumption is less*

*Light Weight*

*Intercommunication between two different threads takes less time as it is happening within a single process*

*Multitasking :*

*execution/performing multiple task at a single time*

*Multitasking is performed in 2 ways :*

*1) Process level multitasking/multiprocessing -> OS based*

*eg. Coding in STS an listening songs on youtube simultaneously these both are different processes (there is no dependency on each other)*

*2) Thread level multitasking/multithreading -> Program level(Different program is written for execution of each thread)*

*thread based multitasking*

*eg. games, netflix movie*

***Multitasking****:- Executing several tasks simultaneously is a concept of Multitasking.*

*There are 2 types of Multitaskng:-*

1. ***Process-based:-*** *Executing several tasks simultaneously where,* ***each task is a separate independent program(process)*** *is called process-based multitasking.*

***Eg:****-While typing a java program on IDE we can listen sounds on same system, at same time we can download a file from internet, all these tasks will be executed simultaneously and independent of each other. Hence it is process-based.*

***Process-****based multitasking is best suitable at* ***OS level.***

1. ***Thread-based:-*** *Executing several tasks simultaneously where,* ***each task is a separate independent part of same program*** *is called process-based multitasking. Each independent part is called* ***Thread.***

***Thread-*** is best suitable at **Programmatic level**.

* *Whether it is process based or thread-*based the main objective is to reduce response time of system and improve performance.

The main important applications areas of Multithreading are:-

1. Develop multimedia graphics.
2. Develop animations
3. Develop video games.
4. Develop web/application servers,etc

When compared with old languages , developing multithreaded applications in Java is very easy because **Java provides inbuilt support for multithreading with rich API.(Thread, Runnable,ThreadGroup…)**

**We can define Thread in 2 ways:-**

1. By extending **Thread class. 🡪java.lang**
2. By implementing **Runnable (Functional Interface)**. 🡪 **java.lang**
3. By **lambda expression**
4. By **Executor Service**

**-----------------------------** By extending **Thread class -------------------------------------------------------**

***Case 01: Thread Scheduler:-***

* *It is responsible to schedule threads. i.e If multiple threads are waiting to get chance of getting executed, then in which order threads will be executed is decided by Thread scheduler.*
* *We can’t except exact algorithm followed by ThreadScheduler, it varies from JVM to JVM. Hence we cannot except thread execution order and exact output.*
* *Hence whenever situation comes to multithreading there is no guarantee of exact output, but we can provide several possible outputs.*

***Case 02: Difference 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() a new thread wont be created and run() will be executed just like normal method called by main thread.*

***Case 03:******Importance of thread.start() in multithreading:-***

* *.start() is responsible to register thread with thread scheduler and all other mandatory activities. Hence, without executing thread.start() there is no chance of starting a new thread in java.*
* *Due to this thread class start() is considered as heart of multithreading.*
* *Once start() method is called -> (1) Register the thread with ThreadScheduler ->(2) Perform all other mandatory operations -> (3) Invoke run() method*

***Case 04:******Overloading of run() :-***

* *It is always possible but Thread class start() will invoke run method with no parameters only.*
* *We will have to call other overloaded method explicitly as normal method.*

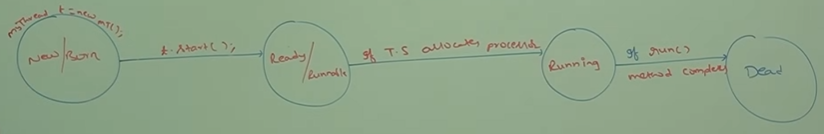
***Case 05: Overriding of run() :-***

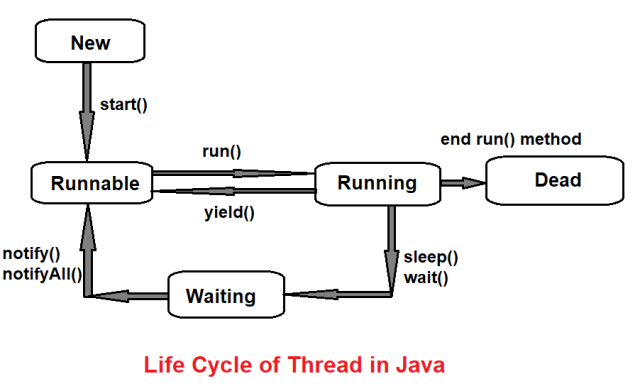
* *If we are not overriding run() method then, thread class run() method will be executed which has empty implementation. Hence, we wont get any output.*
* *It is highly recommended to override run() method otherwise don’t go for multithreading.*

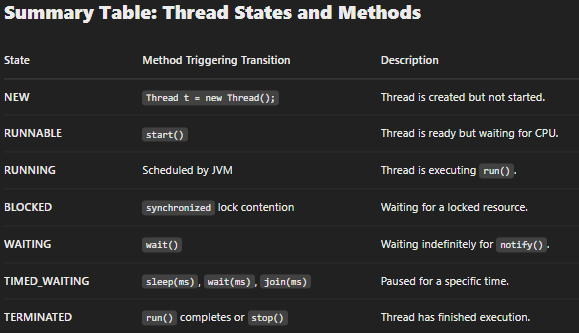
***Case 06: Overriding of start() :-***

* *If we override start() then our start() will be executed just like a normal method call and new Thread won’t be created. If we are not overriding start() then Thread class start() is invoked.*
* *It is not recommended to override start() otherwise , don’t go for Multithreading.*

***Case 07: Thread Lifecycle:-***





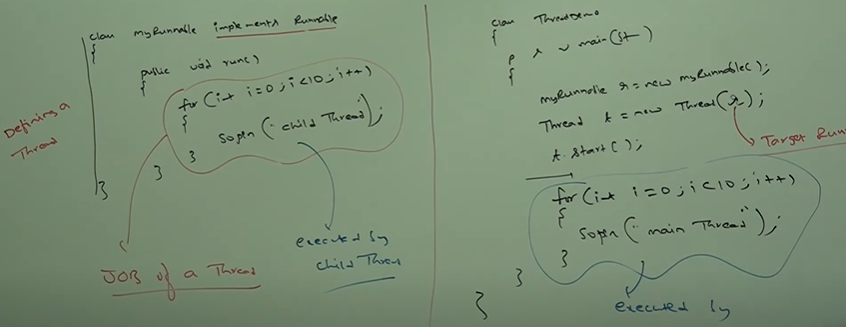


***Case 08:***

* *After starting a thread if we are trying to restart same thread then we will get* ***Runtime Exception saying IllegalThreadStateException.***

***------------------------------------------------*** By implementing **Runnable interface**.-----------------------------------

* *We can define a thread by* ***implementing Runnable Interface.***
* *Runnable interface present in java.lang package*
* *Contains only one method 🡪 run().*



*MyRunnable r=new MyRunnable();*

*Thread t1=new Thread();*

*Thread t2=new Thread(r);*

***Case 01:- t1.start()*** *🡪A new thread will be created which is responsible for execution of Thread class run() method. Which has empty implementation.*

***Case 02:- t1.run() 🡪*** *No new thread will be created and thread class run() method will be executed, just like normal method call.*

***Case 03:- t2.start()*** *🡪 A new thread will be created which is responsible for execution of MyRunnable class run() method.*

***Case 04:- t2.run()*** *🡪 No new thread will be created and MyRunnable run() method will be executed, just like normal method call.*

***Case 05:- r.start()*** *🡪 We will get Compile time error saying, MyRunnable class doesn’t have start capability.* ***CE: cannot find symbol***

***Case 06:- r.run()*** *🡪 No new thread will be created and MyRunnable run() method will be called.*

***Which approach is best to define a Thread?***

*🡪*

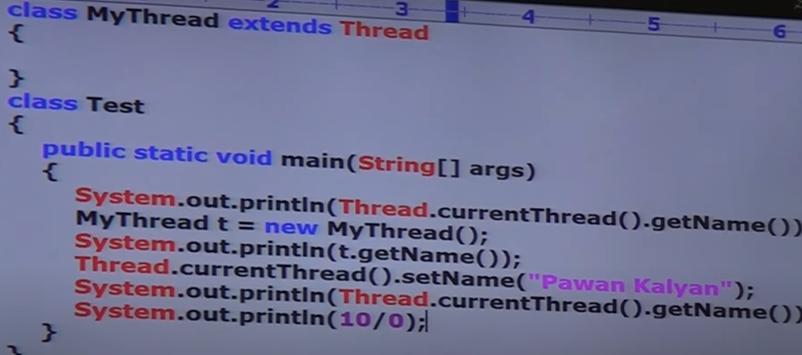
* *In Extends Thread approach there is no chance of extending any other class, hence we are missing Inheritance benefit.*
* *But in Implements Runnable Interface approach we can extend any other class hence we wont miss any inheritance benefit.*
* *Because of this reason Implementing Runnable Interface approach is best recommended*

***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);*

***Getting and Setting name of Thread:-***

* *Every thread in java has some name, it maybe default name generated by JVM or Customised name provided by Programmer.*
* *We can set and get name of thread by using 2 methods of Thread class:-*
* ***Public final String getName();***
* ***Public final void setName(String name)***



*The last line will generate exception as:* ***exception in thread Pawan Kalyan:***

***Thread Priorities:-***

* *Every Thread in java has some priority, it may be defualt priority generated by JVM. Or custimised priority provided by programmer.*
* *The valid range of Thread priorities is 1 to 10****. 1->Min\_Priority , 10->Max\_Priority***
* *Thread class defines following constants to represent some standard priorities,*

***Thread.MIN\_PRIORITY 🡪1***

***Thread.NORM\_PRIORITY🡪5***

***Thread.MAX\_PRIORITY🡪10***

***Thread.currentThread() 🡪 To get details of current thread***

* *Thread scheduler will use priorities while allocating processor.*
* *The thread which is having highest priority will get the first chance to be executed.*
* *If two threads same priority then , we cant expect exact execution order it depends on Thread Scheduler.*
* *Thread class defines following methods to get and set priority of a thread,*
  + ***Public final int getPriority();***
  + ***Public final void setPriority(int p);*** *p value should be between (1 to 10)*

*Otherwise* ***Runtime Exception->IllegalArgumentException***

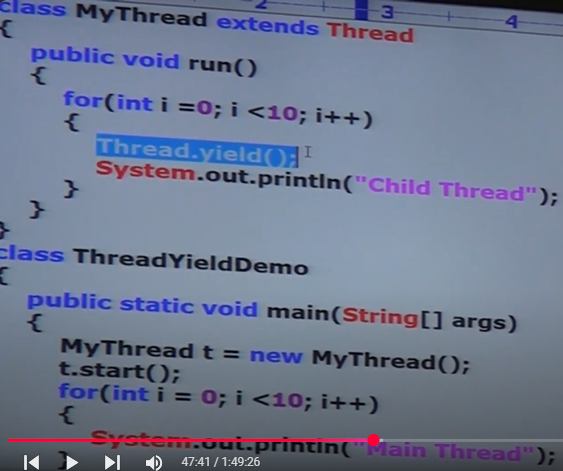
* *Default priority for* ***only main thread is 5****, but for all remaining threads* ***default priority will be inherited from parent to chid.*** *Whatever priority parent thread has will be same for child thread.*
* *Some old platforms wont provide support for Thread priorities.*

*Preventing Thread from execution* ***:- yield(), join()***

* *We can prevent thread execution by using following methods,*

***Yield****()->*

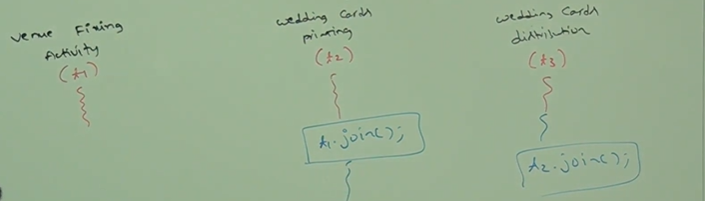
* + *This method causes to pauses current exeuting thread, to give chance for waiting threads of same priority.*
  + *If there is no waiting thread or all waiting threads have low priority then same thread can continue its execution.*
  + *If multiple threads are waiting with same priority, then which thread will get chance we cant predict it totally depends on Thread Scheduler.*
  + *The thraed which is yielded , when it will get chance once again it depends on Thread scheduler, we cant predict which will get executed.*
  + ***public static native void yield()*** *--Syntax*



* + *In the abve program we are commenting highlighted line then both threads will be executed simultaneously, then we cant expect which thread will complete first.*
  + *If we are not commenting highlighted line then child thread always calls yield(), because of that main() thread will get chance more no of time, and the chance of completeing main thread first is high.*
  + *The thread which requires more execution time uses Thread.yield()*
  + *Till java 5 yeild() was calling Thread.sleep() internally but after java 5 whenever yeild() is called ThreadScheduler decides which Thread should be prioritised , despite of we have provided the priority.*

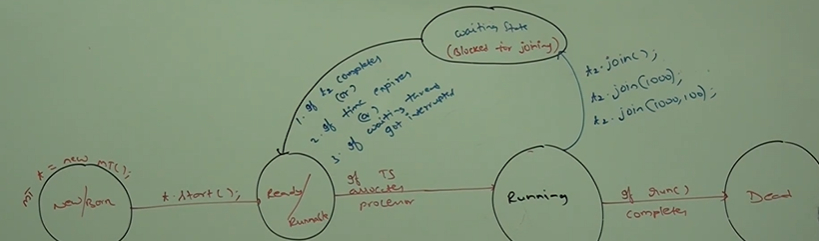
***Join()->***

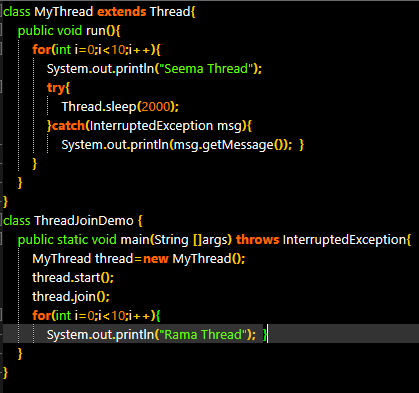
* + *If a thread wants to wait until completing some other thread then we should go for join().*
  + *For eg:- If a thread t1 wants to wait until completing t2 then t1 needs to call t2.join();*
  + *If t1 executes t2.join(), then immediately t1 will enter into waiting state until t2 completes.*
  + *Once t2 completes then t1 can continue its execution.*



* + *Wedding cards printing thread(t2) has to wait until venue fixing thread(t1) completion. Hence t2 needs to call t1.join(). Wedding cards distribution thread(t3) has to wait until wedding cards printing thread(t2) completion. Hence t3 has to call t2.join().*
  + ***public final void join() , public final void join(long ms)***
  + ***public final void join(long milliSeconds, int nanoSeconds)***
  + *Every join methods* ***throws CheckedException****:-* ***InterruptedException***

*Hence compusorily we need to handle this exception either by try-catch, throws .*

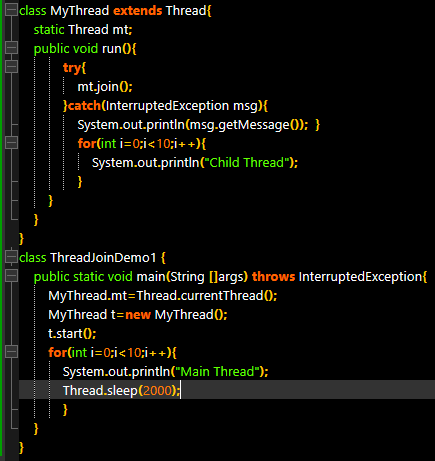




***Case 01:-***

* + *If we comment line “ thread.join() “ then both main and child threads will be executed simultanenously, and we cant expect exact output.*
  + *If we are not commenting that line then main thread calls join method on child thread object. Hence main thread will wait until completing child thread.*
  + ***This will give output Seema Thread ->10times, Rama Thread->10 times***
  + *If we pass thread.join(10000) then main thread will only wait for 10 seconds thus will execute child thread for 10 sec only and after that will execute main thread.*
  + ***This give output Seema Thread->5 time, Rama Thread-> 10 times, Seema ->5time***

***Case 02:-***



* + *In above eg, Child thread calls join() method on main thread object. Hence, child thread has to wait until completing main thread. In this case output is:*

***Main Thread ->10 times, Child Thread ->10 times***

***Case 03:-***

* + *If main thread calls join() method on child thread object and child thread calls join() method on main thread object, then both threads will wait forever. And the program will be stucked.(This is something like deadlock).*

***Case 04:-***

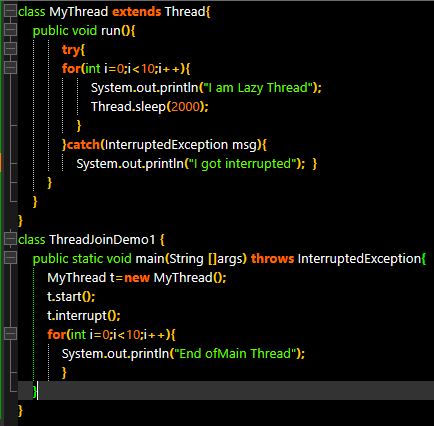
* + *If a thread calls join() method on same thread itself then program will be stucked. This is like deadlock. In this case Thread has to wait infinite amount of time.*

***Sleep():-***

* + *If a thread don’t want to perform any operation for a particular amount of time then we should use sleep method.*
  + ***Public static native void sleep(long ms),******public static void sleep(long ms, int ns)***
  + *Both methods throws InterruptedException*

***How a thread can Interrupt another Thread?***

* *A thread can interrupt sleeping thread or waiting thread by using interrupt() of Thread class.* ***public void interrupt()***

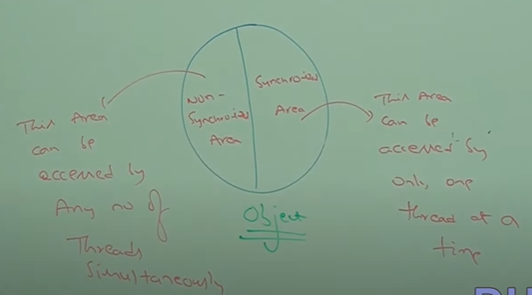


* + *If we comment line t.interrupt() then main thread wont interrupt child thread. In this case child thread will execute for loop 10 times.*
  + *If we are not commenting line t.interrupt() then main thread interrupts child thread, in this case output is:-* ***End of Main Thread , I am Lazy Thread, I got interrupted***
  + *Whenever we are calling interrupt method, if the target thread is not in sleeping state or waiting state then there is no impact of interrupt() immediately. Interrupt call will be waited until target thread entered into sleeping or waiting state. If the target thread entered into sleeping,waiting state then immediately interrupt() will interrupt the target thread.(Not allowing a thread to sleep)*
  + *If the target thread never entered into sleeping or waiting state in lifetime then there is no impact of interrupt(). This is only case where interrupt() call will be wasted.*

|  |  |  |  |
| --- | --- | --- | --- |
| ***Property*** | ***Yield()*** | ***Join()*** | ***Sleep()*** |
| ***Purpose?***  ***When to use?*** | *If a thread wants to pause execution to give chance to remaining waiting threads of same priority.* | *If a thread wants to wait until completion of other thread execution.* | *If a thread don’t want to perform any operation for particular amount of time.* |
| ***Is it overloaded?*** | *No* | *Yes* | *Yes* |
| ***Is it final?*** | *No* | *Yes* | *No* |
| ***Throws IE?*** | *No* | *Yes* | *Yes* |
| ***Is it native?*** | *Yes* | *No* | *Sleep(long ms)->Yes*  *Sleep(long ms,int ns)->No* |
| ***Is it static?*** | *Yes* | *No* | *Yes* |

***Synchronization():-***

* *Synchronized is the modifier applicable only for methods and blocks. But not for calsses and variables.*
* *If multiple threads are trying to operate simultaneously on same java object then there may be a chance of Data inconsistency problem.*
* *To overcome this problem we should go for synchronized keyword.*
* *If a method/block declared as synchronized, then at a time only one thread is allowed to execute that method/block on the given object. So data inconsistency problem will be solved.*
* ***The main advantage of synchronized keyword is:-***
  + *We can resolve data inconsistency problems.*
* ***The main diadvantage of synchronized keyword is:-***
  + *It increases waiting time of threads and creates performance problems.*
* *Internally synchronization 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 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 got the lock then it is allowed to execute any synchronized method on that object.*
* *Once method execution completes automatically thread releases lock.*
* *Acquiring and releasing lock internally taken care by JVM and programmer not responsible.*
* ***While a thread executing synchronized method on given object, the remaining threads are not allowed to execute any synchronized method simultaneously on same object. But remaining threads are allowed to execute non-syncrhonized methods simultaneously.***
* *Lock concept is implemented based on object but not based on method.*



***Class RailwayTicket{***

***Synchronized Area {***

*Wherever we are performing update, operation(****addUser(), removeUser(), delete(), replaceSeats(),bookSeats()).*** *Where state of object is changing. So such methods should always be synchronized.*

***}***

***Non-Synchronized Area {***

*Wherever object state wont be changed like* ***read(), viewAvaialbleTickets, view Trains(),*** *or read display operations performed such methods should be non-synchronized.*

***}***

***}***

* *Sequence of thread execution/ Thread scheduling is totally dependant on JVM/ Thread scheduler (FCFS,Round-Robin Algo)*

*If we are not declaring wish method as synchronized then both threads will be executed simultaneoulsy, hence we will get irregular output.*

* *If we declare wish method as synchronized then at a time only one thread is allowed to execute wish method on given display object.Hence we will get regular output*

***Ref :- Multithreading Project in Eclipse Workspace***

***Case 01:-***

* *Eventhough wish method is synchronized we will get irregular output because threads are operating on different java objects****.***
* *If multiple threads are operating on same Java object then only synchronization is required.*
* *If multiple threads are operating on multiple jva objects then synchronization is not required.*

***Class-Level Lock:- (Comes in picture when synchronized method is static)***

* *Every class in java has unique lock which is nothing but class-level lock.*
* *If a thread wants to execute static synchronized method, thread requires class-level lock.*
* *Once a thread gets class-level lock then it is allowed to execute any static synchronized method of that class.*
* *Once method executon completes automatically , thread releases lock.*
* *While a thread is 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 following methods simultaneously:-*

1. *Normal static methods*
2. *Synchronized instance methods.*
3. *Normal instance methods.*

***Synchronized block{} :-***

* *If very few lines of coe require synchronization then it is not recommended to declare entire method as synchronized. We have to enclose those few lines of code by using synchronized block*
* *Main advantage of synchronized block over method is ,*
  + *It reduces waiting time of thread.*
  + *Thus improves performance of system.*
* *We can declare symchornized block as follows,*
  + *To get lock of current object ->****synchronized(this)***
  + *To get lock of particular object->****synchronized(obj)***
  + *To get class level lock ->****synchronized(Display.class)***
* *Lock concept applicable for object Type and class type but not for Primitive type, hence we cannot pass primitive object to synchronized block which will give CompileTime Error.*

***IQ) What is synchronized keyword? Where can we apply?***

*🡪It is a modifier applicable for methods, blocks but not for variables and classses.*

***IQ) What are advantages and disadvantages fo synchronized keyword?***

***IQ)What is Race condition?***

*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 using synchronized keyword.*

***IQ) What is object lock & When is it required?***

*Every object in java has a unique lock, which is called as object lock and whenever thread wants to execute, synchronized method, then thread requires object lock.*

***IQ)What is class-level Lock & When it is required?***

*Every class in java has a unique lock, which is called as class lock and whenever thread wants to execute, static synchronized method, then thread requires object lock.*

***IQ) While a thread executing synchronized method on given object , is the remaining to execute any other synchronized method simulatneously on same object*** *->No*

***IQ)Can a thread acquire multiple locks simultaneously.***

*Yes ofcourse from different objects classX 🡪Yy=new Y()-> syncronized(y)(Has locks of x and y)->z z=new z()->synchronized(z) (Has locks of x,y,z)*

***IQ)What is synchronized statement?***

*There is no such terminology as synchronized statement , yes we can call the statements inside synchronized methods/blocks as synchronized statement.*

***Inter-Thread communication:-***

* *Two threads can communicate with each other by using wait(), notify() and notifyAll().*
* *Thread which is expecting updation is responsible to call wait() then immediately the thraed will enter into waiting state.*
* *The thread which is responsible to perform updation ,it is responsible to call notify().Then waiting thread will get that notification and continue its execution with those updated items.*

***IQ)Why Wait(), notify(), notifyAll() these methods are in object class and not in Thread class?***

* *Wait(), notify(), notifyAll() present in object class but not in thread class because thread can call these methods on any java object.(Stack, Queue, Student, Employee,etc)*
* *To call Wait(), notify(), notifyAll() methods on any object, thread should be owner of that object i.e the thread should have lock of that object. Thread should be inside synchronized area. Hence we can call Wait(), notify(), notifyAll() only from synchronized area.Otherwise we will get* ***RuntimeException:IllegalMonitorStateException***
* *If a thread calls wait() on any object it immediately releases lock of that particular object and enters into waiting state.*
* *If a thread calls notify() on any object it releases the lock of that object but may not immediately, except Wait(), notify(), notifyAll() there is no other method where thread releases lock.*

***Which of the following are Valid / Invalid? Conclusions***

* *If a thread calls wait () immeditely it enters into waiting state without releasing any lock.--****>Invalid***
* *If a thread calls wait() it releases the lock of that object but may not immediately-****>Invalid***
* *If a thread calls wait() on any method it releases all locks acquired by that thread and immediately enter into waiting state****. ->Invalid***
* *If a thread calls wait() on any object it immediately releases lock of that object and enter into waiting state -****>Valid***
* *If a thread calls nofity() on any object it immediately releases lock of that particular object* ***->Invalid***
* *If a thread calls notify() on any object it releases lock of that object but may not immediately.* ***->Valid***

***Public final void wait() 🡪 Throws Interrupted Exception***

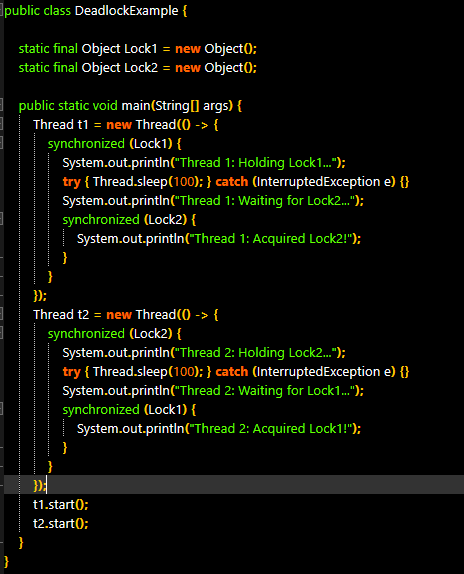
***Public final native void wait(long ms) 🡪 Throws Interrupted Exception***

***Public final void wait(long ms,int ns) 🡪 Throws Interrupted Exception***

***Public final native void notify()***

***Public final native void notifyAll()***

***Deadlock:-***

* *If two threads waiting for each other forever, such type of infinte waiting is called Deadlock.*
* *Synchronized keyword is only reason for deadlock situation, while using synchronized we need to take special care.*
* *There are no resolution technique but several prevention techniques are available.* 
* **Thread 1** locks Lock1 and tries to get Lock2
* **Thread 2** locks Lock2 and tries to get Lock1
* Neither thread can proceed = **deadlock**

***DeadLock vs Starvation:-***

***Deadlock****:- Long waiting of a thread waiting never ends is called Deadlock.*

***Starvation****:-Whereas long waiting of thread where waiting ends at certain point is called Starvation.*

***Eg:-*** *Low priority threads need to wait until completing all high priority threads. It maybe long waiting but ends at certain point , which is eg of Starvation*

***Daemon Threads:-***

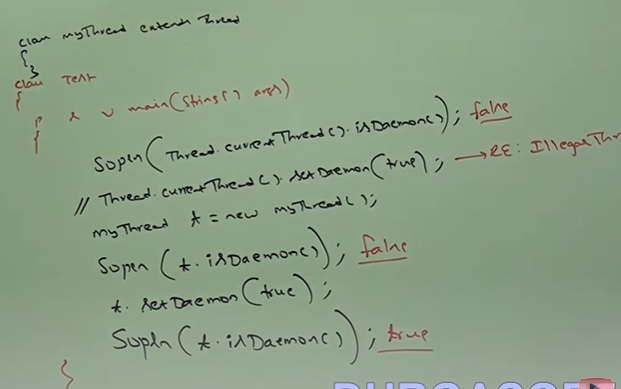
* *The threads which are executing in background are called Daemon threads.*
* *Threads that run* ***behind the scenes*** *to support other threads. They're* ***automatically killed*** *when all* ***non-daemon (user)*** *threads finish*
* *Eg:- Garbage Collector, Finalizer(Responsible for calling finalize()).*
* *The main objective of Daemon threads is to provide support for non-Daemon threads(Main thread). For eg:- If main thraed runs low on memory then JVM runs GC to destroy useless objects. So that no of bytes of free memory will be increases. 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 as well.(Priority allocation Controlled by JVM).*
* *We can check Daemon natur of thread by using below method Thread class*
  + ***Public boolean isDaemon();***
* *We can change Daemon nature of thread by using,*
  + ***Public void setDaemon(boolean b)***

*But changing Daemon nature is possible before starting of a thread only.* ***After starting a thread if we are trying to change Daemon nature, then we will get RuntimeException->IllegalThreadStateException.***

* *What is default nature of thread?*

*Main thread is always non-Daemon and for all remaining threads Daemon nature will be inherited from parent to child. i.e If parent thread is Daemon then automatically Child Thread is Daemon. If parent thread is non-Daemon then automatically child thread is also non-Daemon.*

* *It is impossible to change Daemon nature of main thread. Because it is already started by JVM at beginning. But we can change nature of child Daemon explicitly by using above setDaemon(true)*



* *Whenever last non-Daemon thread terminates automatically all Daemon threads will be terminated irrespective of their position.*