Mutithreading

Agenda:

1. Introduction
2. The ways to define thread  
    01. By extending Thread class

02. By implementing Runnable(I)

3. Setting and Getting the name of the thread

4. Thread Priorities

5. The methods to prevent method execution as below:-

01. yeilds()

02. join()

03. sleep()

6. Synchronization

7. Inter Thread Communication

8. Deadlock

9. Daemen Thread

10. Conclusion

1. Introduction

-> Executing several task simultaneously, is the concept of multi tasking. There are two types of multi tasking:-

01. Process based Multi tasking

02. Thread based Multi tasking

01. Process based Multi tasking

->Executing several task simultaneously, where each task is separate independent process, is called Process based Multi tasking.

Ex-

|  |
| --- |
| * While typing a java program in the editor, we can able to listen mp3 audio songs, at the same time we can download the file from internet. All these tasks are executing simultaneously and independent of each other and hence it is process based multi tasking. * It is best suitable at OS level but not at programmatic level. |

02. Thread based Multi tasking

->Executing several tasks simultaneously where each task is a separate independent part of the same program or process is called thread based multi tasking; where each independent part is called thread.

-> Thread is the independent part of the same process or program that is Thread is the sub-process which has the separate path of execution.

->It is best suitable at programmatic level.

Note:

|  |
| --- |
| * A flow of execution is called a Thread. |
| * In our program, if only one flow of execution is available that program is called Single Threaded Program. |
| * In our Program, if multiple flow of execution is available that program is called Multi Threaded Program. |
| * For every thread there is some independent job is available /defined. |
| * Whether it is process based or thread based, the main objective of multi tasking is to reduce the response time (execution time) and to improve performance of the system. |

* When compared with other language like C++, developing multi threaded application in java is very easy, because java provides in built support of multi threading with rich api(Thread, ThreadGroup, ThreadLocal, Runnable and etc..).
* Wherever independent jobs are there, that is the best place to use multi threading.

02. The ways of defining a thread

->We can define a thread in two ways:-

01. By Extending the Thread class

02. By implementing the Runnable(I)

01. By Extending the Thread class

Ex-

|  |
| --- |
| **package** com.sk.test2;  //Defining a thread  **public** **class** MyThread **extends** Thread {    **public** **void** run(){  //JOB of the thread  **for**(**int** i=0;i<10;i++){  System.***out***.println("Child Thread");  }  }  } |

|  |
| --- |
| **package** com.sk.test2;  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {    //Instantiation of the Thread  MyThread mt=**new** MyThread();  //Starting of the Thread  mt.start();    //JOB of Main Thread  **for**(**int** i=0;i<10;i++){  System.***out***.println("Main Thread");  }  }  } |

Case 1: Thread Schedular:

* It is the part of the JVM, if mutiple threads are waiting to get the chance for execution then in which order they will execute , is decided by Thread Schedular.
* What is Algorithm followed by Thread Schedular, we cant give any guarantee , it is varied from JVM to JVM , Hence in which order threads will be executed, we can’t give the guirantee and hence we can’t aspect exact output for multi threaded programs but we can provide severall possible outputs.
* The following are the various possible output for the above program:-

|  |
| --- |
| Main Thread  Main Thread  Main Thread  Main Thread  Main Thread  ………..  ………..  Child Thread  Child Thread  Child Thread  ………….  …………. |
| Child Thread  Child Thread  Child Thread  ………….  ………….  ………..  ………..  Main Thread  Child Thread  Main Thread  Child Thread  ………….  …………. |

Case 2:

**Diff between t.start() and t.run()**

* In t.start(), a new thread will be started
* But in the case of t.run(), no new thread will be created and run() method will be executed just like a normal method call by Main-Thread.
* In the above program, if replace t.start() with t.run() then output is :

Child Thread

Child Thread

Child Thread

………….

………(10 times)

Main Thread

Main Thread

Main Thread

………

………(10 times)

And total output is produced by only Main-Thread.

Case 3: Importance Thread Class start() method

* Thread class start() method is the responsible to do the following:-

1. Register our thread with ThreadScheduler
2. Other mandatory activities
3. Execute run method

Ex-

Start(){

1. Register our thread with ThreadScheduler
2. Other mandatory activities
3. Execute run method

}

|  |
| --- |
| Hence, without executing Thread class start() method, there is no chance of starting a thread in java.  Hence, Thread class start() method acts as Heart of Multi Threading. |

Case 4: Overloading of run() method

* Overloading of run() is possible but Thread class start() method always call no-arg run() method only.

Case 5: 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 output.
* It is highly recommended run() in our class otherwise don’t go for multi theading concept.

Case 6: If we are overriding start() method

* If we are overriding a start() method then our defined start() method will be executed just like normal method call and new thread/ child thread won’t be started.
* As we are not giving to chance to Thread class start() method , run() method won’t be executed.

|  |
| --- |
| **package** com.sk.test6;  //Defining a thread  **public** **class** MyThread **extends** Thread {    **public** **void** start(){  System.***out***.println("start() method");  }    **public** **void** run(){  //JOB of the thread  **for**(**int** i=0;i<10;i++){  System.***out***.println("Child Thread");  }  }  } |

|  |
| --- |
| **package** com.sk.test6;  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {    //Instantiation of the Thread  MyThread mt=**new** MyThread();  //Starting of the Thread  mt.start();    System.***out***.println("Main-Thread");    }  } |

o/p:-

|  |
| --- |
| start() method  Main-Thread |

Note: The total above output is produced by only Main-Thread.

Note:

* It is not recommended to override start() method in our class otherwise don’t go for multi threading.

Case 7:

Ex-

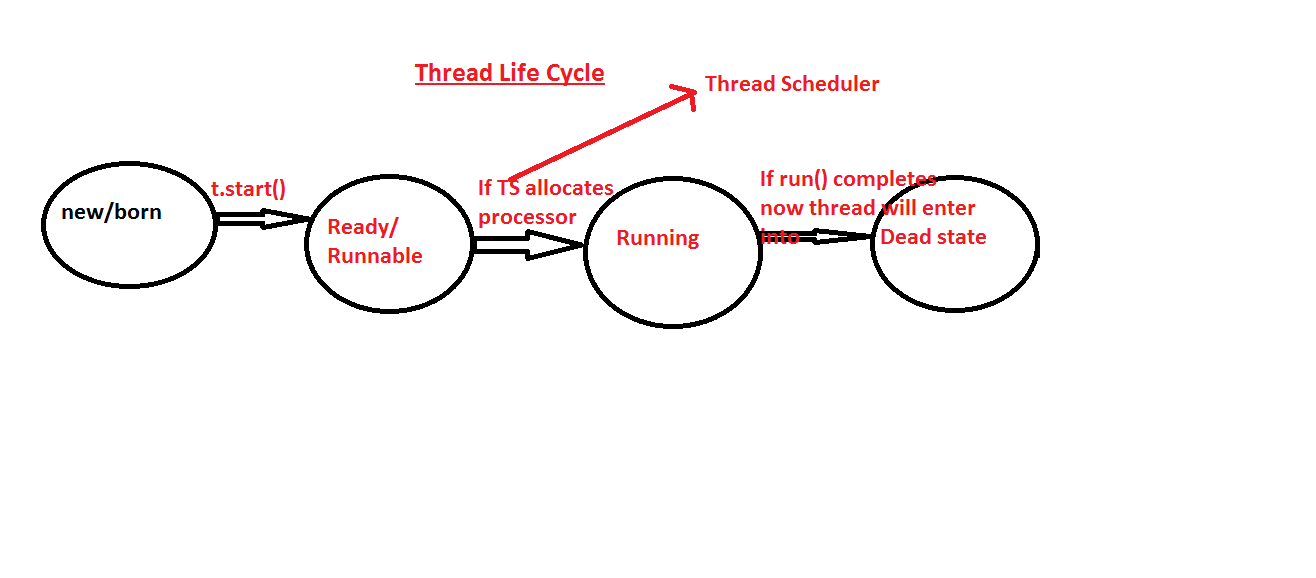
|  |
| --- |
| **package** com.sk.test7;  //Defining a thread  **public** **class** MyThread **extends** Thread {    **public** **void** start(){  **super**.start();  System.***out***.println("start method");  }    **public** **void** run(){  //JOB of the thread  System.***out***.println("run method");  }  } |

|  |
| --- |
| **package** com.sk.test7;  **public** **class** ThreadDemo {  **public** **static** **void** main(String[] args) {    //Instantiation of the Thread  MyThread mt=**new** MyThread();  //Starting of the Main-Thread  mt.start();    //JOB of Main Thread  System.***out***.println("Main Thread");  }  } |

o/p:-(Three possible output)

|  |  |  |
| --- | --- | --- |
| start method  Main Thread  run method | start method  run method  Main Thread | start method  run method  Main Thread |

Case 8:



Case 9:

* After starting a thread, if we are trying to restart same thread once again, then we will get RE: IllegalThreadStateException

Ex-

Thread t=new Thread();

t.start();

………..

………..

t.start();//RE:IllegalThreadStateException

-> We can also define a thread by implementing an inteface:

Pic:

* Runnable Interface is available in java.lang package and it cotains only one method whose prototype as: Public void run();

Case Study:

|  |
| --- |
| MyRunnable mr=new MyRunnable();  Thread t1=new Thread();  Thread t2=new Thread(); |

Case 1:

Q1. In which of the above cases a new thread will be created ?

T1.start() and T2.start();

Q2. In which of the above cases , a new thread started which is the responsible for the execution of MyRunnable class run() method ?

Ans. T2.start();

Q3. In which of the above cases , MyRunnable class run() method is executed ?

T2.start(), t2.run() and mr.run().

Q4. Which is the best way to define a Thread ?

Ans. Among two ways of defining a thread , implements Runnable approach(2nd approach) is recommended .

* In the First Approach, our class always extends the Thread class and hence there is no chance of extending any other class, due to this we are missing inheritence benefits.
* But in the 2nd approach while implementing Runnable Interface, we can extends any other class and hence we won’t miss any Inheritence benefits.
* Because of above reason, we can conclude implements Runnable approach(2nd approach) is recommended to use.

**Thread Class Constructors**

* **There are following are the Thread Constructor :**

|  |
| --- |
| 1. **Thread t=new Thread()** |
| |  | | --- | | 1. **Thread t=new Thread(Runnable r)** |  |  | | --- | | 1. **Thread t=new Thread(String name)** |  |  | | --- | | 1. **Thread t=new Thread(Runnable r,String name)** |  |  | | --- | | 1. **Thread t=new Thread(ThreadGroup ,String name)** |  |  | | --- | | 1. **Thread t=new Thread(ThreadGroup,Runnable r)** |  |  | | --- | | 1. **Thread t=new Thread(ThreadGroup g, Runnable r,String name)** |  |  | | --- | | 1. **Thread t=new Thread(ThreadGroup g, Runnable r,String name ,long stacksize)** | |

Durga’s approach to define instantiate and start thread (Not recommended to use):

Ex-

03. Getting and Setting a name of the Thread in Java:

-> Every Thread in java has some name, it may be explicitly provided by the programmer or default name generated by JVM.

-> Thread class defines the following methods to get and set name of a Thread:-

|  |
| --- |
| Public final String getName();  Public finale void setName(String name); |

Ex-

Program:

* Thread class defines static method to get current executing Thread reference.

|  |
| --- |
| Thread.currentThread().getName(); |

* Inside main(-) method current Thread means Main-Thread and its name is main.
* Inside run() method, current Thread means Child Thread and its name is “Thread-0”.

Program:

1. Thread Priorities

-> Every Thread in java has some priorities, it may be default priorities provided by jvm or customized priorities explicitly provided by programmer.

-> The valid range of the priority is 1 to 10 where 1 is min and 10 is max priority.

->Thread class defines the following constants to represent some standard priorities:-

Thread.MIN\_PRIORITY🡪1

Thread.MAX\_PRIORITY->10

Q1. Which of the following is the valid priority:-

0 1 10

* ThreadScheduler will use priority while allocating processor to thread.
* The thread which is having highest priority , will get chance first.
* If two threads having same priority then which thread will get the chance, we can’t imagine, it depends ThreadScheduler.
* Thread class defines the following methods to get and set priority of a thread :

Public final int getPriority();

Public final void set Priority(int p);

* The allowed range is 1-10, otherwise we will get RE: IllegalArgumentException

Ex-

|  |
| --- |
| Thread t=new Thread();  t.setPriority(10);//ok  t.setPriority(100);//RE: IllegalArgumentException |

Default Priority of Thread

* The default priority only for main-thread is 5 but remaining threads the defaults priority will be inherited from parent to child.
* That is whatever parent thread has priority, the same priority will be there for child thread also.

Pic scjp8.

|  |
| --- |
| scjp8.png |

|  |
| --- |
| **package** com.sk.test3;  **public** **class** MyThread **extends** Thread {    **public** **void** run(){  System.***out***.println("Child Thread");  }  } |

|  |
| --- |
| **package** com.sk.test3;  **public** **class** ThreadDemo **extends** Thread{    **public** **static** **void** main(String[] args) {    MyThread mt=**new** MyThread();  mt.setPriority(10);//---🡪1  mt.start();      }  } |

* If we comment line1 then then both main and child thread has priority 5.
* If we are not commenting line 1 then child thread priority has 10 and main thread has priority 5 hence child thread will get chance first followed by main-thread, In this case output is :

|  |
| --- |
| Child Thread  Main-Thread |

Note: Some plateform won’t provide proper support for Thread priority.

* The methods who prevents thread execution.
* We can prevent a thread execution using the following methods:-

1. Yield();
2. Join();
3. Sleep();
4. Yield()

* Yield() method cause to pause current executing thread to give the chance for remaining waiting threads of the same priority.
* If there are no waiting threads or all waiting threads have low priority then same thread will get the chance for execution.
* same priority then which waiting threads will get the chance , we can’t expect , it depends on ThreadScheduler.
* Q.The thread which is yielded, when it will get chance once again?

Ans. We can’t expect, it depends on mercy of ThreadScheduler.

* Prototype of yield():
* Public static native void yield();

Pic scjp10:

|  |
| --- |
| scjp10.png |

|  |
| --- |
| **package** com.sk.test3;  **public** **class** MyThread **extends** Thread {    **public** **void** run(){  **for**(**int** i=0;i<10;i++){  System.***out***.println("Child Thread");  Thread.*yield*();//-->1  }  }  } |
| **package** com.sk.test3;  **public** **class** ThreadDemo **extends** Thread{    **public** **static** **void** main(String[] args) {    MyThread mt=**new** MyThread();    mt.start();    **for**(**int** i=0;i<10;i++){  System.***out***.println("Main-Thread");  }      }  } |

* If we comment line1 then we can’t expect

Which thread can complete first.

* If we are commenting line1 then the chance of completing Main-Thread is high because child thread is always calling yields() method.

Note:

* Some OS may not provide proper support for yield() method.

1. Join()

* If a thread wants to wait until completing some other thread, then we should go for join() method.
* Ex-  
  Thread t1 wants to wait until completing execution of Thread t2 then t1 has to call t2.join() then t1 will enter into waiting state.
* Once t2 completes their execution then immediately t1 can continue its execution.

Pic scjp11.png:

|  |
| --- |
| scjp11.png |

* Complete join() signature:

1. Public final void join() throws InterruptedException
2. Public final void join(long ms) throws InterruptedException
3. Public final void join(long ms,int ns) throws InterruptedException

Pic scjp12.

* Waiting of main-thread until completing child-thread.

Ex-

Pgm join():

* Main-Thread has to wait until Child-Thread execution completion.

Output:

|  |
| --- |
| Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Seetha Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread  Rama-Thread |

* Wating of child-thread until Main-Thread compeletion.

Case 3:

* If Main-Thread call join() method on Child-Thread and Child-Thread calls join() on Main-Thread() then both Threads will wait for each other forever.
* In this case, the program will be hanged(this is something like Deadlock).

Ex-

Case 4:

* If a thread call a join() method on the sam thread itself then the program will be hanged(this is something like Deadlock).

Ex-

Note:

* Wherever we are using join() method , compulsory we should handle InterruptedException because every join() method throws InterruptedException which is checked Exception otherwise we will get Compile time Error.

1. sleep(-)

-> If a thread don’t want to perform some operations for particular amount of time then we should go for sleep() method.

-> Prototype:

|  |
| --- |
| public static native void sleep(long ms) throws IntrruptedException  public static void sleep(long ms,int ns) throws IntrruptedException |

Q. How a thread interrupted to another thread?

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

|  |
| --- |
| public void interrupt(); |

Ex-

* In the above example, Main-Thread interrupts Child-Thread.

Note:

* Whenever we call interrupt() method, If the target thread is not in sleeping or waiting state then there is no impact of interrupt() method call immediately.
* Interrupt() method call will wait until enter into sleeping or waiting state.
* If the target thread entered into waiting or sleeping state then interrupt() call will interrupt to that thread.
* If target thread never entered sleeping or waiting state in its life time then, then only interrupt() call will be wasted.
* In the above program interrupt() call waited until child thread completing for loop 1000 times.

Saturday, March 21, 2015

**Comparison between yield() ,join() and sleep()**:-

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Yield() | Join() | Sleep() |
| Purpose | It causes to pause current executing thread to give the chance for remaining waiting of same priority | If a thread wants to wait until completing some other thread then we should go for join | If a thread don’t want to perform any operation for apartincaular amount of time ie just pausing is required |
| Is it static method? | Yes | No | Yes |
| Is It final method? | No | Yes | No |
| Is it overloaded or not? | No | Yes | Yes |
| Is it throws IE? | No | Yes | Yes |
| Is it native method? | Yes | No | Sleep(long ms)->native  Sleep(long,int ns)->non-native |

Daemon Thread

->The threads which are executing in the background are called Daemon thread.

-> Garbage collector

-> The main objective of Daemon thread is to provide support for non-Daemon threads (like main-thread).

->For Example, if main-thread runs with low memory then jvm runs Daemon-Thread Gc then GC destroys useless objects and provide some free memory so that main-thread can continue its execution without any problems.

Real Time Example:-

* It is just like Producer, Director , Camera man and other members while Hero, heroins are acting in a movie.
* Usually Daemon runs with low priority but based on our requirement they can with high priority also.
* We can check whether thread is Daemon or not by using isDaemon() method of Thread class.
* Public boolean isDaemon();
* We can change daemon nature of a thread by using setDaemon(-) of Thread class.
* Public void setDaemon(boolean b);
* But we can change the daemon nature of a thread before starting of a thread only that is once thread is started, we can’t change its daemon nature otherwise we will get RE: IllegalThreadStateException.

Default Nature of Daemon Thread

->By default main-thread is non-daemon but all remaining daemon nature is inheriting from parent to child that is if the parent thread is daemon then child thread is also daemon and if parent thread is not daemon then child thread is also not daemon.

Ex1-

-> Whenever last non-daemon threads terminate the automatically all daemon threads also will be terminate with respect to their position.

Ex2-

Note:-

* It is impossible to change daemon nature of main-thread because it is already started by jvm at beginning only.

**Synchronization**

* Synchronization is the keywords which is only applicable for method and blocks but not for classes and variables.
* If multiple threads are operating simultaneously on the same java object then there may be a chance data inconsistence, this is called RACE condition.
* We can solve this problem using synchronized keyword.
* If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object.
* As threads are executing one by one , data inconsistency problem will be solved automatically.
* The main disadvantage of synchronized keyword is increases waiting time of the threads and creates performance problems; hence if there is no specific requirement then it is not recommended to use synchronized keyword.

Real Time Example:

1. Dogs
2. Sweets
3. Bank

* Internally synchronization concept is implemented using lock concept.
* Whenever we are using synchronized keyword then only lock concept comes into the picture.
* If a thread wants to execute synchronized method on the given object; first it has to get lock of the object. Once thread got a lock then it is allowed to execute any synchronized method on the given object.
* Once synchronized method execution completes, automatically thread releases the lock.
* Acquiring and releasing the lock takes care by JVM and programmer not required to do anything.
* 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 method simultaneously.

|  |
| --- |
| Class X{  synch m1()  Synch m2()  M3()  } |

* T1 wants to execute m1()
* T2 came for m1 --🡪X
* T3 cam for m3 ---🡪Ok

Note:

Pic scjp13:

* Every object in java has a unique lock.
* Lock concept is defined based on object but not based on method.
* Inside a method, if we are performing any update operation and the state of application (or) object is changing then that method should be declared as synchronized.
* Inside a method, jus we are performing read operation and application/object won’t be change then that method should be declared non-synchronized.

Ex1-

Banking Application

* In the Banking application, withdraw operation changing state of application and should be performed by only one thread at a time on the given the account object. Hence it should be synchronized.
* balance\_Enquiry() won’t perform any update operations and can be executed by multiple threads simultaneously ; hence it should be non-synchronized.

Ex2:-

Reservation System

* check\_ Availability() of ticket won’t perform any update operation and it is just read operation and hence this method should be non-synchronized.
* book\_Tickets() method performs update operations and hence this method should be perform by only one thread at a time; hence it should be synchronized.

Ex-

Program (sync2):

* if wish method is not synchronized then both threads will be executed simultaneously and hence we will get irregular output.

Output:

|  |
| --- |
| Good Moring: Good Morning  Dhoni  Good Morning |

* if we declare wish method as synchronized then only one thread is allowed to execute that method on the given object ; hence we will get regular output.
* Output:

|  |
| --- |
| Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Raina  Good Morning:Raina  Good Morning:Raina |

Case Study:

|  |
| --- |
| Display d1=new Display();  Display d2=new Display();  MyThread mt1=new MyThread(d1,”Dhoni”);  MyThread mt2=new MyThread(d1,”Raina”);  mt1.start(); mt2.start(); |

pic scjpsync1:

* Even though wish method is synchronized then also we will get irregular output because both thread will be executed on different Display objects simultaneously.

Note:

* If multiple threads are operating on multiple java objects then synchronization is not required.
* If multiple threads are operating on same java object then synchronization is required.

Class level Lock

* Every class in java has unique lock which is also class level lock.
* If a thread wants to execute static synchronized method ; first it has to get class level lock; if Thread got class level then it is allowed to execute any static synchronized method.
* Once method execution completes automatically , thread releases the lock.

Q-> why a thread is executing static synchronized method then the remaining threads are not allowed to execute any static method of that class simultaneously;

But remaining threads are allowed to execute simultaneously:-

1. nonsynchronized static method
2. Synchronized instance method
3. Nonsynchronized instance method

Ex-

|  |
| --- |
| Class x{  Static synch m1()  Static synch m2()  Static m3()  Synch m4()  m5()  } |

Pic scjpsync2:

Synchronized block:

->If very few lines of the code require synchronization then it is not recommended to declare entire method as synchronized.

-> We have to declare those few lines of the code by using synchronized block.

-> The main advantage of synchronized block over synchronized method is performance will be improved by reducing waiting time of threads.

-> We can synchronized block as follows:

a) To get lock of current object:-

Ex-

Synchronized(this){

…………..

…………..

}

* If a thread got lock of current object then only it is allowed to execute this area.

b) To get lock of a particular object

Ex-

Synchronized(b){

…………

………….

}

* If a thread got lock of ‘b’ object then only it is allowed to execute this area.

c) To get class level lock

Ex-

Synchronized(Display.class){

}

* If a thread got class level lock then only it is allowed to execute its block.

Note:

* Lock concept available only for object and classes but not for primitive hence we can’t pass primitive type as argument to synchronized block otherwise we will get compile time error saying unexpected type

Found: int required : required

Ex-

Int x=10;

Synchronized(x){…………….}

FAQ1. What is the synchronized keyword and where we can apply?

FAQ2: Explain advantage of synchronized keyword?

FAQ3: Explain disadvantage of synchronized keyword?

FAQ4: What is RACE condition?

FAQ5: What is Object lock and when it is required?

FAQ6: What is class level and when it is required?

FAQ7: What is difference between class level lock and object level lock?

FAQ8: Why a the thead executing synchronized method and any onject is the remaining thread are allowed to execute ?;;;;;;;;corection

Ans: No

FAQ10: What is synchronized block ? Explain advantage block over synchronized method?

11. Explain the difference between synchronized block and synchronized method?

12. Is a thread required multiple locks simultaneously ?

Ans: yes from different object.

Ex-

Class y{  
 public synch void m1(){

X x=new X();

Synchronized(x){

………………

………………

}

}}

Y y=new Y();

y.m1();

* To reach this area thread required two locks of X and Y.

14. What is synchronized statement?

Ans. Interview people created terminology

* The statements which are present in synchronized block or synchronized method; called synchronized statement.

Inter Thread Communication

* Two threads can communicate with each other by using the following methods:

1. wait()
2. notify()
3. notifyAll()

* The thread which is expecting updation ; is responsible to call wait() method then immediately thread will enter into waiting() state.
* The thread which is performing updation, after performing the updation it is the responsible to call notify() method; then waiting thread will get the notification and continue its execution with those updation.
* wait(), notify() and notifyAll() present in Object class because thread can these methods on any java object.
* If a thread wants to call wait(), notify() and notifyAll(); it should be owner of that object that is thread should have lock of that object that is thread should be inside synchronized area. Hence, we can call wait(), notify() , notifyAll() methods only from synchronized area; otherwise we will get RE:IllegalMonitorStateException.
* If a thread call a wait() on any object, it immediately releases the lock of that particular object and entered into waiting state.
* If a thread calls notify() method on any object , it releases lock of that particular object but may not immediately.
* Except wait(), notify(), notifyAll(); no other methods where thread releases lock.

|  |  |
| --- | --- |
| Methods | Is thread releases the lock? |
| Wait() | Yes |
| Notify() | Yes |
| notifyAll() | Yes |
| Yield() | No |
| Sleep() | No |
| Join() | No |

* Which are the following is valid:-

1. If a thread calls wait() on any object it immediately release all lock acquired by that thread and entered into waiting state.
2. If a thread call wait() on any object , it releases the lock of that object, may not immediately.
3. If a thread call wait() on any object, it immediately entered into waiting state without releasing the any locks.
4. If a thread calls wait() on any object, it immediately releases the lock of that particular object and entered into waiting state.
5. If a thread calls notify() on any object, it immediately releases the lock of that particular object.
6. If a thread calls notify() method on any object, it releases all locks acquired by that thread immediately.
7. If a thread calls notify() method on any object, it releases lock of that particular but not immediately.

|  |
| --- |
| Public final void wait() throws IE |
| Public final native void wait(long ms) t IE |
| Public final void wait(long ms,int ns) IE |
| Public native final void notify() |
| Public native final void notifyAll() |

Ex-(Pic incom3):

|  |
| --- |
| scjpintercomm3.png |

* Ex-Program:

|  |
| --- |
| **package** com.sk.test3;  **public** **class** MyThread **extends** Thread {    **int** tot=0;    **public** **void** run(){    **synchronized**(**this**){    System.***out***.println("Child thread starts calculation");  **for**(**int** i=1;i<=100;i++){    tot+=i;  }    System.***out***.println("Child thread giving notification");  **this**.notify();  }  }  } |
| **package** com.sk.test3;  **public** **class** InterThreadComTest {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  // **TODO** Auto-generated method stub    MyThread mt=**new** MyThread();    mt.start();    **synchronized**(mt){  System.***out***.println("Main thread calling wait method");    mt.wait();    System.***out***.println("Main thread got notification");    System.***out***.println("Total result:"+mt.tot);  }  }  }  /\*  output:  Main thread calling wait method  Child thread starts calculation  Child thread giving notification  Main thread got notification  Total result:5050  \*/ |

Producer – Consumer Problem

Ex-Fig:-

* Producer thread is responsible to the Queue.
* Consumer thread is responsible to consume items from the Queue.
* If Queue is empty then consumer thread is responsible to call wait() and immediately Consumer thread will enter into waiting state.
* After producing items to the Queue; Producer thread is responsible to call notify() method then the waiting Consumer thread will get that notification and continue its execution with those updated items.

Q. what is difference between notify() and notifyAll()?

Ans:-

* We can use notify() method to give the notification for only one waiting thread.
* If multiple threads are waiting then only one thread will be notified and the remaining threads have to wait for further notifications.
* But which thread will be notified; we can’t imagine. It depends upon jvm(ThreadSchedular).
* We can use notifyAll() to give notification to all waiting threads of a particular object then all waiting threads which are waiting for that particular object; will be notified but execution should be one by one(because threads require lock and only one lock is available for that object).

=🡺 on which object wait has call that object lock should be acquire by the thread.

-> If a thread want to call wait() on s1 then thread has to get lock of s1 but not lock of s2.

Deadlock

* If two threads are waiting for each other forever, such type of infinite waiting is called Deadlock.
* Synchronized keyword is the only reason of deadlock, hence while using synchronized; we have to take special care.
* For Deadlock, there are no resolution technique but several prevention techniques are available.

Deadlock Vs Starvation

-> A long waiting of a thread where waiting thread never ends, is called Deadlock.

-> A long waiting of a thread where waiting thread ends at a certain point is called Starvation.

Ex-

* Low priority thread has to wait until completing all high priority threads, it may be long waiting but ends at a certain point and hence it is called Starvation.

Green Thread

->Java multithread is concept is implemented by using models:

01. Green Thread Model

02. Native OS Model

01. Green

-> The threads which are managed completely managed by JVM without taking underlying OS support are called Green Threads.

02. Native OS Model

-> The threads which are managed by the JVM with help of underlying OS support are called Native OS threads.

Note:

* Very few OS like Sun Solaris provides support for Green Thread Model; anyway Green Thread Model is deprecated, not recommended to use.
* All windows OS support only native OS Model.

How to Stop a Thread In The Middle of Execution

* We can stop a thread execution by using stop() method of Thread class then Thread will enter into Dead state.
* Public void stop(); anyway stop() method is deprecated and not recommended to use.

Suspending and Resuming of a Thread

* A thread can suspend other thread by using suspend() of Thread classmethod then immediately thread will enter into suspended state.
* A Thread can resume to a suspended thread by using resume() method of Thread class then suspended thread can continue its execution.

|  |
| --- |
| public void suspend();  public void resume(); |

* Anyway suspend() and resume() are deprecated and not recommended to use.

Saturday, 28 March, 2015

ThreadGroup

* Based on the functionality we can group threads into a single unit which is nothing but threadgroup represents a set of threads.
* In addition to threads ThreadGroup cal also contain SubThreadGroup.
* The main advantage of maintaining thread int the form of ThreadGroup is we can perfrom common operations very easily.

Ex-

Give MAX\_PRIORITY for all producer thread.

Start all consumer thread etc.

* ThreadGroup is a java class present in java.lang package and it is the direct child class of Object class.
* Every Thread in java belongs to some group; main-thread belongs to main-thread-group.
* Every ThreadGroup is the child group of the SystemGroup either directly or indirectly and hence SysteGroup access not for all ThreadGroup in java.
* SystemGroup contains several System level theads like reference handler,finalizer,attacheListener, Signal Dispatcher and ect.
* Fig1.
* Ex-

|  |
| --- |
| Class ThreadGroupDemo1{  P s v m(String[] args){  Sopln(Thread.currentThread().getName());  Sopln(Thread.currentThread().getThreadGroup().getName());  Sopln(Thread.currentGroup().getThreadGrooup().getParent().getName());  }} |

* Main() method is executed by main-thread and main-thread belongs to main-Thread-group.

Constructor

1. ThreadGroup g=new ThreadGroup(String gname);

* Creates a new ThreadGroup with specified group name.
* The parent thread of the new group is the thred-group of currently executed thread.

1. ThreadGroup g=new ThreadGroup(ThreadGroup pg,String name);

* Creates a new ThreadGroup with specified group name; parent thread of this thread-group is specified threadgroup.

Ex-

|  |
| --- |
| Class ThreadGroupDemo{ p s v m(String args[]){ Sopln(Thread.currentThread().getThreadGroup().getName());//main  Sopln(Thread.currentThread().getThreadGroup().getParent().getName());//System  ThreadGroup pg=new ThreadGroup(“ParentGroup”);  Sopln(pg.getParent().getName());//main  ThreadGroup cg=new ThreadGroup(pg,”ChildGroup”);  Sopln(cg.getParent().getName());//parent thread  }} |

Java.util.cuncurrent package

* The problems with synchronized keywords are:-

1. If a thread releases the lock then which waiting thread will get the lock, we are not having control on this.
2. We can’t specify maximum waiting for a thread to get a lock so that it will wait until getting lock which may create performance problems and which may cause deadlock.
3. We are not having any flexibility to try for a lock without waiting.
4. There is no API to list out all waiting thread for a lock.
5. The synchronized keyword, we have to use only within a method and not possible to declare across multiple methods.

* To overcome from above problem SUN people introduced java.util.concurrent.locks package in jdk1.5v and it also provides several enhancement to the programmer to more control on concurrency.

Lock(I)

* Lock object is similar to implicit thread acquired by a thread to execute synchronized method or synchronized block.
* Lock implementation provides more flexibility and extensive operations to the programmer then traditional implicit lock.
* Important methods of Lock(I):-

1. Void lock()

* To get the lock.
* If the lock is available immediately thread will get that lock.
* If the lock is not available then thread will wait until getting that lock; this behavior is exactly implicit lock.

1. Boolean tryLock()

* To acquire the lock it is available then the thread acquires lock and return true.
* If the lock is not available then this method return false and continue its execution without waiting; in this case thread never be blocked.

|  |
| --- |
| If(l.tryLock()){ perform some safe operations}  Else{perform some alternative operations} |

1. Boolean tryLock(long time, TimeUnit unit)

* To acquire the lock if is available then immediately that thread will get that lock and if the lock is not available then the thread will wait until specified amount of time ,still if the thread’s lock is not available then the thread continue its execution.

Ex-

If(l.tryLock(1000,TimeUnit.MILLISECONDS)){}else{}

TimeUnit

* It is a enum presenti java.util.concucurrent package

Ex-

Enum TimeUnit{

NANOSECONDS;

MICROSECONDS;

MILLISECONDS;

SECONDS;

MINUTES

SECONDS

1. Void interruptly()

* While locks interruptly acquire the available; if the lock is not available then It will wait.
* While waiting if it is interrupted then the thread won’t the lock.

1. Void Unlock()

* To unlock the locks
* The call the unlock() method compulsory current thread should be owner of that thread otherwise we will get compiler time error saying IllegalMonitorException.

ReentrantLock(C)

* It implements Lock(I).
* It is the direct child class of the object class.
* Reentrant means a thread can acquire same lock multiple times without any issues.
* Internally ReentrantLock increment threads personal; whenever we call unlock and lock will be release whenever count reaches zero.

Constructor

1. ReentrantLock rl=new ReentrantLock()

* Creates a new lock.

1. ReentrantLock rl=new ReentrantLock(boolean fairness)

* Creates a new lock with specified fairness policy.
* If fairness is true then longest waiting thread can get the lock if it is available that is it follows FCFS(First comes first server).
* If fairness is false then we can’t expect which waiting .
* If we are not setting then default value is false.

Q. which of the following are eqals:

01. ReentrantLock rl=new ReentrantLock();

02. ReentrantLock rl=new ReentrantLock(true);

03. ReentrantLock rl=new ReentrantLock(false) ;

Method of ReentrantrantLock

1. Void lock()
2. Boolean tryLock()
3. Boolean tryLock(long l,TimeUnit t)
4. Void lockInterruptly()
5. Void unlock()

Other Methods are:-

1. Int getHoldCount()

* Returns the no. of holds on this lock by current .

1. Boolean isHeldByCurrentThread()

* Return true if only if if lock is held by current thread.

1. Int getQueueLength()

* Return the no. of the lock waiting for the lock.

1. Collection of thread which are waiting to get the lock
2. Boolean hasqueuedThread()
3. Boolean isLocked()

* If the lock is acquired by some thread.

1. Boolean isFair()

* Returns true if the lock object is fair.

1. Thread getOwner()

* Returns the thread which acquire the lock.

Sunday, 29 March, 2015

* Ex-

|  |
| --- |
| Import java.util.concurrent.locks.\*;  Public class ReentrantLock2{  Public static void main(String[] args){  ReentrantLock I=new ReentrantLock();  l.lock(); l.lock();  Sopln(l.isLocked());//true  Sopln(l.isHeldByCurrentThread());//true  Sopln(l.getQueueLength())//0  l.unlock();  Sopln(l.getHoldCount());//1  Sopln(l.isLocked());//true  l.unlock();  Sopln(l.isLocked());//false  Sopln(l.isFair())//false  }} |

Ex-Program:

|  |
| --- |
| **package** com.sk.reentrant1;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** Display {  ReentrantLock l=**new** ReentrantLock();  **public** **void** wish(String name){  l.lock();  **for**(**int** i=0;i<10;i++){  System.***out***.print("Good morning");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(name);  }  }  }    **package** com.sk.reentrant1;  **public** **class** MThread **extends** Thread {  Display d;  String name;  MThread (Display d,String name){  **this**.d=d;  **this**.name=name;  }  **public** **void** run(){  d.wish(name);  }  }  **package** com.sk.reentrant1;  **public** **class** ReentrantLockDemo {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  Display d=**new** Display();  MThread t1=**new** MThread(d,"Dhoni");  MThread t2=**new** MThread(d,"Yuvraj");  t1.start();  t2.start();  }  } |

* If you comment both line 1 and 2 then all thread will execute wish method simultaneously and we will get irregular output.
* If we are not commenting lines 1 and 2 then threads will be executed one by one and hence we will get regular output.

Ex-Prog For tryLock()

|  |
| --- |
| **package** com.sk.reentrant2;  **import** java.util.concurrent.locks.ReentrantLock;  **import** com.sk.reentrant1.Display;  **public** **class** MThread **extends** Thread {  **static** ReentrantLock *l*=**new** ReentrantLock();  MThread (String name){  **super**(name);  }  **public** **void** run(){  **if**(*l*.tryLock()){  System.***out***.println(Thread.*currentThread*().getName()+"got the lock and performing the safe oprations.");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  *l*.unlock();  }  **else**{  System.***out***.println(Thread.*currentThread*().getName()+" unable to get the lock and perfroming some alternative operations");  }  }  }  **package** com.sk.reentrant2;  **public** **class** ReentrantLockDemo {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  MThread t1=**new** MThread("Dhoni");  MThread t2=**new** MThread("Yuvraj");  t1.start();  t2.start();  }  }    //output  Yuvrajgot the lock and performing the safe oprations.  Dhoni unable to get the lock and perfroming some alternative operations |

* Ex-Program for tryLock with time period

|  |
| --- |
| package com.sk.reentran3;  import java.util.concurrent.TimeUnit;  import java.util.concurrent.locks.ReentrantLock;  import com.sk.reentrant1.Display;  public class MThread extends Thread {  static ReentrantLock l=new ReentrantLock();  MThread (String name){  super(name);  }  public void run(){  do{  try{  if(l.tryLock(5000,TimeUnit.MICROSECONDS)){ System.out.println(Thread.currentThread().getName()+"got the lock and performing the safe oprations.");  Thread.sleep(3000);  l.unlock();  break;  }  else{  System.out.println(Thread.currentThread().getName()+" unable to get the lock and perfroming some alternative operations");  }  }catch(Exception ex){}  }while(true);}} |
| **package** com.sk.reentran3;  **public** **class** ReentrantLockDemo {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  MThread t1=**new** MThread("Dhoni");  MThread t2=**new** MThread("Yuvraj");  t1.start();  t2.start();  }  //output  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhoni unable to get the lock and perfroming some alternative operations  Dhonigot the lock and performing the safe oprations. |

FAQ01. When we will get ConcurrentModificationException?

1. What is the need of ConcurrentCollection?
2. What is diff between HashMap and ConcurrentHashMap?
3. What is diff Hashtable and ConcurrentHashMap?
4. What is diff Fail-Safe Iterators and Fail-Fast Iterators?
5. What is the diff between AL and CopyOnWriteAL?

ThreadLocal

* ThreadLocal class provides ThreadLocal variables.
* ThreadLocal class maintaince values for Thread basis.
* Each thread local object maintains a separate values (like userId , transactionId and etc) for each thread that accesses that object.
* Thread can access its local value, can manipulates its value and can remove also.
* Every part of the code which is by the thread we can access its local variable.

Ex-

Consider a servlet which invoke some business methods; we have a requirement to generate a unique Transaction Id for each and every request and we have to pass this transaction id to the business method for logging purpose; for this requirement we can use ThreadLocal to maintain a separate Transaction Id for every request that is for every thread.

Note:

* ThreadLocal class introduced in Jdk1.2v.
* ThreadLocal can be associated with ThreadScope.
* All the code which is executed by the Thread has access to corresponding ThreadLocal variable.
* A thread can access its own local variable and can’t other thread local variables.
* Once Thread enters into Dead state, all its local variable or by default eligible for garbage collection.

Constructor

ThreadLocal tl=new ThreadLocal();

Methods:

1. Object get()

* Returns the value associated with current thread in the ThreadLocal object.

1. Object initialValue()

* Returns initial value of the ThreadLocal variable associated with current thread.
* The default implementation of this method is null.
* To customize our own initial value; we have to override this method.

1. Void set(Object newValue)

* To set a new value.

1. Void remove()

* To remove current thread local variable value.
* It is the newly added method in Jdk1.5v.
* After removal, if we are trying to access; it will be re-initialized once again by invoking its initialValue() method.

Overriding on initialValue()

* Pgm2:
* In the above program for every customerThread a separate customer Id will be maintained by ThreadLocal Object.

ThreadLocal vs Inheritance

|  |  |
| --- | --- |
| 01 ThreadLocal bydefault not available to the child thread. | 1. If you want to make parent thread local value available to the child thread then we should go for InheritableThreadLocal. |
| Bydefult child the the value is exactly same as parent thread value but we can provide customize value to the child thread by overriding chidValue() method. |  |
|  |  |
|  |  |
|  |  |

* Constructor
* InheritableThreadLocal tl=new InheriatableThreadLocal();
* InheritableThreadLocal is the child class of ThreadLocal and hence all the methods present in the thread local by default available to InheritableThreadLocal.
* In addition to these method if contains only one new method as below:

Public Object childValue(Object parentValue);

Ex-

Prg3:

* In the above if we replace InheritableThreadLocal with ThreadLocal and if we are not overriding child value method then the output is:

parentThead value🡪pp

childthread value->null

* If in the above if we are maintaining inheriatableThreadLocal and we are not overriding childValue() method then

Output:

parentThread value is: pp

childThread value: pp