Multi-Tasking & Threading:

Threading is used to achieve a multitasking in a Java program, it means one program of Java can perform more than one task simultaneously.

In this generation every software or Apps are multi-threaded.

Real world example of multi-tasking: Human body is the best example of multi-tasking. Another example is washing machine, at the same time it washes and dry the clothes.

Need of Threading:

Before Multi-tasking, There were single tasking OS that was only able to run one application at a time. DOS is an example of single tasking application. Requirement came to run multiple program at the same time and threading concept came in the picture to fulfil this requirement.

Example of similar requirement: As we often use MS word for making documents/Assignment etc. We may noticed, spell checking program runs in parallel that checks mistakes while we are typing and it also notifies us to correct the mistake. Even at the same time it creates a list of suggestion to fix our typing errors. This is the requirement where it demands to run more than one program in parallel.

How OS does executes more than one tasks simultaneously?

Actually no work executes at the same time. There are always difference of Nano/Pico seconds. It works so fast that it looks like all tasks are running at the same time. But as human we are only able to see differences of seconds so it looks all works are happening at the same time but actually it’s not.

Problem and How Multi-tasking achieved: To do a job, we requests to OS and OS sends the command to processor. Now OS is free and able to take another job request from us. But processor is still busy and unable to take another command from OS.

Solution: Every time when processor do his job, it does in set of instructions. In some point processor waits means pause for a while and here OS gives another job to processor.

Every application has some instructions that makes processor idol for a while. For example Input-Output.

It means, processor jumps from one task's set of instructions to another task's set of instructions every time it pauses. But always remember only one task executes at a time.

Definition:

Running state of instructions is known as "**Process**" and static state of instructions is known as "**Program**".

There are two types of process based Multi-tasking.

- Heavy weight process multi-tasking, shuffles between applications.
- Light weight process multi-tasking

Those instructions which are taking a separate memory area on a separate address space in a RAM to run their self are forming a "**Heavy-weight Process**".

Those instructions which are not taking a separate memory area or an address space in a RAM to run their-self are forming a "**light weight process**".

OS always has a Heavy-weight process. All Applications like MS word, excel, PowerPoint takes separate space. That's why we say if we want to improve the performance then increase the RAM.

Need Of Light-Weight process also called thread.

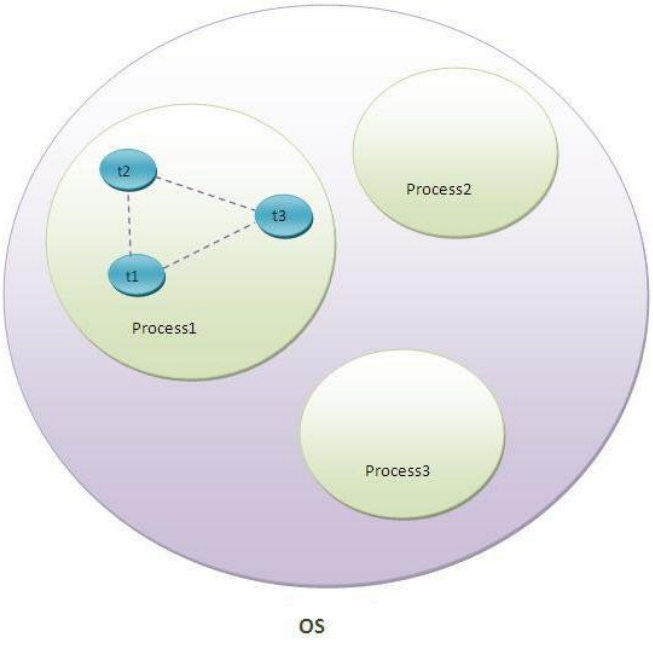
As we discussed above, multitasking achieved through context-switching/shuffling between tasks. It means processor jumps from one space area to another space area that dropped the application performance. If we again take example of MS word typing and Spell checker then processor is jumping from MS word space area to spell checker space area time to time.

In case of light weight process, processor also shuffles between MS typing and Spell checker program but this time spell checker program is inside space area of MS word that obvious decrease the travel time and improve the performance. In this case we are not giving separate memory area to spell checker.

Real World example: Manish sir has given example of his life when he was coming to ducat from merath and that was taking too much time. After that he moved to Noida and travel time decreased, performance increased.

If we take example of java then we know that JVM is a Heavy-weight process because it takes separate memory in a RAM. JVM runs about 32 light weight processes (threads) at a time. One of them is _____ thread who converts our Java program into byte codes and saves in hard disk.

Heavy-weight processing and lightweight processing both are used to achieve multitasking. But we often used lightweight process because threads share a common memory area so they don't allocate separate memory area. It saves memory and context switching between the Threads and take less time than heavy weight process.



As shown in the figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Light-Weight process called thread why?

As we know thread means in Hindi is “धागा”. Thread is like a puppet game. I am not able explain it.

Threading In Java.

#. Threads are also represents in the form of object. “**Thread**” class is inside “Lang” package that is implicitly imports in our Java program.

#. One object of a thread class is responsible to create only one thread in Java. Without creating a “**Thread**” class object we cannot start thread in Java. Even JVM makes object of Main Thread implicitly.

#. Every thread we create is a child of main thread. Every thread runs separately, that’s why if somehow exception comes or main thread ends, the child thread won’t dead.

Advantages of Threading.

1. It does not block the user because Threads are independent and can perform multi operation at the same time.
2. We can perform many operations together so it saves time.
3. Threads are independent so it does not affect other Threads if exception occurs in a single thread.

Two ways to create a Thread.

1 By extending Thread Class. Also called “Via Inheritance” or “Is a Relationship”.

2. By implementing Runnable/Callable interface. Also called “Via Association” or “Has a Relationship”.

Something Important: As a programmer we are not able to know when processor is going to be idol and our thread will only get chance to run when processor is free. As we know JVM connects with the OS it means Java technology knows when processor is free. So we will provide our tasks to java technology and Java technology will execute our job when processor is free.

“**Kaam kisi aur ka karna kisi aur ne hai**”.

“**Kaam Mera Aur Karna Java Technology Ne Hai**”.

To achieve this abstraction, java technology has given us an interface named “Runnable”. Runnable has only one method run () that we will override and put out task inside it.

Create a thread via Inheritance.

Thread class has Implemented the “**Runnable interface**” and overridden the run () method. If we extend the “Thread” class then we don’t need to implement the “Runnable” interface. In this case we also don’t need to create object of thread class because child class object will have everything.

1. Extends the thread class and override the run () method of "**Thread**" class. Write the task's code inside run () method.

Start a thread via implementing Runnable Interface.

If we are starting a thread via implementing "Runnable" Interface then we have to create object of "Thread" class and send the child class reference id to "Thread" class constructor.

Tips:

1. Thread ka memory me aana aur thread ka chalna alag bat hai.
2. The ending of thread means there is nothing to execute in run ().
3. To start a thread we need to call start () method of "Thread" class.
4. Always keep a separate class to start a main thread. Main thread is started by JVM Implicitly. You may notice we never call start () method to start main thread.
5. If we want to perform a different task on each thread then always keep a separate thread class for each task.
6. If we want to perform common task on each thread then make a single thread class and create a multiple object that thread class.
7. Almost all methods of thread class is native. Native method does not have any body.
8. Threading is the best example of abstraction because thread class has all methods as native and all works are done by operating system.
9. If we don't give a name to a thread then JVM will give a name by default.
10. If we call run () method explicitly then no new thread is going to be started. It will call normally as we call other normal non-static methods.

Program Create Thread Via Inheritance:

// If we want to perform a different-different task on each thread then always keep a separate thread class for each task.

// Thread.currentThread() returns the reference id of current object. It never returns null because in java no programs run without thread.

```
class ThreadOne extends Thread{

   ThreadOne(String s){
     super(s);       // Thread Constructor that accept thread name

// If we want to start a thread just after creating an object
    // start();
   }

   public void run(){
      for(int i=0; i<5; i++){
                        System.out.println(Thread.currentThread().getName());

// To See the output with normal speed, sleeping the thread for a while
         try{
            Thread.sleep(1000);
         }catch(Exception e){
                e.printStackTrace();
         }

      }
 System.out.println(Thread.currentThread().getName() + " Dead");
   }
}

class ThreadTwo extends Thread{

   ThreadTwo(String s){
      super(s);
   }

   public void run(){
      for(int i =0; i<10; i++){
                        System.out.println(Thread.currentThread().getName());
         try{
            Thread.sleep(1000);
         }catch(Exception e){

         }
      }

 System.out.println(Thread.currentThread().getName() + " Dead");
   }
}

class ThreadThree extends Thread{
   ThreadThree(String s){
      super(s);
   }
```

```
    public void run(){

        for(int i=0; i < 15; i++){
                        System.out.println(Thread.currentThread().getName());
        try{
           Thread.sleep(1000);
        }catch(Exception e){}
      }

 System.out.println(Thread.currentThread().getName() + " Dead");
    }
}

class RunThread{

    public static void main(String...a){
       ThreadOne to1 = new ThreadOne("Thread One");
       ThreadTwo tw1 = new ThreadTwo("Thread Two");
       ThreadThree tt1 = new ThreadThree("Thread Three");

       // Which Thread will execute firs is decides by the JVM
       to1.start();
       tw1.start();
       tt1.start();

       for(int i=0; i<20; i++){
                        System.out.println(Thread.currentThread().getName());

// To See the output with normal speed, sleeping the thread for a while
          try{
             Thread.sleep(1000);
          }catch(Exception e){
                e.printStackTrace();
          }
       }
    }
}

// If we want to perform a common task on each thread then make a single thread class and
create a multiple objects of thread class.

class CommonThread extends Thread{

    CommonThread(String s){
       super(s);
    }


public void run(){
   System.out.println(Thread.currentThread().getName() + " Common Tasks");
    }
```

```
}

class RunThread{

  public static void main(String...a){

   CommonThread ct1 = new CommonThread("Thread Common Task 1");
   CommonThread ct2 = new CommonThread("Thread Common Task 2");
   CommonThread ct3 = new CommonThread("Thread Common Task 3");

  // main thread always executes first
  // Which Thread will execute firs is decides by the JVM
       ct1.start();
       ct2.start();
       ct3.start();
    }
}
```

Program to start a thread via Runnable interface:

If we are starting a thread via implementing “Runnable” Interface then we have to create object of “**Thread**” class and send the child class reference id to “Thread” class constructor.

```
class ThreadOne implements Runnable{
    int x;
    public void run(){
       for(int i=0; i<3; i++){
          System.out.println(Thread.currentThread().getName() + " " + x);

            }

       System.out.println(Thread.currentThread().getName() + "Dead");
    }
}

class ThreadTwo implements Runnable{
    int x;
    public void run(){
       for(int i=0; i<3; i++){
          System.out.println(Thread.currentThread().getName() + " " + x);
       }

       System.out.println(Thread.currentThread().getName() + "Dead");
    }
}
```

```
class RunThread{
   public static void main(String...a){

        ThreadOne to = new ThreadOne();
      ThreadTwo tw = new ThreadTwo();
    // Initializing the data member of our thread classes.
      to.x = 10;
      tw.x = 20;

        // (registration) sending the reference id via constructor
      Thread to1 = new Thread(to, "Thread One");
      Thread tw1 = new Thread(tw, "Thread Two");
    // starting the constructor
      to1.start();
      tw1.start();
   }
}
```

What happens when calls the run () method explicitly???

If we call run () method explicitly then no new thread is going to be started. It will call normally as we call other normal non-static methods.

```
class ThreadOne extends Thread{

   ThreadOne(String s){
      super(s);
   }

   public void run(){

// It will print main thread name because no new thread is started. Main thread always runs.
      System.out.println(Thread.currentThread().getName());
   }
}


class RunThread{

   public static void main(String...a){
      ThreadOne to = new ThreadOne("Sayeed Thread");
      to.start();

      to.run();
   }
}
```

Java Scheduler & Life Cycle of Thread:

Thread scheduler in java is the part/thread of the JVM that decides which Threads should run.

There is no guarantee that which runnable thread will be chosen by thread scheduler.

Only one thread at a time can run in a single process.

Important: The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task come into existence.

Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The Scheduler then determines which tasks should execute next based on priority and other factors.

As we know well that at a time only one thread is executed if I sleep a thread for the specified time then the thread scheduler picks up another thread and so on.

Always maximum priority thread will be in running state. This is hard core rule that if lower priority thread is in running state then thread scheduler will stop the lower priority thread and start the higher priority thread.

If all Threads have same priority then it will run **randomly**. No threat can be made dead forcedly.

Put the Image of Life Cycle Made By Manish Sir

If a thread scheduler of a JVM is not mapped with OS thread scheduler then it is known as **green thread.** It does not happen usually only 1% chances that it can happen.

Constructors of Thread class:

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r, String name)
5. Thread(ThreadGroup tg, Runnable r)
6. Thread(ThreadGroup tg, Runnable r, String s)

Commonly used methods of Thread class:

public void run(): is used to perform action for a thread.

public void start(): starts the execution of the thread. JVM calls the run() method on the thread.

public void sleep(long milliseconds):

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The argument value for Milliseconds can't be negative, else it throws IllegalArgumentEsception. It comes in checked exception so demands to handle the exception.

public void sleep (long milliseconds, int Nanos): There is another overloaded method of sleep(). Nanoseconds takes values 0 to 9, 99,999.

public void join(): waits for a thread to die.

Scenario where thread will be use:

#. Order of execution of thread will be achieve using join() method.

#. To achieve dependency we will use join () method.

#. Join () method from Thread class is an important method and used to impose order on execution of multiple Threads. Concept of joining multiple threads is very popular on multi-threading Interview-

Questions.

Scenario: You have three threads T1, T2, and T3. How do you ensure that they have finished in order T1, T2 and T3. This question illustrate power of join () method on multi-threading programming.

Solution: We can do this by using join () by calling T1.join() from T2 and T2.join() from T3. In this case thread T1 will finish first followed by T2 and T3.

#. Join() method is final in java.lang.Thread class and we can't override it.

#. Join() method throws InterruptedException if another thread interrupted waiting thread as a result of join() call.

public void join(long milliseconds): waits for a thread to die for the specified milliseconds.

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY = 1;
2. public static int NORM_PRIORITY = 5;
3. public static int MAX_PRIORITY = 10

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10. We cannot give more than 10. It will throw IllegalArgumentException.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

public boolean isAlive(): tests if the thread is alive. Return “true” if thread is alive else “false”.

public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void suspend(): is used to suspend the thread(depricated).

public void resume(): is used to resume the suspended thread(depricated).

public void stop(): is used to stop the thread(deprecated).

public boolean isDaemon(): tests if the thread is a daemon thread.

public void setDaemon(boolean b): marks the thread as daemon or user thread. If we want to make a user thread as **daemon** then It must not be started. It will throw **IllegalThreadStateException**.

public void interrupt(): interrupts the thread.

public boolean isInterrupted(): tests if the thread has been interrupted.

public static boolean interrupted(): tests if the current thread has been interrupted.

Set Thread Priority

Program setPriority():

```
class TakeFood implements Runnable{
   public void run(){
      for(int i=0; i<3; i++){
         System.out.println(Thread.currentThread().getName());
      }

      System.out.println("Thread Dead");
   }

}

class GetCloths implements Runnable{
   public void run(){
      for(int i=0; i<5; i++){
         System.out.println(Thread.currentThread().getName());
      }

      System.out.println("Thread Dead");
   }

}

class RunThread{

   public static void main(String...a){

      System.out.println(Thread.currentThread().getPriority());

      TakeFood tf = new TakeFood();
      GetCloths gc = new GetCloths();

      Thread tt = new Thread(tf, "Thread One");
      Thread tt1 = new Thread(gc, "GetCloths");

      tt.setPriority(Thread.MIN_PRIORITY);
      tt1.setPriority(Thread.MAX_PRIORITY);
      //tt1.setPriority(Thread.NORM_PRIORITY);

      System.out.println(tt.getPriority());
      System.out.println(tt1.getPriority());

      tt.start();
      tt1.start();
   }
}
// getCloths thread will run first because we set the high priority.
```

Join Thread

Program 1 for join():

```
// Jis Bhi Thread ke andar join() chalta hai, use wait karna padega.

class ThreadOne implements Runnable{

    public void run(){
        System.out.println("Start " + Thread.currentThread().getName());

// Thread One is sleeping for 4 second so main thread must continue running. But
we have joined ThreadOne in main thread so main thread won't run and wait to end
ThreadOne

        try{ Thread.sleep(4000); }
        catch(Exception e){
            System.out.println(e);
        }
        System.out.println("End " + Thread.currentThread().getName());
    }
}


class ThreadTwo implements Runnable{

    public void run(){
        System.out.println("Start " + Thread.currentThread().getName());
        try{ Thread.sleep(2000); }
        catch(Exception e){
            System.out.println(e);
        }
        System.out.println("End " + Thread.currentThread().getName());
    }
}

class ThreadThree implements Runnable{

    public void run(){
        System.out.println("Start " + Thread.currentThread().getName());
        try{ Thread.sleep(2000); }
        catch(Exception e){
            System.out.println(e);
        }
        System.out.println("End " + Thread.currentThread().getName());
    }
}

class RunThread{
```

```
    public static void main(String...a){
        // Objects of our thread classes
          ThreadOne t1 = new ThreadOne();
          ThreadTwo t2 = new ThreadTwo();
          ThreadThree t3 = new ThreadThree();


// Sending reference id to Thread Class via constructor
        Thread th1 = new Thread(t1, "Thread One");
        Thread th2 = new Thread(t2, "Thread Two");
        Thread th3 = new Thread(t3, "Thread Three");

        // Only Starting ThreadOne for better understanding.
        th1.start();
        //th2.start();
        //th3.start();

// joined the ThreadOne inside main Thread. Even ThreadOne become idol for a
while, main thread won't continue.
        try{
           th1.join();
           //th2.join();
           //th3.join();
        }
        catch(Exception e){
           e.printStackTrace();
        }

// This part will run after ending of all threads because have joined other threads.
Because Jis Bhi Thread ke andar join() chalta hai, use wait karna padega.

        for(int i=0; i<3; i++){
           System.out.println(Thread.currentThread().getName());
        }

        try{ Thread.sleep(1000); }
        catch(Exception e){
           System.out.println(e);
        }

        System.out.println(Thread.currentThread().getName() + " Dead");

    }
}
```

OutPut:

```
Start Thread Two
Start Thread Three
Running by: Thread One at Sun Feb 18 12:11:11 IST 2018
End Thread Two
End Thread Three
```

```
Running by: Thread One at Sun Feb 18 12:11:13 IST 2018
Running by: Thread One at Sun Feb 18 12:11:14 IST 2018
End by: Thread One at Sun Feb 18 12:11:15 IST 2018
main
main
main
main Dead
```

Program 2 join():

```
class MyRunnable implements Runnable{
  public void run()
   {
    System.out.println("Thread started "+Thread.currentThread().getName());

     try{
           Thread.sleep(4000);
       }
       catch(InterruptedException e)
       {
          e.printStackTrace();
       }
     System.out.println("Thread ended "+Thread.currentThread().getName());
    }
}


public class ThreadJoin
{
   public static void main(String...s)
   {
       // Our Common Task Thread, Our Different Thread Object
       Thread t1 = new Thread(new MyRunnable(),"t1");
       Thread t2 = new Thread(new MyRunnable(),"t2");
       Thread t3 = new Thread(new MyRunnable(),"t3");

// Starting the 1st Thread, and calling join() for 2 second, so main will continue after
2 second.
       t1.start();
       try
       {
          t1.join(2000);
       }
       catch(Exception e)
       {
```

```
            e.printStackTrace();
        }

// Main thread is continue after 2 second. Now starting 2nd thread.

        t2.start();

// Only start a thread when thread1 is dead.
// Joining thread1 to with no time it means thread1 will run completely then main
thread will continue.
        try{
            t1.join();
        }catch(InterruptedException e)
        {
            e.printStackTrace();
        }

// Thread1 is ended. Main thread is again got life cycle.
//Thread3 is starting.
        t3.start();

// Now joining all threads with not time. means all threads will be end then main
thread will continue.

       try{
       // Thread1 is already ended above.
        t1.join();
        // Thread will complete.
        t2.join();
        // Thread3 will complete
        t3.join();
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }

        // Main Thread Continuing.
        System.out.println("All threads are dead,exiting main thread");
    }
}
```

OutPut:

```
Thread started:::t1
Thread started:::t2
Thread ended:::t1
Thread started:::t3
Thread ended:::t2
Thread ended:::t3
All threads are dead, exiting main thread
```

Program 3 join():

```
import java.util.Date;

class ThreadOne implements Runnable{

   public void run(){
      for(int i=0; i<3; i++){

         System.out.println("Running by: " + Thread.currentThread().getName() + "
at " + new Date());

          try{
            Thread.sleep(1000);
         }catch (Exception e){
            e.printStackTrace();
         }
      }
      System.out.println("End by: " + Thread.currentThread().getName() + " at " +
new Date());
   }
}

class RunnableThread{

   public static void main(String...a){
      // Main thread started by JVM

      ThreadOne to = new ThreadOne();
      Thread t1 = new Thread(to, "T1");
      Thread t2 = new Thread(to, "T2");
      Thread t3 = new Thread(to, "T3");

// First t1 will finish because t1 has joined the main thread. Even t1 is sleeping for 1
second.

      t1.start();
      try{
         t1.join();
      }catch(Exception e){
         e.printStackTrace();
      }

// Main thread got the life cycle.
// Then t2 will finish because t2 has also joined the main thread.
      t2.start();
      try{
         t2.join();
```

```
        }catch(InterruptedException e){
            e.printStackTrace();
        }

        // Main thread got the life cycle.
// Then t3 will finish because t2 has joined the main thread.
        t3.start();
        try{
            t3.join();
        }catch(InterruptedException e){
            e.printStackTrace();
        }

// Main thread got the life cycle.
        System.out.println("--- More Threads And Main Thread Will Run Simultaneously
---");

// Now starting more threads but they will run with main threads
        Thread t4 = new Thread(to, "T4");
        Thread t5 = new Thread(to, "T5");

            t4.start();
            t5.start();

        for(int i=0; i<3; i++){
            System.out.println("Main is running by: " +
Thread.currentThread().getName() + " at " + new Date());
            try{
                Thread.sleep(1000);
            }catch (Exception e){
                e.printStackTrace();
            }
        }

        System.out.println("Main Thread is Dead");
    }
}
```

```
E:\Java\JavaPrograms\Threading\Join()>java RunnableThread
Running by: T1 at Sun Feb 18 11:59:49 IST 2018
Running by: T1 at Sun Feb 18 11:59:51 IST 2018
Running by: T1 at Sun Feb 18 11:59:52 IST 2018
End by: T1 at Sun Feb 18 11:59:53 IST 2018
Running by: T2 at Sun Feb 18 11:59:53 IST 2018
Running by: T2 at Sun Feb 18 11:59:54 IST 2018
Running by: T2 at Sun Feb 18 11:59:55 IST 2018
End by: T2 at Sun Feb 18 11:59:56 IST 2018
Running by: T3 at Sun Feb 18 11:59:56 IST 2018
Running by: T3 at Sun Feb 18 11:59:57 IST 2018
Running by: T3 at Sun Feb 18 11:59:58 IST 2018
End by: T3 at Sun Feb 18 11:59:59 IST 2018
--- More Threads And Main Thread Will Run Simultaneously ---
Running by: T5 at Sun Feb 18 11:59:59 IST 2018
Running by: T4 at Sun Feb 18 11:59:59 IST 2018
Main is running by: main at Sun Feb 18 11:59:59 IST 2018
Running by: T5 at Sun Feb 18 12:00:00 IST 2018
Running by: T4 at Sun Feb 18 12:00:00 IST 2018
Main is running by: main at Sun Feb 18 12:00:00 IST 2018
Running by: T4 at Sun Feb 18 12:00:01 IST 2018
Running by: T5 at Sun Feb 18 12:00:01 IST 2018
Main is running by: main at Sun Feb 18 12:00:01 IST 2018
End by: T4 at Sun Feb 18 12:00:02 IST 2018
End by: T5 at Sun Feb 18 12:00:02 IST 2018
Main Thread is Dead
```

Synchronization

Synchronization means (Taal Mel). (Bahut Sare Kaamo Ka Taal Mel) like Audio Debugging, Pared, Metro Trains Timing and etcetera.

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

We don’t need synchronization all the time. We use synchronization when scenarios like we want share common resource to multiple threads. We mainly use synchronization where we want to allow only one thread to access the shared resource (object) at a time.

Why do we want to share common object?

Suppose there is an EMI calculator web-app and there is a class named “EMICalc” that calculates the employee EMI. If we take example of web application (Website) where millions of user comes simultaneously. In this case we can’t make that much object because heap-area can’t have that much space. To solve this problem we can make single object and share them between all millions of users.

Problem came while sharing the common object:

Whenever we share common object, we also share data member value. It means if one thread change the data member value the other thread will also affect.

Let’s understand this concept with a java program.

```
class ThreadOne extends Thread{

    Shared s;

    ThreadOne(Shared s, String str){
        super(str);
```

```
        this.s = s;
        start();
    }

    public void run(){
        System.out.println("Sum Of 10, 20 is: " + s.add(10,20));
        //s.show(Thread.currentThread().getName(), 10);
    }
}

class ThreadTwo extends Thread{
    Shared s;

    ThreadTwo(Shared s, String str){
        super(str);
        this.s = s;
        start();
    }

    public void run(){
        System.out.println("Sum Of 100, 200 is: " + s.add(100,200));
        //s.show(Thread.currentThread().getName(), 20);
    }
}

class ThreadThree extends Thread{
    Shared s;

    ThreadThree(Shared s, String str){
        super(str);
        this.s = s;
        start();
    }

    public void run(){
        System.out.println("Sum Of 1000, 2000 is: " + s.add(1000,2000));
        //s.show(Thread.currentThread().getName(), 30);
    }

}

class Shared{
    int x,y;

    synchronized void show(String s, int a){

        x = a;
        System.out.println("Starting in Method " + s + " " + x);
        try{
            Thread.sleep(2000);
        }catch(Exception e){
```

```
            e.printStackTrace();
        }

        System.out.println("Exit From Method" + s + " " + x);
    }

     int add(int a, int b){

// Suppose ThreadOne gets the first entry that is sending 10 and 20. 10 is inside x. 20 is inside 20
        this.x = a;
        this.y = b;

// ThreadOne is sleeping for 2 seconds, means life cycle is free and ThreadTwo gets chance to
run .
 // When ThreadTwo run, it changed the value of x, y 100 and 200. ThreadTwo Also slept for 2
seconds.

        try{
            Thread.sleep(2000);
        }catch(Exception e){
            e.printStackTrace();
        }

// Now Again ThreadOne gets the chance and continue.

// ThreadOne is continuing and returning the sum of x and y that is now 100 and 200.

// But do you remember ThreadOne has sent 10,20 and he is getting back 300.
// Here One Thread is interfering other Thread data, also called data curropt.
        return x+y;
    }
}


class RunSync{
    public static void main(String...a){
        Shared sd = new Shared();
        ThreadOne to = new ThreadOne(sd, "TOne");
        ThreadTwo tw = new ThreadTwo(sd, "TTwo");
        ThreadThree tt = new ThreadThree(sd, "TTh");
    }
}
```

Implementation of Synchronization:

Only non-static data member goes on heap area that means we only have to protect non-static data members. As development rule says always make non-static data member

private. Set their value inside method and also get their value by calling method. Usually those kind of methods called setter and getter.

So all we need to do in above program is qualify the show() and add() method with keyword “**synchronized**”.

How Synchronization works:

In java every object maintains a one implicit “**lock**” on it which is known as monitor. If any thread wants to call any synchronized method of a particular object then it must be having a lock of that object.

In above program “**Shared**” class object also has one implicit “**lock**”. We have three threads in our above program, if any of them wants to call add() or show() method of shared class then it must have lock of shared class. At a time only one thread can have lock of shared class.

It means if ThreadOne reaches first to add() method then ThreadOne will get “**lock**” of shared class and if it sleeps then it will go to blocked pool with keeping that “lock”. Now ThreadTwo will get chance to run but it won’t be able to call add() method because that demands “**Shared class object lock**”. Shared class object lock is in blocked pool it mean no other thread can access add() or show() method until ThreadOne dead.

5 Frequently Asked Interview Question From Synchronized Methods.

1. **Condition One:** Two Threads sharing common object (T1 and T2). That common object has two synchronized method show1 () and show2 (). T1 is calling show1 () and T2 is calling show2 (). If T1 first gets entry inside show1 () and sleep for a while, will T2 be able to get entry inside show2 () method? Because both are calling different methods.

Ans: No T2 won't be able to get inside because T1 first enters and get the Object lock. When T1 sleeps, it also keep the object lock. So T2 won't be able go inside show2 cause of Object lock.

Synchronized class also called "**Thread safe**". If we want to make our own synchronized class then make all the methods of that class synchronized. If you open "**StringBuffer**" class then you will get that all methods there are synchronized. Even this is the only difference between **StringBuffer**" class and "**StringBuilder**" class. StringBuilder class methods are not synchronized, except this both classes are same.

If any class is by default synchronized or thread safe that means at a time only a one thread can access the object of that class if all the threads are having same object of that class.

Program: One By One Access.

```
class Girl{

    synchronized void g1(String str, int x){
        System.out.println("G1 Starting: Thread Name is : " + str + " and
value of x is: " + x);

        try{
            System.out.println("G1 is sleeping but G2 is not able to access
g2().");
            Thread.sleep(2000);

        }
        catch(Exception e){
            e.printStackTrace();
        }

        System.out.println("G1 Ending: Thread Name is : " + str + " and
value of x is: " + x);
    }

    synchronized void g2(String str, int x){
        System.out.println("G2 Starting: Thread Name is : " + str + " and
value of x is: " + x);
```

```
        try{
            Thread.sleep(2000);
        }
        catch(Exception e){
            e.printStackTrace();
        }

        System.out.println("G2 Ending: Thread Name is : " + str + " and
value of x is: " + x);
    }
}

class Boy1 extends Thread{
    Girl g;
    Boy1(Girl g, String s){
        super(s);
        this.g = g;
        start();
    }

    public void run(){
        g.g1(Thread.currentThread().getName(), 10);
    }
}

class Boy2 extends Thread{
    Girl g;
    Boy2(Girl g, String s){
        super(s);
        this.g = g;
        start();
    }

    public void run(){
        g.g2(Thread.currentThread().getName(), 20);
    }
}

class GoForDate{
    public static void main(String...a){
        Girl grl = new Girl();
        Boy1 b1 = new Boy1(grl, "Vineet");
        Boy2 b2 = new Boy2(grl, "Javed");
    }
}
```

2. **Condition Two:** Two Threads sharing common object (T1 and T2). That common object has two method. First is synchronized show1 () and simple show2 (). T1 is calling show1 () and T2 is calling show2 (). If T1 first gets entry inside show1 () that is synchronized and sleep for a while, will T2 be able to get entry inside show2 () method? Because both are calling different methods.

Ans: Yes T2 will be able access show2 (). Because show2 () is not synchronized so it does not require an object lock. Both program will run parallel. In this case data will also corrupt.

3. **Condition Three:** Two Threads sharing common object (T1 and T2). That common object has one method. That method is static synchronized show1 (). T1 and T2 is both calling same method show1 () by ClassName. If T1 first gets entry inside show1 () and sleep for a while, will T2 be able to get entry inside show1 () method?

Ans: In java, every class is also having a one Implicit lock on it. So both will access show1 () method one by one.

4. **Condition Four:** Two Threads sharing common object (T1 and T2). That common object has one method. That method is static synchronized show1 (). T1 is calling by ClassName and T2 is calling by Object. If T1 first gets entry inside show1 () and sleep for a while, will T2 be able to get entry inside show1 () method?

Ans: In case of static synchronized method, lock is always achieve on a class hardly matters whether it is called by the class name or by the object. So both will access show1 () method one by one.

5. **Condition Five:** Two Threads sharing common object (T1 and T2). That common object has two method. First is **static synchronized show1 ()** and **non-static synchronized show2 ()**. T1 is calling show1 () by ClassName and T2 is calling show2 () by Object. If T1 first gets entry inside show1 () that is static synchronized and sleep for a while, will T2 be able to get entry inside non-static synchronized show2 () method?

Ans: One By One.

Ways to Achieve Synchronization:

Synchronization always degrades the performance that's why there are 6 ways to achieve synchronization. We have already seen one of them called "**synchronized method**".

2nd Way to Achieve Synchronization called "Synchronized block"

As we know, we make synchronized method to prevent data member corruption. Suppose a method has 100 lines and data member initialization/assignment code is only of 5 lines. It means if we make this method synchronized and once first thread entered in this method then waiting time for second thread will be 100 lines. So instead of making whole method synchronized, why don't we synchronized only those 5 lines. It will reduce the waiting time 95%.

Definition & Differences: In case of "**synchronized method**", we make whole method synchronized. In case of "**synchronized block**" we only make the particular portion of a method synchronized rather than a whole method.

In case of synchronized method "**lock**" is always achieved on current object. In case of synchronized block "**lock**" can be achieved on any object.

Synchronized block can also be used to make object of any class synchronized.

Java Program:

```
class Shared{

    int x;
    void show(String str, int x){

    // suppose ThreadOne gets the chance to run.

        System.out.println("Starting: " + str + " and value of x is: " + x);
        System.out.println(str + " run 50 lines");
        synchronized(this){
            this.x  = x;
        }

// ThreadOne is sleeping with lock. But still ThreadTwo will get chance to run and it will run his first
50 lines then wait.

// You may notice synchronized method don't allow to even enter inside method but using
synchronized block ThreadTwo is getting chance to run at-least 50 lines.

// It is reducing the waiting time. :)

        try{
            Thread.sleep(2000);
        }catch(Exception e){
            e.printStackTrace();
        }

        System.out.println(str + " is running more 40 lines");
        System.out.println("Ending: " + str + " and value of x is: " + x);

    }

}

class ThreadOne extends Thread{

    Shared s;

    ThreadOne(Shared s, String str){
        super(str);
        this.s = s;
        start();
    }

    public void run(){
        s.show(Thread.currentThread().getName(), 100);
    }
}
```

```
class ThreadTwo extends Thread{

   Shared s;

   ThreadTwo(Shared s, String str){
      super(str);
      this.s = s;
      start();
   }

   public void run(){
      s.show(Thread.currentThread().getName(), 200);
   }
}

class ThreadThree extends Thread{

   Shared s;

   ThreadThree(Shared s, String str){
      super(str);
      this.s = s;
      start();
   }

   public void run(){
      s.show(Thread.currentThread().getName(), 300);
   }
}


class RunSync{
   public static void main(String...a){

      Shared ss = new Shared();
      ThreadOne t = new ThreadOne(ss, "Thread One");
      ThreadTwo tw = new ThreadTwo(ss, "Thread Two");
      ThreadThree th = new ThreadThree(ss, "Thread Three");

   }
}

// In this program using synchronized block Temp class also became synchronized because
Threads are also visiting Temp class One by one.
/* Output
   Starting: Thread Two and value of x is: 200
   Starting: Thread Three and value of x is: 300
   Starting: Thread One and value of x is: 100
   Thread Three run 50 lines
```

```
   Thread Two run 50 lines
   Thread One run 50 lines
   Thread Two is running more 40 lines
   Thread Three is running more 40 lines
   Thread One is running more 40 lines
   Ending: Thread Two and value of x is: 200
   Ending: Thread Three and value of x is: 300
   Ending: Thread One and value of x is: 100
*/



// What happens when synchronized method and synchronized block in Shared object.
No data will corrupt.

class Shared{

   int x;

   synchronized public void show(String str){
      System.out.println("Starting show: " + str);

      try{
         Thread.sleep(2000);
      }catch(Exception e){
         System.out.println(e);
      }

      System.out.println("Ending show" + str);
   }

   public void display(String str, int a){
      System.out.println("Starting Display " + str);

      synchronized(this){
         x = a;
         System.out.println("Synchronized block " + str + " " + x);

         try{
            Thread.sleep(2000);
         }catch(Exception e){
            System.out.println(e);
         }
      }

      System.out.println("Ending Display" + str);
   }
}

class ThreadOne extends Thread{
```

```
    Shared s;

    ThreadOne(Shared s, String str){
        super(str);
        this.s = s;
        start();
    }
    public void run(){
        this.s.show("Thread One");
    }
}

class ThreadTwo extends Thread{
    Shared s;
    ThreadTwo(Shared s, String str){
        super(str);
        this.s = s;
        start();
    }

    public void run(){
        this.s.display("Thread Two", 10);
    }
}

class RunSync{
    public static void main(String...a){

        Shared s = new Shared();

        ThreadOne t1 = new ThreadOne(s, "Thread One");
        ThreadTwo t2 = new ThreadTwo(s, "Thread Two");
    }
}
```

3rd Way to Achieve Synchronization (Reentrant Lock)

As we know we use synchronized method or synchronized block when we want to share an object and wants to perform task one by one. Here we achieve this via object lock. Only one thread can have object at a time.

But requirement came to perform Thread tasks one by one but they are not sharing the same object. They all have different resource (object) so synchronized block won't fulfill

the requirement. Because synchronized block do this via object lock and in this case all threads have different object lock. To solve this perform **ReEntrantLock** came.

HR of a company wants to hire a 3 Java developers. For this he selects 3 Tech-Leads that will conduct the test and select candidates. One Tech Leads will select only one candidate. The problem is there is only one Question-Paper and Xerox machine is not working. So now at time only one Tech-Lead is having question-paper. One Tech-Lead will conduct the test and other tech-leads have to wait for question-paper. When one tech-lead has taken the test then he gives question-paper back. Now other tech-leads can take the question-paper and conduct the test.

We will give ReentrantLock class object to all threads and the one who got chance to run first will call lock() method of ReentrantLock class. Once the work is done we can call unlock() method of ReentrantLock().

Question paper is one and and three interviewer is going to use it one by one.

Tips: We can't use more than one unlock method in a single method.

```
import java.util.concurrent.locks.;
class ThreadOne extends Thread{

   ReentrantLock qp;

   ThreadOne(ReentrantLock qp, String s){
      super(s);
      this.qp = qp;
   }

   public void run(){
      System.out.println(getName() + " is waiting for question paper");
      // Lock the question-paper
      qp.lock();

      System.out.println(getName() + " got the question paper");
      System.out.println(getName() + " Start taking test");


      try{
         Thread.sleep(2000);
      }catch(Exception e){
         System.out.println("Thread is interrupted");
      }
```

```
System.out.println(getName() + " has taken the test and returning the lock")

// Test has taken, unlock the paper so other tech-leads will be able to conduct test.
      qp.unlock();
   }

}

class RunThread{

   public static void main(String...a){
      // Predefined class given by java.
      ReentrantLock qp = new ReentrantLock();

      new ThreadOne(qp, "Tech Lead 1").start();
      new ThreadOne(qp, "Tech Lead 2").start();
      new ThreadOne(qp, "Tech Lead 3").start();
   }
}
```

suspend() and resume()

suspend(): suspend method is used to send a thread into blocked pool for infinity time. It will be in blocked pool until other thread call resume().

Why suspend() and resume() deprecated: because whenever we call suspend() from synchronized method, or from synchronized block then one Dead-Lock will be created. The program will be hang

Suppose ThreadOne is suspended and it is inside blocked pool with lock. If resume method is calling for ThreadOne from synchronized method then ThreadTwo will require a lock. But lock is inside Blocked pool with ThreadOne. This situation will create a dead-lock.

Program Example:

```
class ThreadOne extends Thread{

   Shared s;

   ThreadOne(Shared s, String str){
      super(str);
      this.s = s;
   }
```

```
    public void run(){
        s.show(this);

    }
}

class ThreadTwo extends Thread{

    ThreadOne to;
    Shared s;

    ThreadTwo(Shared s, ThreadOne to, String str){
        super(str);
        this.to = to;
        this.s = s;
    }

    public void run(){
        s.display(to);
    }
}


class Shared{

    synchronized void show(ThreadOne to){
        System.out.println("Entered " + Thread.currentThread().getName());

        // Going To Blocked Pool For Infinity Time
        to.suspend();

        System.out.println("Exit " + Thread.currentThread().getName());
    }

    synchronized void display(ThreadOne to){
        System.out.println("Entered " + Thread.currentThread().getName());
        // trying to resume the ThreadOne but it won't even come inside this synchronized method.
        // But if we resume the ThreadOne from out the synchronized method or block, then it will
run fine.
        to.resume();

        System.out.println("Exit: " + Thread.currentThread().getName());
    }
}

class RunThread2{
    public static void main(String...a){
        Shared s = new Shared();
        ThreadOne to = new ThreadOne(s, "Thread One");
        ThreadTwo tw = new ThreadTwo(s, to, "Thread Two");
```

```
        to.setPriority(10);

        to.start();
        tw.start();
    }
}
```

To solve this problem wait() and notify() introduced.

#. wait(), notify() and notifyAll are methods of object class.

#. **suspend()** method only releases a processor life cycle from any Thread State but **wait()** releases both processor life cycle and a lock from any Thread.

```
class Shared{

    int flag = 0;
    int data = 0;

    synchronized public void submit(){

        // Proper ways is first submit the amount then withdraw.
        // setting the flag value 1 so withdraw() won't call wait()
            // because flag=1 means I have already visited submit() and value is submitted

        // setting flag value 1
        flag = 1;
        try{
            Thread.sleep(2000);
        }catch(Exception e){
            e.printStackTrace();
        }

        // submitting data and returning.
        this.data = 10;
        System.out.println("Value Deposited");
// It will resume the Submit Thread. there is no problem if calling notify() without calling wait()
        notify();    // notifyAll resumes the all threads from blocked pool.
    }

    synchronized public int withdraw(){
// If flag is 0, it means submit method is not called yet. So I need to call wait()
// that will leave the processor and a lock and submit() will call.

      if(flag == 0){
          try{
              System.out.println("Sending to wait so the submit() will run");
```

```
            // will wait until
            wait();
         }catch(Exception e){ e.printStackTrace(); }
      }


      return this.data;
   }
}

class Withdraw extends Thread{

   Shared s;

   Withdraw(Shared s, String str){
      super(str);
      this.s = s;
   }

   public void run(){
      s.withdraw();
   }
}

class Submit extends Thread{

   Shared s;

   Submit(Shared s, String str){
      super(str);
      this.s = s;
   }

   public void run(){
      s.submit();
   }
}



class RunThread3{
   public static void main(String...a){
      Shared s = new Shared();
      Withdraw wt = new Withdraw(s, "Withdraw Thread");
      Submit sb = new Submit(s, "Submit Thread");

      // Want to run withdraw method first so i will be able to see a scenario.
      wt.setPriority(10);

      wt.start();
      sb.start();
```

```
    }
}
```

Dead-Lock Condition

Dead lock is a condition where a thread is waiting for an object lock that another thread holds. Since each thread is waiting for the other thread to release a lock, they both remain waiting for ever in the blocked pool for lock …….. State. The threads are set to be dead lock.

Dead Lock Condition: Object one needs Object Two reference and Object Two need Object One Reference id.

Story: Dikhta nahi Rummy Chal rahi.

```
class ObjectOne{

    public void show(){
        System.out.println("Show From Object One");
    }
}

class ObjectTwo{
    public void display(){
        System.out.println("Display From Object Two");
    }
}

class Shared{

    ObjectOne oo = new ObjectOne();
    ObjectTwo ot = new ObjectTwo();

    public void methodOne(){
        System.out.println("Start methodOne from Shared");

        // oo is having lock of Object One
        synchronized(oo){
            oo.show();

            // sleeping for two second so Thread Two will get chance to run
            try{
```

```
            Thread.sleep(2000);
         }catch(Exception e){
            e.printStackTrace();
         }

         // When it comes back to run, it needs Object Two reference to go inside but object two is
inside
               // blocked pool waiting for object of ObjectOne class
         synchronized(ot){
            ot.display();
         }
      }

      System.out.println("Ending methodOne from Shared");
   }

   public void methodTwo(){
      System.out.println("Start methodTwo from Shared");

      // ot is having lock of Object Two
      synchronized(ot){
         ot.display();

         // Sleeping for two second so Thread One will get chance to run.
         try{
            Thread.sleep(2000);
         }catch(Exception e){
            e.printStackTrace();
         }


         // need object of object One reference to go inside
         synchronized(oo){
            oo.show();
         }
      }

      System.out.println("Ending methodTwo from Shared It Won't Reach Here");
   }
}

class ThreadOne extends Thread{

   Shared sd;

   ThreadOne(Shared sd, String s){
      super(s);
      this.sd = sd;
   }

   public void run(){
```

```
      this.sd.methodOne();
   }
}

class ThreadTwo extends Thread{

   Shared sd;

   ThreadTwo(Shared sd, String s){
      super(s);
      this.sd = sd;
   }

   public void run(){
      this.sd.methodTwo();
   }
}

class RunThread{
   public static void main(String...a){
      Shared s = new Shared();
      new ThreadOne(s, "Thread One").start();
      new ThreadTwo(s, "Thread Two").start();
   }
}
```

Interrupting Threads

A thread terminates when it's run () method returns. In JDK 1.0 there was a stop () method that another thread could call to terminate a thread however that method is now deprecated. There is no longer way to force a thread to terminate however the interrupt () method can be used to request termination of a thread. When the interrupt () method is

called on a thread, the interrupt status of the thread is set. This is a Boolean flag that is set present in an every thread.

Each thread should occasionally check whether it has interrupted. **However, if a thread is blocked, it cannot check an interrupted status**. This is the situation where "**InterruptedException**" comes-in when the interrupt method is called on a blocked thread. The blocking call such as sleep () or wait () is terminated by an InterruptedException. There is no language requirement that a thread that is interrupted should terminate, interrupting a thread simply grapes its attention.

Simple Explanation: Suppose a "ThreadOne" runs and sleeps/waits for 5 seconds. The "ThreadTwo" got life cycle, It executes the interrupt () method on "ThreadOne" like this: "**ThreadOne.interrupt()**" then ThreadOne will break the sleeping time and come back from blocked pool with "**InterruptedException**". It means ThreadOne will only get life cycle to run catch block. It will not continue run () method. If we want to continue after interrupting then we have to sleep ThreadTwo.

Program:

```
class ThreadOne extends Thread{

   ThreadOne(String s){
      super(s);
   }

   public void run(){

      // First ThreadOne will run
      System.out.println(Thread.currentThread().getName() + " Start");

         try{
          Thread.sleep(5000 * 10);
                // sleeping but when ThreadTwo runs interrupt(), it will come back with exception and
                jump to catch block.
      }catch(Exception e){

                System.out.println("Forcefully Interrupted: " +
                Thread.currentThread().getName());

               System.out.println(e);
      }
```

```
        // Only Continuing because thread two sleeps after interrupting.
        System.out.println("Dead Thread: " + Thread.currentThread().getName());
    }
}

class ThreadTwo extends Thread{

    ThreadOne to;

    ThreadTwo(String s, ThreadOne to){
        super(s);
        this.to = to;
    }

    public void run(){

        System.out.println(Thread.currentThread().getName() + " Start");
        to.interrupt();

        // Sleeping ThreadTwo so after interrupting ThreadOne will get back the life cycle.
        try{
            Thread.sleep(2000);
        }catch(Exception e){
            e.printStackTrace();
        }

        System.out.println("Dead Thread: " + Thread.currentThread().getName());
    }
}


class RunThread{
    public static void main(String...a){

        // Wants to run ThreadOne First
          ThreadOne to = new ThreadOne("First");
         to.setPriority(10);
         to.start();

          // Sending reference id of the ThreadOne because wants to run interrupt () on it
        ThreadTwo tt = new ThreadTwo("Second", to);

        tt.start();
    }
}
```

Task Scheduling

Task-Scheduling is used to run a task at scheduled time. You can see, I am inheriting TimeTask that also implements runnable interface.

```
import java.util.*;
import javax.swing.JFrame;

class ThreadOne extends TimerTask {

   int count = 1;

   // run() is a abstract method that defines task performed at scheduled time.

   public void run(){
      if(count == 20){
         System.exit(0);
      }

      JFrame jf = new JFrame();
      jf.setSize(250,300);
      jf.setVisible(true);

      count++;
   }
}


class RunThread{
   public static void main(String...a){
      ThreadOne to = new ThreadOne();
      Timer tm = new Timer();

      // First time running: delay from current time.
      int delay = 5000;

      // Repeating after every defined period
      int period = 1000;

      // Schedule the task.
      System.out.println("Task will start after 5 seconds");
      tm.scheduleAtFixedRate(to, delay, period);
   }
}
```

Daemon Thread

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the **jconsole** in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread? That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

1. **public void setDaemon(boolean status):** is used to mark the current thread as daemon thread or user thread.

2. **public boolean isDaemon():** is used to check that current is daemon.

Rules:

1. Daemon threads are service provider thread.

2. They provide services to other thread.
3. They never show any output.
4. They always run in a background.
5. They will be dead automatically whenever a thread to whom they were providing a service is dead.

For example: Garbage collector's thread: - Daemon thread provide services to main thread and it will automatically dead when main thread dead.

If we want to make a "**User-Thread**" as "**daemon**, it must be started otherwise it will throw "IllegalThreadException".

Inside the Java Virtual Machine, threads come in two flavors, Daemon thread and non-daemon thread. A Daemon thread is ordinarily a thread used by java virtual machine itself such as thread that performs garbage collection.

The application, however can make any thread it creates as Daemon threads. The initial thread of an application is the one that begins at main () is non-demon threads. A java application continues to execute (the virtual machine instances continues to live) as long as any non-daemon threads are still running. When all non-daemon threads of java application terminates the virtual machine instance will exit.

ThreadGroup

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now suspend (), resume () and stop () methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

1) ThreadGroup(String name): creates a thread group with given name.

2) ThreadGroup(ThreadGroup parent, String name): creates a thread group with given parent group and name.

Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

1) int activeCount(): returns no. of threads running in current group.

2) int activeGroupCount(): returns a no. of active group in this thread group.

3) void destroy(): destroys this thread group and all its sub groups.

4) String getName(): returns the name of this group.

5) ThreadGroup getParent(): returns the parent of this group.

6) void interrupt(): interrupts all threads of this group.

7) void list(): prints information of this group to standard console.

Let's see a

```
ThreadGroup tg1 = new ThreadGroup("Group A");

Thread t1 = new Thread(tg1,new MyRunnable(),"one");

Thread t2 = new Thread(tg1,new MyRunnable(),"two");
```

Thread t3 = new Thread(tg1,new MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, “MyRunnable” is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

Thread.currentThread().getThreadGroup().interrupt();

Code to group multiple threads.

```
public class ThreadGroupDemo implements Runnable{
   public void run() {
        System.out.println(Thread.currentThread().getName());
   }

   public static void main(String[] args) {
      ThreadGroupDemo runnable = new ThreadGroupDemo();
      ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

      Thread t1 = new Thread(tg1, runnable,"one");
      t1.start();
      Thread t2 = new Thread(tg1, runnable,"two");
      t2.start();
      Thread t3 = new Thread(tg1, runnable,"three");
      t3.start();

      System.out.println("Thread Group Name: "+tg1.getName());
      tg1.list();
   }
}
```

Program 2: This programs proofs that all threads are applying same work. Threads loop must be run 5 times but before main thread does stops all threads.

```
class Vineet implements Runnable{

   public void run(){
      for(int i =0; i<5; i++){
         System.out.println(Thread.currentThread().getName() + " is playing");
         try{
            Thread.sleep(1000);
         }catch(Exception e){
            System.out.println(e);
```

```
        }
      }

    System.out.println("Won't reach here because i am going to suspend it");
   }
}



class Javed implements Runnable{

   public void run(){
      for(int i =0; i<5; i++){
         System.out.println(Thread.currentThread().getName() + " is playing");
         try{
            Thread.sleep(1000);
         }catch(Exception e){
            System.out.println(e);
         }
      }

   System.out.println("Won't reach here because i am going to suspend it");

   }
}

class RunProgram{


   public static void main(String...a){

      RunProgram friends = new RunProgram("Ducat Group");

      Thread t2 = new Thread( friends, new Javed(), "Business Man" );
      Thread t3 = new Thread( friends, new Vineet(), "NotePad++" );

      t1.start(); t2.start(); t3.start();


      try{
         Thread.sleep(1000);
      }catch(Exception e ){
         System.out.println(e);
      }


     System.out.println(Thread.currentThread().getName());
     friends.stop();
   }
}
```

What is Thread Pulling?

What is Thread Group?

What us JConsole?

Mine:

1. Threading is very important concept for any programming language because using this we can make multi-tasking application. That's why this concept is very important for interview purpose.
2. Every processor have several algorithms that decides which task need to execute first.
3. Windows 3.1 was the first Operating System who achieved Multi-tasking.
4. How one application copy something to another application?

For example: Whenever we copy something from MS word then it stores in clipboard and from there we paste it into other applications. Java also has clipboard API.

#. 10 lakhs Nano seconds => 1 Milliseconds

#. 1 thousand milliseconds => 1 second

#. Can we make our own synchronized class?

Synchronized class also called "**Thread safe**". If we want to make our own synchronized class then make all the methods of that class synchronized. If you open "**StringBuffer**" class then you will get that all methods there are synchronized. Even this is the only difference between **StringBuffer**" class and "**StringBuilder**" class. StringBuilder class methods are not synchronized, except this both classes are same.

If any class is by default synchronized or thread safe that means at a time only a one thread can access the object of that class if all the threads are having same object of that class.