**Multi Threading**

**Information about multithreading:-**

1) The earlier days the computer’s memory is occupied only one program after completion of one program it is possible to execute another program is called uni programming.

2) Whenever one program execution is completed then only second program execution will be started such type of execution is called co operative execution, this execution we are having lot of disadvantages.

a. Most of the times memory will be wasted.

b. CPU utilization will be reduced because only program allow executing at a time.

c. The program queue is developed on the basis co operative execution

**To overcome above problem a new programming style will be introduced is called multiprogramming.**

1) Multiprogramming means executing the more than one program at a time.

2) All these programs are controlled by the CPU scheduler.

3) CPU scheduler will allocate a particular time period for each and every program.

4) Executing several programs simultaneously is called multiprogramming.

5) In multiprogramming a program can be entered in different states.

a. Ready state.

b. Running state.

c. Waiting state.

6) Multiprogramming mainly focuses on the number of programs.

**Advantages of multiprogramming:-**

1. CPU utilization will be increased.

2. Execution speed will be increased and response time will be decreased.

3. CPU resources are not wasted.

**Thread:-**

1) Thread is nothing but separate path of sequential execution.

2) The independent execution technical name is called thread.

3) Whenever different parts of the program executed simultaneously that each and every part is called thread.

4) The thread is light weight process because whenever we are creating thread it is not occupying the separate memory it uses the same memory. Whenever the memory is shared means it is not consuming more memory.

5) Executing more than one thread a time is called multithreading.

**Single threaded model:-**

class Test

{

public static void main(String[] args)

{

System.out.println("Hello World!");

System.out.println("hi rattaiah"); body

System.out.println("hello durgasoft");

}

}

above program only one thread is available is called main thread to know the name of the thread we have to execute the fallowing code.

class Test

{

public static void main(String[] args)

{

System.out.println("Hello World!");

Thread t=Thread.currentThread(); // to know the thread name

System.out.println("currrent thread information is : "+t);//[main,5,main]

System.out.println("currrent thread priority is : "+t.getPriority());//5

System.out.println("currrent thread name is : "+t.getName());

System.out.println("hi suresh");

System.out.println("hello javasoft");

}

}

**In the above program only one thread is available name of that thread is main thread.**

**Multithreaded model:-**

Main thread

Starts Starts starts

Thread C

Thread B

Thread A

Thread may switch or exchange data/result.

**The main important application areas of the multithreading are**

1. Developing video games

2. Implementing multimedia graphics.

3. Developing animations

**There are two different ways to create a thread isavilable**

1) Create class that extending standered java.lang.Thread Class

2) Create class that Implementing java.lang.Runnable interface

**Creation of threads in java**

Thread

Runnable

Thread

extends implements

MyClass

MyThread

**(a)Objects are threads (b)objects with run() body**

**First approach to create thread extending Thread class:-**

**Step 1:-**

**Creates a class that is extend by Thread classes and override the run() method**

class MyThread extends Thread

{

public void run()

{

System.out.println("business logic of the thread");

System.out.println("body of the thread");

}

};

**Step 2:-**

**Create a Thread object**

MyThread t=new MyThread();

**Step 3:-**

**Starts the execution of a thread.**

t.start();

**In this approach take one user defined class class that is extending Thread class .**

**Ex:-**

class MyThread extends Thread

{

public void run()

{

System.out.println("Rattaiah from durgasoft");

System.out.println("body of the thread");

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t=new MyThread();

t.start();

}

}

Note :-

1) Whenever we are calling t.start() method the JVM search for the start() in the MyThread class but the start() method is not present in the MyThread class so JVM goes to parent class called Thread class and search for the start() method.

2) In the Thread class start() method is available hence JVM is executing start() method.

3) Whenever the thread class start() that start() is responsible person to call run() method.

4) Finally the run() automatically executed whenever we are calling start() method.

5) Whenever we are giving a chance to the Thread class start() method then only a new thread will be created.

**Life cycle stages are:-**

**1) New**

**2) Ready**

**3) Running state**

**4) Blocked / waiting / non-running mode**

**5) Dead state**

**New :-**

MyThread t=new MyThread();

**Ready :-**

t.start()

**Running state:-**

If thread scheduler allocates CPU for particular thread. Thread goes to running state The Thread is running state means the run() is executed.

**Blocked State:-**

If the running thread got interrupted of goes to sleeping state at that moment it goes to the blocked state.

**Dead State:-**

If the business logic of the project is completed means run() over thread goes dead state.

**Second approach to create thread implementing Runnable interface:-**

**Step 1:-**

**Creates a class that implements Runnable interface.**

class MyClass extends Runnable

{

public void run()

{

System.out.println("Rattaiah from durgasoft");

System.out.println("body of the thread");

}

};

**Step 2:-**

**Creating a object.**

MyClass obj=new MyClass();

**Step 3:-**

**Creates a Thread class object.**

Thread t=new Thread(obj);

**Step 4:-**

**Starts the execution of a thread.**

t.start();

**implementing Runnable interface**

class MyThread implements Runnable

{

public void run()

{

System.out.println("Rattaiah from durgasoft");

System.out.println("body of the thread");

}

}

class ThreadDemo {

public static void main(String[] args)

{

MyClasss obj=new MyClass();

Thread t=new Thread(obj);

t.start();

}

}

**Step 1:-**

the Class MyClass implements the Runnable interface and overriding run() method and contains the logic associates with the body of the thread.

**Step 2:-**

Creates the object of implementation class this is not like a first mechanism.

**Step 3 :-**

Creates a generic thread object then pass the MyClass reference variable as a parameter to that object.

**Step 4:-**

As a result of third step 3 a thread object is created in order to execute this thread method we need to class start() method. Then new thread is executed.

**We are having two approaches:-**

**First approach:-**

1. By extending the thread class, the derived class itself is a thread object and it gains full control over the thread life cycle.
2. 2) Another important point is that when extending the Thread class, the sub class cannot extend any other base classes because Java allows only single inheritance.
3. **if the program needs a full control over the thread life cycle, then extending the Thread class is a good choice.**

**Second approach:-**

1) Implementing the Runnable interface does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread.

2) By implementing the Runnable interface, the class can still extend other base classes if necessary.

**if the program needs more flexibility of extending other base classes, implementing the**

**Runnable interface would be preferable.**

We are having two approaches two create a thread use any approach based on application requirement.

**Thread life cycle:-**

New

ready

Start()

Running state

If CPU is allocated

sleeping

Sleep()

If time is expired

waiting

Wait()

Notify()

Dead state

If run() completes/

Stop()

**Thread life cycle**

**Internal Implementation of multiThreading:-**

Runnable-------abstract method

Thread----------empty implementation run() method

Mythread-----based on our requirement we are providing implementation (overriding run() method)

**interface Runnable**

**{**

**public abstract void run();**

**}**

**class Thread implements Runnable**

**{**

**public void run()**

**{**

**}**

**};**

**class MyThread extends Thread**

**{**

**public void run()**

**{**

**for (int i=0;i<5 ;i++ )**

**{**

**System.out.println("user implementation");**

**}**

**}**

**};**

**Thread Schedular:-**

Thread scheduler is a part of the JVM. It decides which thread is executed first and which thread is executed next.

Only one thread is executed at a time.

We can’t expect exact behavior of the thread scheduler it is JVM vendor dependent. So we can’t expect output of the multithreaded examples we can say the possible outputs.

Thread Scheduler mainly uses preemptive (or) time slicing to schedule the threads.

**Preemptive scheduling:-**

In this highest priority task is executed first after this task enters into waiting state or dead state then only another higher priority task come to existence.

**Time Slicing Scheduling:-**

A task is executed predefined slice of time and then return pool of ready tasks. The scheduler determines which task is executed based on the priority and other factors.

**Difference between t.start() and t.run():-**

In the case of t.start(), Thread class start() is executed a new thread will be created that is responsible for the execution of run() method.

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

**Note :-**

**Here we are not overriding the run() method so thread class run method is executed which is having empty implementation so we are not getting any output.**

class MyThread extends Thread

{

}

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t=new MyThread();

t.start();

for (int i=0;i<5;i++ )

{

System.out.println("main thread");

}

}

}

**Note :-**

**If we are overriding start() method then JVM is executes override start() method at this situation we are not giving chance to the thread class start() hence n new thread will be created only one thread is available the name of that thread is main thread.**

class MyThread extends Thread

{

Public void start()

{

System.out.println(“override start method”);

}

}

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t=new MyThread();

t.start();

for (int i=0;i<5 ;i++ )

{

System.out.println("main thread");

}

}

}

**Particular task is performed by the number of threads:-**

1) Particular task is performed by the number of threads here number of threads(t1,t2,t3) are executing same method (functionality).

2) In the above scenario for each and every thread one stack is created. Each and every method called by particular Thread the every entry stored in the particular thread stack.

Here Four Stacks are created

Main -----------stack1

t1---------------stack2

t2--------------stack3

t3-------------stack4

class MyThread extends Thread

{

public void run()

{

System.out.println("durgasoft task");

}

}

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

MyThread t3=new MyThread();

t1.start();

t2.start();

t3.start();

}

}

**Ex5:-multiple threads are performing multiple operation.**

class MyThread1 extends Thread

{

public void run()

{

System.out.println("mythread1 task");

}

}

class MyThread2 extends Thread

{

public void run()

{

System.out.println("mythread2 task");

}

}

class MyThread3 extends Thread

{

public void run()

{

System.out.println("Mythread3 task");

}

}

class ThreadDemo

{

public static void main(String[] args)

{

MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

t1.start();

t2.start();

t3.start();

}

}

**Getting and setting names of Thread:-**

1) Every Thread in java has some name if may be default name provided by the jvm or customized name provided by the programmer.

The fallowing methods are useful to set and get the name of a Thred.

**a. Public final String getName()**

**b. Public final void setName(String name)**

**Ex:-**

class MyThread extends Thread

{

}

class Test

{

public static void main(String args[])

{

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

MyThread t=new MyThread();

System.out.println(t.getName());

Thread.currentThread().setName("meena");

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

}

}

**Thread Priorities:-**

1. Every Thread in java has some property. It may be default priority provided be the JVM or customized priority provided by the programmer.

2. The valid range of thread priorities is 1 – 10. Where one is lowest priority and 10 is highest priority.

3. The default priority of main thread is 5. The priority of child thread is inherited from the parent.

4. Thread defines the following constants to represent some standard priorities.

5. Thread Scheduler will use priorities while allocating processor the thread which is having highest priority will get chance first and the thread which is having low priority.

6. If two threads having the same priority then we can’t expect exact execution order it depends upon Thread Scheduler.

7. The thread which is having low priority has to wait until completion of high priority threads.

8. Three constant values for the thread priority.

**a. MIN\_PRIORITY = 1**

**b. NORM\_PRIORITY = 5**

**c. MAX\_PRIORITY = 10**

Thread class defines the following methods to get and set priority of a Thread.

**a. Public final int getPriority()**

**b. Public final void setPriority(int priority)**

Here ‘priority’ indicates a number which is in the allowed range of 1 – 10. Otherwise we will get

Runtime exception saying “IllegalArgumentException”.

Ex:-

class RattaiahThread extends Thread

{

public void run()

{

System.out.println("Enter into thread Rattaiah");

System.out.println("thread Rattaiah is started");

for (int i=0;i<10 ;i++ )

{

System.out.println("Rattaiah");

}

System.out.println("thread Rattaiah is ended");

}

};

class NagoorThread extends Thread

{

public void run()

{

System.out.println("Enter into thread Nagoor");

System.out.println("thread Nagoor is started");

for (int i=0;i<10 ;i++ )

{

System.out.println("Nagoor");

}

System.out.println("thread Nagoor is ended");

}

};

class RamiReddyThread extends Thread

{

public void run()

{

System.out.println("Enter into thread Ramereddy");

System.out.println("thread RamiReddy is started");

for (int i=0;i<10 ;i++ )

{

System.out.println("RamiReddy");

}

System.out.println("thread RamiReeddy is ended");

}

};

class ThreadDemo

{

public static void main(String[] durga)

{

RattaiahThread thread1=new RattaiahThread();

NagoorThread thread2=new NagoorThread();

RamiReddyThread thread3=new RamiReddyThread();

thread1.setPriority(Thread.MAX\_PRIORITY);

System.out.println(thread1.getPriority());

Thread2.setPriority(Thread.MIN\_PRIORITY);

System.out.println(thread2.getPriority());

Thread3.setPriority(thread2.getPriority()+1);

System.out.println(thread3.getPriority());

System.out.println("starting of Rattaiah Thread");

thread1.start();

System.out.println("starting of Nagoor Thread");

thread2.start();

System.out.println("starting of RamiReddy Thread");

thread3.start();

}

};

**Some of the thread class methods:-**

**Sleep():-**

The sleep() method causes the current thread to sleep for a specified amount of time in milliseconds.

**public static void sleep(long millis) throws InterruptedException**

**public static void sleep(long millis,int nanosec) throws InterruptedException**

For example, the code below puts the thread in sleep state for 5 minutes:

**Ex:-**

class Test

{

public static void main(String[] args)

{

try

{

for (int i=0;i<10 ;i++)

{

System.out.println("Rattaiah");

Thread.sleep(5\*1000);//5 seconds

Thread.sleep(5\*60\*1000);// 5 minits

}

}

catch (InterruptedException ie)

{

System.out.println("the thread is got innterupted");

}

}

}

**Ex :-**

class Test

{

public static void main(String[] args)throws InterruptedException

{

System.out.println("Rattaiah");

Thread.sleep(3\*1000);

}

}

**Java.lang.Thread.yield():-**

Yield() method causes to pause current executing Thread for giving the chance for waiting threads of same priority.

If there are no waiting threads or all threads are having low priority then the same thread will continue its execution once again.

**Syntax:-**

**Public static native void yield();**

**Ex:-**

class MyThread extends Thread

{

public void run()

{

for(int i=0;i<10;i++)

{

Thread.yield();

System.out.println("child thread");

}

}

}

class ThreadYieldDemo

{

public static void main(String[] args)

{

MyThread t1=new MyThread();

t1.start();

for(int i=0;i<10;i++)

{

System.out.println("main thread");

}

}

}

**Java.lang.Thread.join():-**

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

1. Public final void join()throws InterruptedExcetion

2. Public final void join(long ms)throws InterruptedException

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

**Ex:-**

class MyThread extends Thread

{

public void run()

{

for (int i=0;i<5;i++ )

{

try{

System.out.println("rattaiah");

Thread.sleep(3\*1000);}

catch(InterruptedException iee)

{

System.out.println("gettting innterupted exctpion");

}

}

}

}

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

try

{

t1.join();

}

catch (InterruptedException ie)

{

System.out.println("interrupted Exception");

}

t2.start();

}

};

**Ex 2:-**

class MyThread extends Thread

{

public void run()

{

for (int i=0;i<5;i++ )

{

Try

{

System.out.println("rattaiah");

Thread.sleep(3\*1000);}

catch(InterruptedException ie)

{

System.out.println("getting excception");

}

}

}

}

class ThreadDemo

{

public static void main(String[] args)throws InterruptedException

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t1.join();

t2.start();

}

};

**isAlive():-**

used to check whether the thread is live or not.

Public Boolean isAlive()

class MyThread extends Thread

{

public void run()

{

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

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t=new MyThread();

System.out.println(t.isAlive());

t.start();

System.out.println(t.isAlive());

}

};

**Java.lang.Thread.activeCount():-**

This method is used to find out the number of methods in active state.

Public static int activeCount();

**Ex:-**

class MyThread extends Thread

{

public void run()

{

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

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

MyThread t3=new MyThread();

t1.start();

t2.start();

t3.start();

System.out.println(Thread.activeCount());//4

}

};

**Java.lang.currentThread():-**

This method is used to represent current thread class object.

Public static thread currentThread()

**Ex:-**

class MyThread extends Thread

{

public void run()

{

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

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

}

};

**Java.lang.Thread.getId():-**

getId() is used to generate id value for each and every thread.

**Public long getId()**

**Ex:-**

class MyThread extends Thread

{

public void run()

{

System.out.println(" rattaiah thread is running ");

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

System.out.println("the thread id :"+t1.getId());

System.out.println("the thread name is :"+t1.getName());

System.out.println("the thread priority is "+t1.getPriority());

System.out.println("the thread id :"+t2.getId());

System.out.println("the thread name is :"+t2.getName());

System.out.println("the thread priority is "+t2.getPriority());

}

};

**Interrupted():-**

A thread can interrupt another sleeping or waiting thread.

For this Thread class defines interrupt() method.

**Public void interrupt()**

**Effect of interrupt() method call:-**

class MyThread extends Thread

{

public void run()

{

try

{

for (int i=0;i<10;i++ )

{

System.out.println("i am sleeping ");

Thread.sleep(5000);

}

}

catch (InterruptedException ie)

{

System.out.println("i got interupted by interrupt() call");

}

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t=new MyThread();

t.start();

t.interrupt();

}

};

**No effect of interrupt() call:-**

class MyThread extends Thread

{

public void run()

{

for (int i=0;i<10;i++ )

{

System.out.println("i am sleeping ");

}

}

};

class ThreadDemo

{

public static void main(String[] args)

{

MyThread t=new MyThread();

t.start();

t.interrupt();

}

};

**NOTE:-**

The interrupt() is working good whenever our thread enters into waiting state or sleeping state.

The interrupted call will be wasted if our thread doesn’t enters into the waiting/sleeping state.

class MyThread extends Thread

{

public void run()

{

for (int i=0;i<10;i++)

{

Thread.sleep(2000);

System.out.println("durgasoft task");

}

}

};

class ThreadDemo

{

public static void main(String[] args)throws Exception

{

MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

t1.start();

t2.start();

t3.start();//4-threads

t1.join();

System.out.println(t1.getName());//thread-0

System.out.println(t2.getName());

System.out.println(t3.getName());

t1.setName("sneha");

System.out.println(t1.getName());//sneha

System.out.println(Thread.currentThread().getName());//main

Thread.currentThread().setName("poornima");

System.out.println(Thread.currentThread().getName());//poornima

System.out.println(Thread.activeCount());

System.out.println(t1.isAlive());

System.out.println(t1.getId());

System.out.println(t2.getId());

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

System.out.println(t1.getPriority());

Thread.currentThread().setPriority(10);

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

for (int i=0;i<5;i++)

{

Thread.sleep(5000);

Thread.yield();

System.out.println("main thread");

}

}

};

**Synchronized :-**

Synchronized modifier is the modifier applicable for methods but not for classes and variables.

If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.

The main advantage of synchronized modifier is we can resolve data inconsistency problems.

But the main disadvantage of synchronized modifier is it increases the waiting time of the Thread and effects performance of the system .Hence if there is no specific requirement it is never recommended to use.

The main purpose of this modifier is to reduce the data inconsistence problems.

**Non-synchronized methods**

void m1()

{ non-synchronized method any number of threads can access

}

Every thread accessing simultaneously

Thread 1 Thread 2 Thread 3---------Thread N

1) In the above case multiple threads are accessing the same methods hence we are getting data inconsistency problems. These methods are not thread safe methods.

2) But in this case multiple threads are executing so the performance of the application will be increased.

**Synchronized methods**

synchronized void m2()

{

Synchronized method only one thread is allow to access.

}

Only one thread allowed to access

Thread 1 Thread 2 Thread 3---------Thread N

1) In the above case only one thread is allow to operate on particular method so the data inconsistency problems will be reduced.

2) Only one thread is allowed to access so the performance of the application will be reduced.

3) If we are using above approach there is no multithreading concept.

Hence it is not recommended to use the synchronized modifier in the multithreading programming.

**Daemon threads:-**

The threads wchich are executed at background is called daemon threads.

Ex:- garbage collector,ThreadSchedular.default exceptional handler.

**Non-daemon threads:-**

The threads which are executed fore ground is called non-daemon threads.

Ex:- normal java application.

**Volatile:-**

Volatile modifier is also applicable only for variables but not for methods and classes.

If the values of a variable keep on changing such type of variables we have to declare with volatile modifier.

If a variable declared as a volatile then for every Thread a separate local copy will be created.

Every intermediate modification performed by that Thread will take place in local copy instead of master copy.

Once the value got finalized just before terminating the Thread the master copy value will be updated with the local stable value. The main advantage of volatile modifier is we can resolve the data inconsistency problem.

But the main disadvantage is creating and maintaining a separate copy for every Thread

Increases the complexity of the programming and effects performance of the system