**-: Multithreading :-**

1. Introduction
2. Ways to define a thread
3. By Extending Thread class
4. By implementing Runnable Interface
5. Getting and stting name of thread
6. Thread Priority
7. Method to prevent Thread Execution
8. yield()
9. join()
10. sleep()
11. Synchronization
12. Interthread Communication
13. Deadlock
14. Demon Threads
15. Multithreading enhancements

**Introduction:**

Multitasking : Executing several task simultaneously is the concept of multitasking. There are two types of multitasking. 1. Process based multitasking

2. Thread based multitasking

**Process based multitasking:**

Executing several tasks simultaneously where each task is a separate independent program (process) is called process based multitasking.

Ex:

While typing a java program in the editor we can listen music from that system at the same time we can download a file from net. All these task will be executed simultaneously and independent of each other hence it is process based multitasking.

Process based multitasking is best suitable at OS level.

**Thread based multitasking :**

Executing several task simultaneously where each task is a separate independent part of same program is called thread based multitasking and independent part is called a Thread.

Thread based multitasking is best suitable at programmatic level.

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

The main important areas of multithreading are to develop multimedia graphic, animation, videogames, to develop web servers and application servers etc.

**Defining a Thread:**

We can define a thread in the following two ways

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

**By extending Thread Class:**

Ex:

class myThread extends Thread{

public void run(){ // Code within run() is called **job of Thread**

for(int i=0;i<5;i++){ // This code will executed by child Thread

Sop(“Child Thread”);

}

}

}

class TestThread{  
 public static void main(String [] args){

myThread t = new myThread(); // Instantiation of Thread

t.start(); // Starting of thread

for(int i=0;i<5;i++){ //Executed by main thread

Sop(“Main Thread”);

}

}

}

**Case 1: Thread Scheduler**

It is the part of JVM. It is responsible to schedule threads i.e. if multiple threads are waiting for execution then in which order the threads will be executed is decided by thread scheduler.

We can’t except exact algorithm followed by scheduler it is valid from JVM to JVM. Hence we can’t except thread execution order and exact output.

Hence whenever situation comes to multithreading there is no guaranty of exact output, but we can several possible outputs.

There will be mixed output.

**Case 2: Difference between t.start() and t.run()**

In the case of t.start() a new thread will be created which is responsible for execution of run method.

But in the case of t.run() a new thread won’t be created and run method will be executed just like a normal method call by main thread.

In the above problem if we will change the t.start() with t.run() then the output is

O/P:-

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread

This total output produced by only main thread.

**Case 3: importance of Thread class start()**

Thread class start() is responsible to register the thread with thread scheduler and all other activities.

Hence without executing thread class start() method there is no chance of starting a thread in java.

Due to this thread class start() method is considered as heart of multithreading.

start() {

Register this thread with thread scheduler

Perform all other mandatory operation

Invoke run();

}

**Case 4: overloading of run() method**

Overloading of run() method is always possible but thread class start() method can invoke no argument run method. The other overloaded method we have to call explicitly like a normal method call.

Ex:

class myThread extends Thread{

public void run(){

Sop(“No-arg run method”);

}

public void run(int i){

Sop(“int-arg run method”);

}

}

class TestThread{

public static void main(String [] args){

myThread t = new myThread();

t.start();

}

}

O/P: No-arg run method

**Case 5: if we are not overriding run()**

If we are not overriding run() then thread class run() method will be executed which has empty implementation. Hence we won’t get any output.

EX;-

class myThread extends Thread{

}

class Test{

public static void main(String [] args ){

myThread t = new Mythread();

t.start();

}

}

No output

**Note:** it is highly recommended to override run() otherwise don’t go for multithreading concept.

**Case 6: Overriding of start() method**

If we are overriding the start() then our start() will be executed just like normal method call and a new thread won’t be created.

Ex:

class myThread extends Thread{

public void start(){

Sop(“Start Method”);

}

public void run(){

Sop(“Run Method”);

}

}

class Test{

public static void main(String [] args ){

myThread t = new Mythread();

t.start();

Sop(“Main Thread”);

}

}

O/P: Start Method // Output produced by only main Thread

Main Method

**Note:** It is not recommended to override start() method otherwise don’t go for multithreading concept.

**Case 7:**

Ex:

class myThread extends Thread{

public void start(){

super.start(); // Thread class start() is invoked

Sop(“Start Method”);

}

public void run(){

Sop(“Run Method”);

}

}

class Test{

public static void main(String [] args ){

myThread t = new Mythread();

t.start();

Sop(“Main Thread”);

}

}

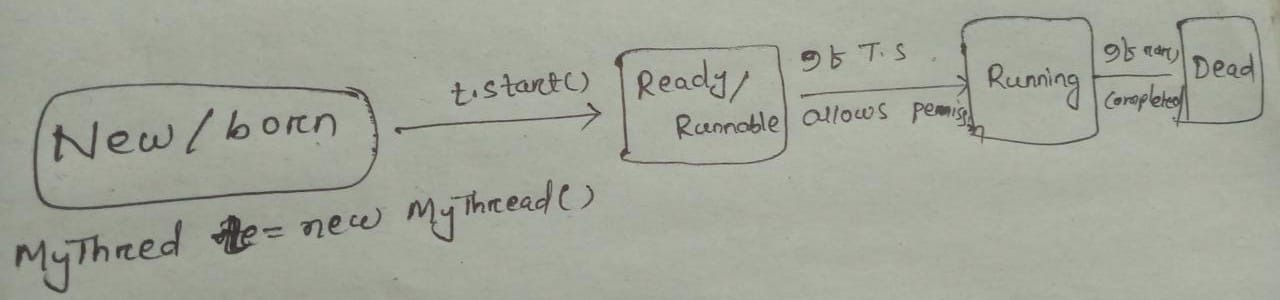
o/p: mixed outputs of

Run Method

Start Method

Main Method

**Case 8: Thread Life cycle**



**Case 9: IllegalThreadStateException**

After starting a thread if we are trying to restart the same thread then we will get runtime exception saying IllegalThreadStateEXception.

Ex:

Thread t = new Thread();

t.start();

..

t.start(); //RE: IllegalThreadStateEXception

**Defining a thread by Implementing Runnable Interface:**

We can define a thread by implementing Runnable interface. Runnable interface present in java .lang package and it contains only one method i.e. run() method.

Ex:

class MyRunnable implements Runnable{

public void run(){ // Job of a Thread

for(int i=0;i<5;i++){ // Executed by child thread

Sop(“Child Thread”);

}

}

}

Class Test{

Public static void main(String [ ] args){

MyRunnable r = new MyRunnable();

Thread t = new Thread(r); // r = Target Runnable

t.start();

for(int i=0;i<5;i++){ // Executed by main Thread

Sop(“Main Thread”);

}

}

}

O/P: we will get mixed output and we can’t tell exact output

**Case Study**

MyRunnable r = new MyRunnable();

Thread t1 = new Thread();

Thread t2 = new Thread(r);

**Case 1: t1.start()**

A new thread will be created and which is responsible for execution of thread class run() method, which has empty implementation.

**Case 2: t1.run()**

No new thread will be created and thread class run method will be created just like a normal method call.

**Case 3: t2.start()**

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

**Case 4: t2.run()**

A new thread won’t be created and MyRunnable run() method will be executed just like a normal method call.

**Case 5: r.start()**

We will get compile time error saying: MyRunnable class doesn’t have start capability

CE: can’t find symbol, location class MyRunnable, symbol method start

**Case 6: r.run()**

No new thread will be created and MyRunnable run() method will be executed like normal method call.

Among two ways of defining a thread implements Runnable approach is recommended.

In first approach our class always extends thread class, there is no chance of extending any other class hence we are missing inheritance benefits.

But in the second approach while implementing Runnable interface we can extend any other class hence we won’t miss any inheritance benefits.

**Thread class Constructors:**

1. Thread t = new Thread ();
2. Thread t = new Thread (Runnable r);
3. Thread t = new Thread (String name);
4. Thread t = new Thread (Runnable r, String name);
5. Thread t = new Thread (ThreadGroup g, String name);
6. Thread t = new Thread (ThreadGroup g, Runnable r);
7. Thread t = new Thread (ThreadGroup g, Runnable r, String name);
8. Thread t = new Thread (ThreradGroup g, Runnable r, String name, long stack\_size);

**Durga’s Approach to define Thread (**Not recommended to use**)**

class MyThread = new Thread{

public void run(){

Sop(“child thread”);

}

}

class ThreadDemo{

public static void main(String [ ] args){

MyThread t = new MyThread();

Thread t1 = new Thread(t);

t1.start();

Sop(“main thread”);

}

}

O/p:- child thread or main thread

main thread child thread

**Getting and setting name of a thread :**

Every thread in java has some name it may be default name generated by JVM or customized name given by programmer.

We can get and set name of a thread by using the following two methods of Thread class

* public final String getName()
* public final void setName(String name)

Ex:

class MyThread extends Thread{

}

class Thread{

public static void main(String [] args){

Sop(Thread.currenThread().getName()); // main

MyThread t = new MyThread();

Sop(t.getName); //Thread-0

Thread.currentThread().setName(“Himansu Patra”);

Sop(Thread.currenThread().getName()); //Himansu Patra

}

}

**Note:** We can get current executing thread object by Thread.currentThread().

Ex:

class MyThread extends Thread{

public void run(){

Sop(“run method is executed by : ” + Thread.currentThread().getName());

}

}

class Thread{

public static void main(String [] args){

MyThread t = new MyThread();

t.start();

Sop(“main method is executed by : ” + Thread.currentThread().getName());

}

}

**Thread Priority:**

Every thread in java has some priority. It may be default priority generated by JVM or customized priority given by programmer.

The valid range of Thread priority is 1 to 10. Where 1 is min priority and 10 is max priority.

Every Thread class defines the following constants to represent some standard priority.

* Thread.MIN\_PRIORITY 🡪 1
* Thread.NORM\_PRIORITY 🡪 5
* Thread.MAX\_PRIORITY 🡪 10

Thread scheduler will use Thread priorities while allocating processor. The thread having highest priority will get first chance.

If two threads having same priority then we can’t execution order it depends on thread scheduler.

Thread class defines the following method to get set priority of a thread

* public final int getPriority()
* public final void setPriority()

Ex:

t.setPriority(17); //valid range 1-10 RE: IllegalArgumentException

**Default Priority:**

The default priority only for main thread is 5 but for all remaining threads default priority will be inherited from parent to child i.e. whatever the priority parent thread has the same priority will be there for the child thread.

Ex:

class MyThread extends Thread{

}

class Thread{

public static void main(String [] args){

Sop(Thread.currentThread().getPrority()); //5

Thread.currenThread().setPririty(7);

MyThread t = new MyThread();

Sop(t.getPriority()); //7

}

}

class MyThread extends Thread{

public void run(){

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

Sop(“Child Thread”);

}

}

}

class Thread{

t public static void main(String [] args){

MyThread t = new MyThread();

t.setPriority(10);

t.start();

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

Sop(“Main Thread”);

}

}

}

O/P: Child Thread 5 times then Main Thread 5 times

**Preventing thread execution :**

We can prevent thread execution by using the following methods.

1. yield ()
2. join ()
3. sleep ()

**yield ():**

yield () method causes to pause current executing thread to give the chance for waiting thread of same priority. If there is no waiting thread or all waiting thread have low priority then same thread can continue its execution.

If multiple threads are waiting with same priority then which waiting will get the chance we can’t except in depends on thread scheduler.

The thread which is yielded, when it will get the chance to start again it depends on thread scheduler and we can’t except exactly.

Prototype: **public static native void yield ();**

In the life cycle of a thread when we call yield method the cycle is going from running state to ready/runnable class.

Ex:

class MyThread extends Thread {

public void run (){

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

Sop(“Child Thread”);

Thread.yield(); // Line - 1

}

}

}

class ThreadDemo{

public static void main(String [] args){

MyThread t = new MyThread();

t.start();

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

Sop(“Main Thread”);

}

}

}

In the above program if we are commenting line 1 then both thread will be executed simultaneously and we can’t except which will complete first.

If we are not commenting line 1 then child thread always call yield method because of that main thread will get chance more number of times and the chance of completing main thread class is high.

**Join ():**

If a thread wants to until completing some other thread then we should go for join method. Ex:

If a thread t1 wants to wait until t2 got completed then t1 has to call t2.join();

If t1 executes t2.join() then immediately t1 will be entered into waiting state until t2 completes. Once t2 completes then t1 can continue its execution.

Prototype: **public final void join () throws InterruptedException**

**public final void join (long milliseconds) throws InterruptedException**

**public final void join (long milliseconds, int nanoseconds) throws InterruptedException**

Note: Every join method throws InterruptedException which is checked Exception. Hence compulsory we should handle this exception either by using try catch or by throws keyword. Otherwise we will get compile time error.

Ex: waiting of main thread until completing child thread

class MyThread extends Thread {

public void run (){

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

Sop(“Child Thread”);

try{

Thread.sleep(1000);

}catch(InterruptedException e){

}

}

}

}

class ThreadDemo{

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

MyThread t = new MyThread();

t.start();

t.join(); // Line-1

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

Sop(“Main Thread”);

}

}

}

If we comment line-1 then both main and child thread will be executed simultaneously and we can’t except exact output.

If we are not commenting line-1 then main thread calls join method on child thread object hence main thread will wait until completing child thread in this case output is

Child Thread

Child Thread

Child Thread

Child Thread

Child Thread

Main Thread

Main Thread

Main Thread

Main Thread

Main Thread