**1. Multi Threading Introduction.**

**2. Custom Threads.**

**3. Thread Life Cycle.**

**4. Thread Priorities.**

**5. Thread Names.**

**6. Getting Thread Object Reference.**

**7. Thread group.**

**8. Thread Types.**

**9. Controlling Thread Executions.**

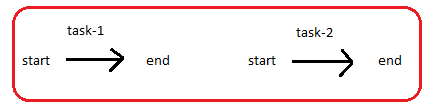
**10. Inter Thread Communication-1**

**11. Inter Thread Communication-2**

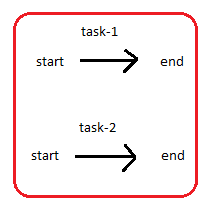
**MULTI THREAD INTRODUCTION**

1. **Different ways of executing tasks:-** we have different flows of executing tasks.
2. Sequential flow of execution.
3. Parallel flow of execution.
4. Concurrent flow of execution.

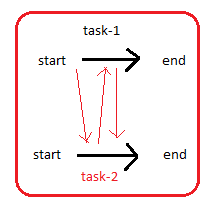
**1.1.Sequential flow of execution:-**In sequential flow of execution, multiple tasks are executed one after one. It means one task execution is executed only after previous task is completed fully.



**1.2.Parallel flow execution:-** Two or more tasks are executed at a time with out depending on other task.



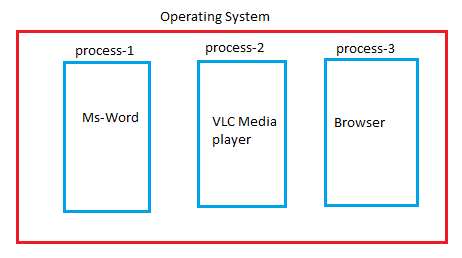
**1.3.Concurrent flow of execution:-** In concurrent flow of execution, two or more multiple tasks are executed simultaneously (at a time). That means, one task execution is started only when currently running task execution is paused.



**2.MultiTasking:-** Executing the multiple-tasks at a time is called multi tasking.

Example:-

While file is being downloaded, we type in ms-word and at the same time we listen song. It is multi tasking.



**3.Different Types of multi-tasking**:- There are two types of multi-tasking.

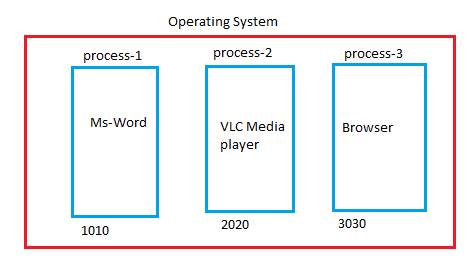
3.1. process based multi-tasking.

3.2. Thread based multi-tasking.

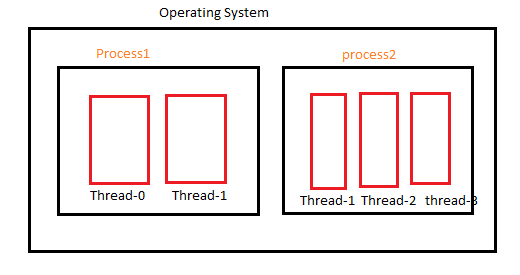
**3.1. process based multi-tasking:-** Executing the multiple-tasks at a time is called multi tasking. The OS allocates separate process to every task.Each task is executed by separate independent process. Each process has separate address space. This type of multi tasking is implemented at OS level.

Example:-

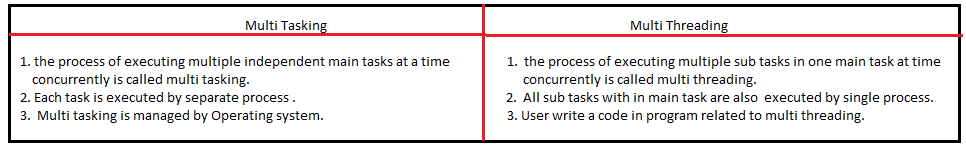
While file is being downloaded, we type in ms-word and at the same time we listen song. It is multi tasking.



**3.2 Thread based multitasking:-** The main task may consists of several sub tasks. The main task is technically called as process and the each sub task is technically called as thread. Executing multiple sub taks with in main task simultaneously is called **thread based multitasking or multi threading.**

****

**4. Difference between multitasking and multi threading.**

****

**5. Over view of Java threads:**

* The thread is an independent sequential flow of execuiton path.
* When JVM instance is created, Inside JVM by default two threads are created.

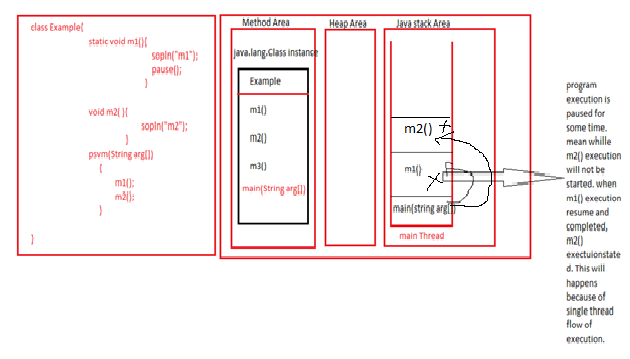
1. Main thread.
2. Garbage collector thread.

The static blocks ,blocks,constructors,static methods and non-static methods are executed in main thread by providing separate memory block in it. The memory block is technically called stack frame. Once their execution is completed then stack frame is destroyed.

This execution flow is called single thread flow execution because all are only executed in single thread.

**Drawback of single thread flow execution**:- If one task execution is paused, then this paused time will not be used for executing another task. Here execution time is wasted. Due to this program execution takes more time.

**Example**:



**CUSTOM THREADS**

**1.Custom Thread:-** The thread which is created by programmer is called custom thread. As many independent tasks we want to execute concurrently, we must create those many new custom threads.

In Java language we have in-built API for developing new threads. This API consists of one interface and 3 classes to create and work with threads. They are

1. Runnable Interface.
2. Thread class.
3. ThreadGroup class.
4. ThreadLocal<T> class.

**2. Thread class:-** creating thread extending from Thread class is 3 steps process.

a. Create class which extends Thread class.

b. Overriding run() method and place the logic that we want to execute from this custom thread.

c. create this subclass object and invoke start() method using this subclass object.

Example:-1

classMyThread extends Thread

{

public void run(){

System.out.println("This is customThread");

}

}

class sample {

public static void main(String arg[]){

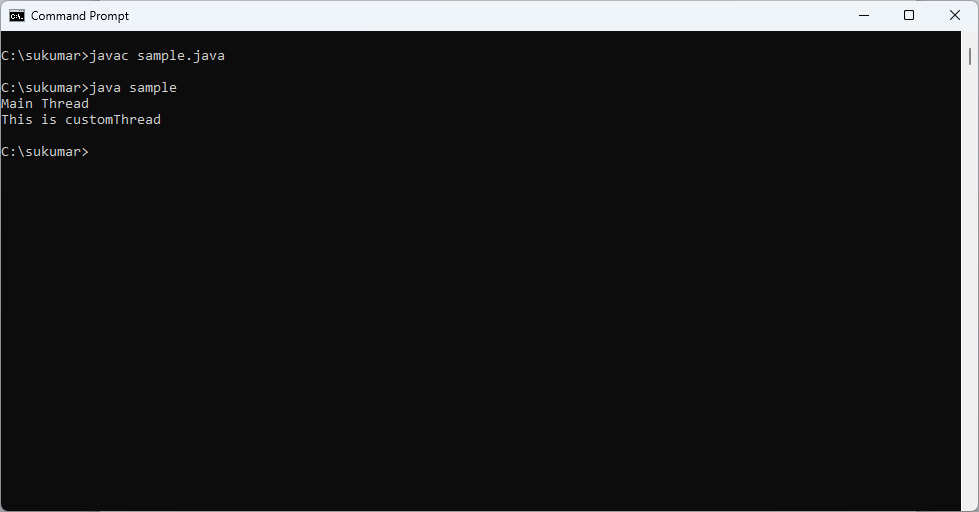
MyThread m1=new MyThread();

System.out.println("Main Thread");

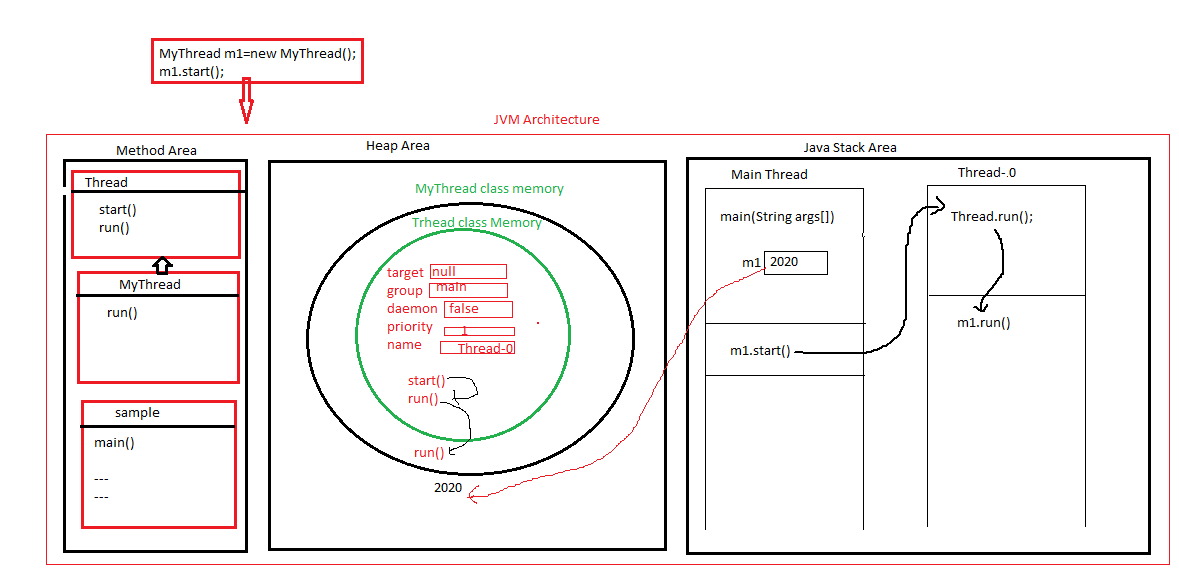
m1.start();

}

}



JVM architecture diagram with Thread Execution flow:



Example:2

class sample extends Thread{

public void run(){

System.out.println("This is customThread");

}

public static void main(String arg[]){

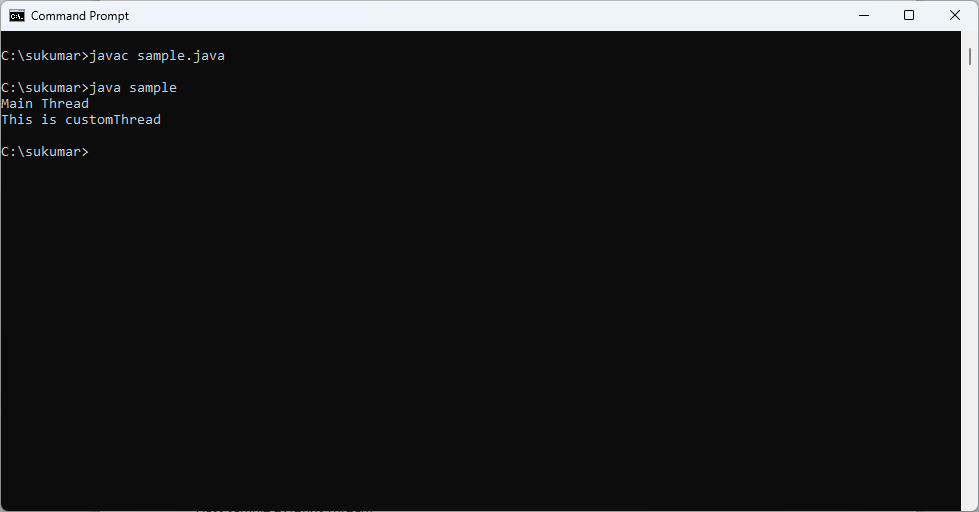
sample m1=new sample();

System.out.println("Main Thread");

m1.start();

}

}



**3. Runnable Interface:-**Creating custom thread implementing from Runnable interface is four steps process:

1. Create class which implements runnable interface.
2. Implement run() method and place the logic that we want to execute from this custom thread.
3. Create this subclass object.
4. Create Thread class object explicitly by passing this runnable subclass object as argument, then invoke start() method using Thread class object.

Example:-

classMyThread implements Runnable{

public void run(){

System.out.println("This is customThread");

}

}

class sample {

public static void main(String arg[]){

MyThread m1=new MyThread();

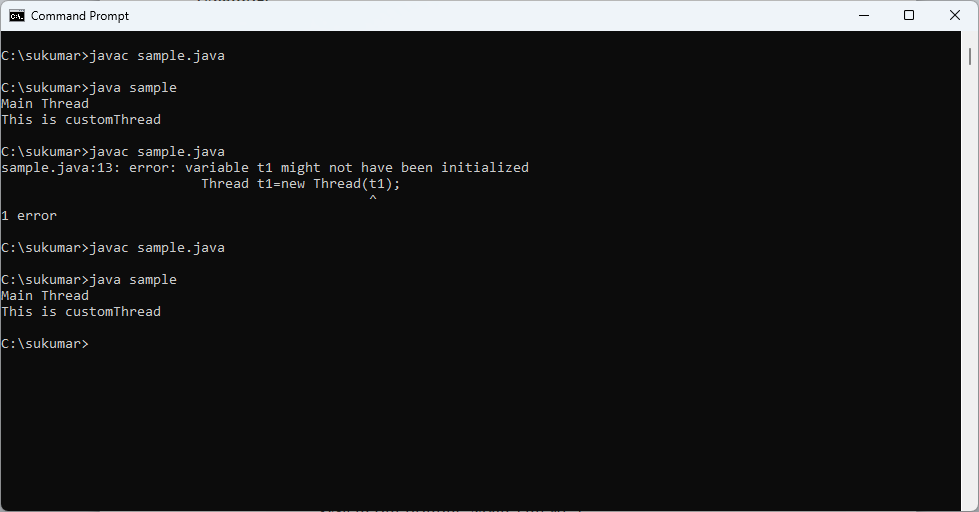
Thread t1=new Thread(m1);

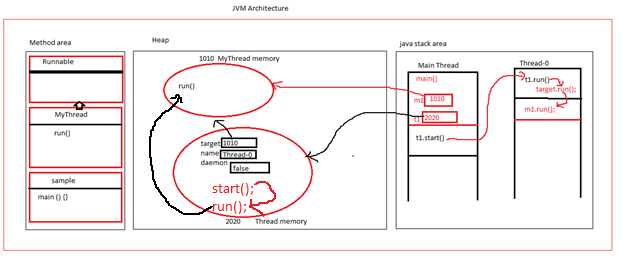
System.out.println("Main Thread");

t1.start();

}

}





1.The application shows creating multiple custom threads to execute same logic from multiple threads.

classMyThread extends Thread{

public void run(){

for(int i=1;i<5;i++){

System.out.println(getName()+"Run:"+i);

}

}

}

class sample {

public static void main(String arg[]){

MyThread m1=new MyThread();

MyThread m2=new MyThread();

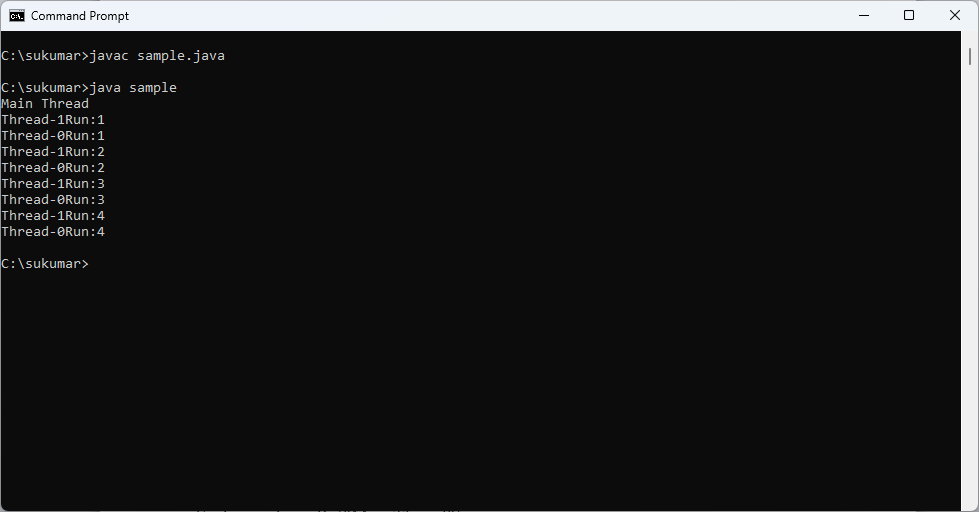
System.out.println("Main Thread");

m1.start();

m2.start();

}

}



2.The application shows creating multiple custom threads to execute same logic from multiple threads with different inputs.

classMyThread extends Thread{

publicint max;

publicMyThread(int max){

this.max=max;

}

public void run(){

for(int i=1;i<max;i++){

System.out.println(getName()+"Run:"+i);

}

}

}

class sample {

public static void main(String arg[]){

MyThread m1=new MyThread(5);

MyThread m2=new MyThread(10);

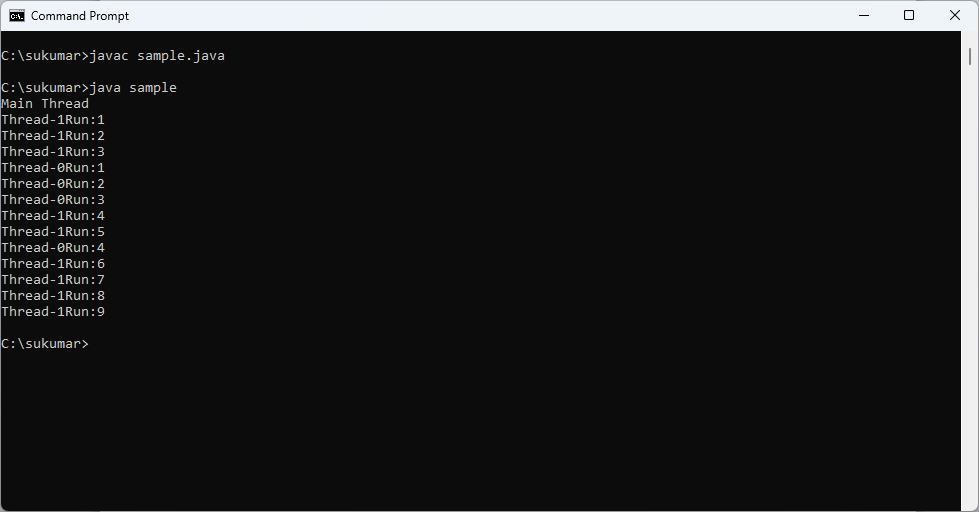
System.out.println("Main Thread");

m1.start();

m2.start();

}

}



3. The application shows creating multiple custom threads to execute different logic from multiple threads.

classMyThread extends Thread{

publicint max;

publicMyThread(int max){

this.max=max;

}

public void run(){

for(int i=1;i<max;i++){

System.out.println(getName()+"Run:"+i);

}

}

}

class MyThread1 extends Thread{

publicint max;

public MyThread1(int max){

this.max=max;

}

public void run(){

for(int i=max;i>1;i--){

System.out.println(getName()+"Run:"+i);

}

}

}

class sample {

public static void main(String arg[]){

MyThread m1=new MyThread(5);

MyThread1 m2=new MyThread1(10);

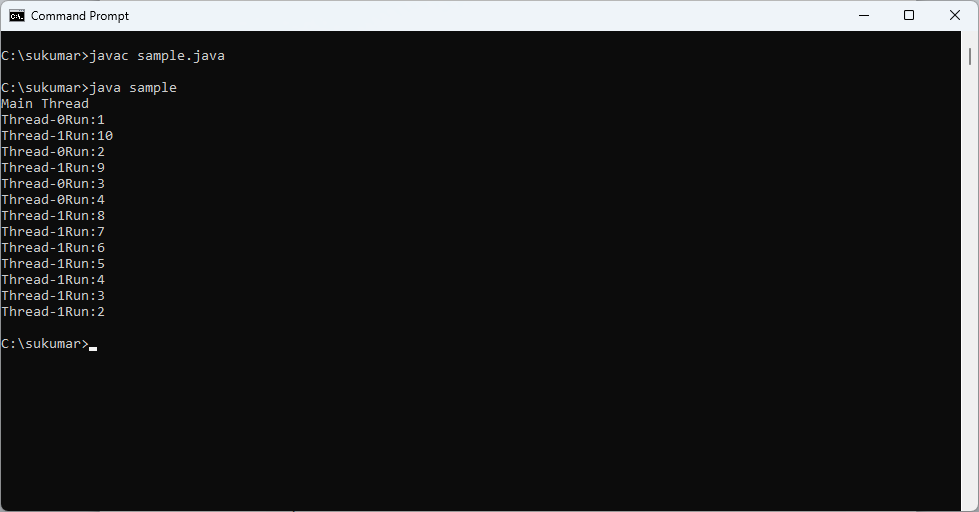
System.out.println("Main Thread");

m1.start();

m2.start();

}

}



**THREAD LIFE CYCLE**

In Java Thread is always in one of the following states.

1. New
2. Ready-to-run
3. Running.
4. Blocked
5. Waiting
6. Timed Waiting.
7. Terminating.

**1.new**:- whenever a new thread is created, It is always in new state.

2.when thread invokes start() method, It moves from new state to one of the followng state.

**2.1. Ready-to-run state/Runnable state:**- The start() method call is just request to JVM to start thread exectuion for this thread object. Then when processor is busy, the thread enters the Ready-to-run state.

**2.2. Running state:-** The start() method call is just request to JVM to start thread exectuion for this thread object. Then when processor is free, JVM calls the run method of this thread and executed its logic. That means thread gets CPU and it enters running state from ready-to-run state.

**3. Blocked:-** The thread enters this state when it is waiting for monitor lock and is trying to access a section of code that is locked by other thread.

(or)

A thread(a) may want to print some data from the printer. However, at the same time, the other thread(B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus thread A is in the blocked state.

When resource is free, the thread moves from blocked state to **“ready-to-run”** state.

**4. waiting**:- when the thread invokes the join() method then, it is said that the thread is in waiting state. The thread then waits for the child threads to complete their tasks. When the child thread complete their job, a notification is sent to main thread, which again moves the thread from waiting to “**ready-to-run”** state.

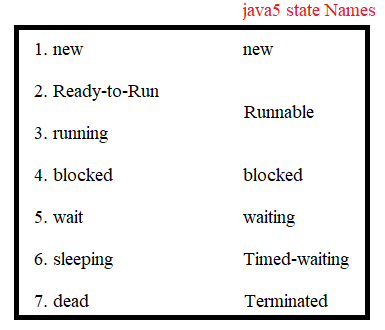
**5.sleeping**:- when we invoke the sleep() method on specific thread, the sleep() method puts the thread in timed-waiting state. After specified-time completed , the thread moves from timed-waiting state to **“ready-to-run”** state.

**6. Dead:**- The thread reaches terminated state because of following reasons.

i.when thread has finished its job, then it terminates normally.

ii.when some unusual events such as “un handled exception” happens , then thread terminates abnormally.

Java5 onwards, thread states names have been changed.



Thread Life Cycle Diagram:

Example:

public class Test14 {

public static void main(String[] args) { //Old states Java 5v

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

MyThread7 mt = new MyThread7();//thread object is created - 1.New

System.out.println("in main #1: "+ mt.getState());

mt.start(); //thread is ready to execute - 2. Ready-To-Run RUNNABLE

System.out.println("In main #2: "+ mt.getState());

try {Thread.sleep(2000);}

catch (InterruptedException e) {}

System.out.println("In main #4: "+ mt.getState());

try {Thread.sleep(7000);}

catch (InterruptedException e) {}

System.out.println("In main #5: "+ mt.getState());

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

}

}

class MyThread7 extends Thread {

@Override

public void run() { //thread is running 3. Running RUNNABLE

System.out.println("run start");

System.out.println("In run #3: "+ this.getState());

try {Thread.sleep(5000);} //thread is blocked 4. Blocked TIMED\_WAITING

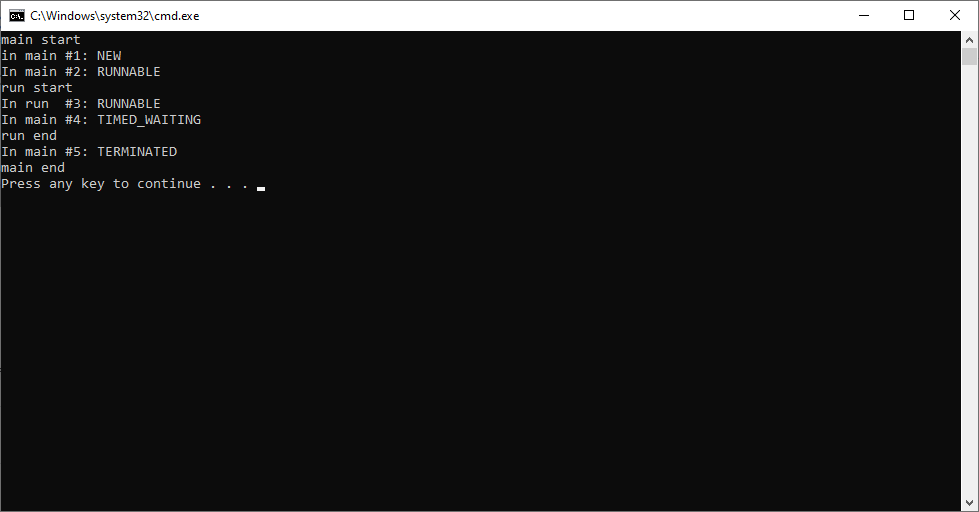
catch(InterruptedException e) { } //WAITING

//BLOCKED

System.out.println("run end");

} //thread is dead 5. Dead TERMINATED

}



**THREAD PRIORITIES**

Schedulers in JVM implementations usually employ one of the two following strategies:

1. **Preemptive Scheduling:-** If a thread with higher priotity than the current running thread comes then higher priotity thread moves to the Ready-to-run state and current running thread can be preeempted(moved to the Ready-rto-run state) to let the higher prioity thread execute.
2. **Time-Sliced or Round-robin Scheduling:-** A running thread is allowed to exeute for a fixed length of time , after which it moves to Ready-to-run state to wait its turn to run again.

Note:-The JVM executes Threads based on their priority and scheduling.

Every thread is created in JVM with a priority. The priority range is between 1 and 10.

1 is called minimum priority.

5 is called normal priority.

10 is called maximum priority.

Every thread inherits the priority from its parent thread. The default priority of every thread is normal priority 5 because main thread priority is 5.

In Thread class have below 3 variables are defined to represent above 3 values & methods:

1. Public static final int MIN\_PRIORITY
2. Public static final int NORM\_PRIORITY
3. Public static final int MAX\_PRIORITY

The priority of a thread can be set using setPriority() method .

Syntax:

Public final void setPrority(int priority)*throws* ***illegalArgumentException*;**

If priority is less than 1 and greater than 10 , It leads to exception.

**Java.lang.illegalArgumentException**.

The priority of thread can be read using getPriority() method.

Syntax:

Public final int getPriority();

Example:

public class Test14 {

public static void main(String[] args) {

System.out.println("Main Thread Priority:" +Thread. currentThread(). getPriority());

MyThread7 mt = new MyThread7();

mt.start();

}

}

class MyThread7 extends Thread {

@Override

public void run() {

System.out.println("Child Thread");

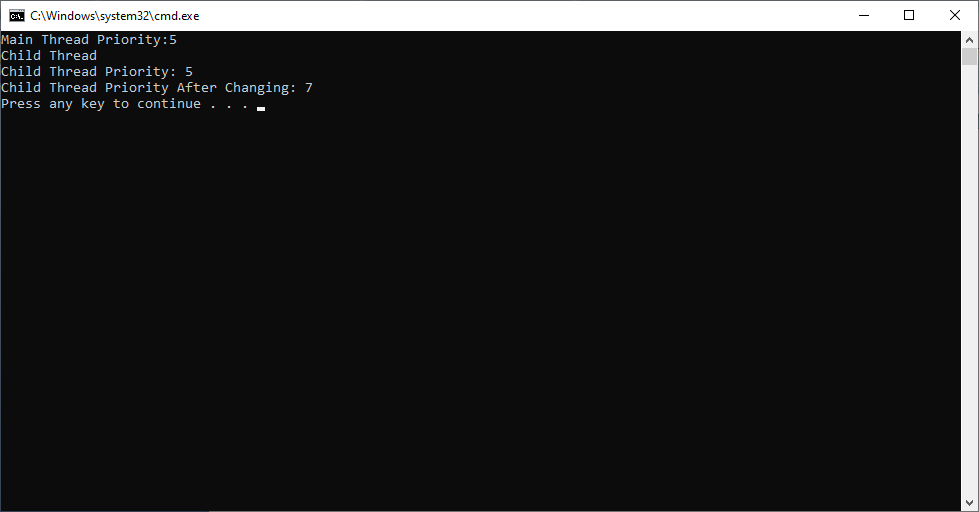
System.out.println("Child Thread Priority: "+ this.getPriority());

this.setPriority(7);

System.out.println("Child Thread Priority After Changing: "+ this.getPriority());

}

}



Note:- windows os follows round robin scheduling . so that we got above output.

Example:2

public class Test14 {

public static void main(String[] args) {

System.out.println("Main Thread Priority:"+Thread.currentThread().getPriority());

MyThread7 mt = new MyThread7();

MyThread7 mt1 = new MyThread7();

mt.setPriority(5);

mt1.setPriority(7);

mt.start();

mt1.start();

System.out.println(" Main End");

}

}

class MyThread7 extends Thread {

@Override

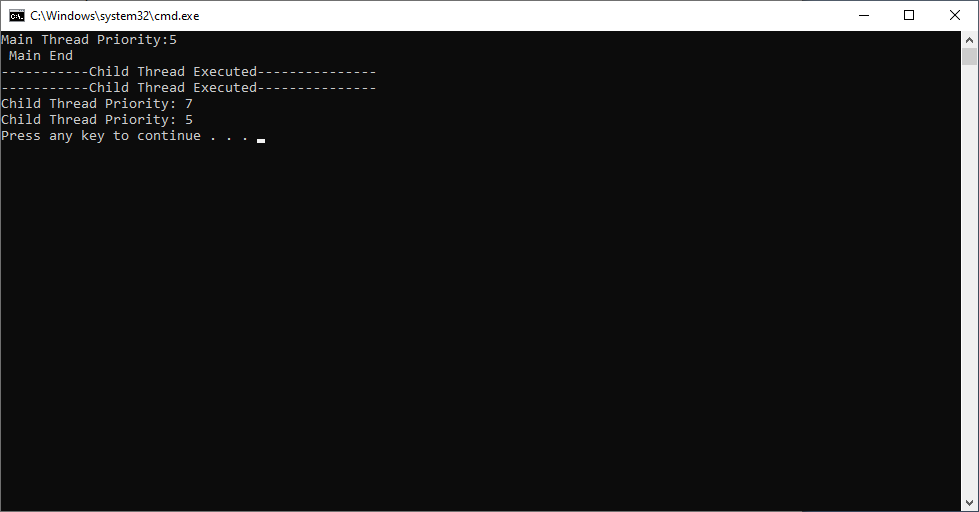
public void run() {

System.out.println("-----------Child Thread Executed---------------");

System.out.println("Child Thread Priority: "+ this.getPriority());

}

}



Example:3

public class Test14 {

public static void main(String[] args) {

Thread.currentThread().setPriority(9);

System.out.println("Main Thread Priority:"+Thread.currentThread().getPriority());

MyThread7 mt = new MyThread7();

mt.start();

}

}

class MyThread7 extends Thread {

@Override

public void run() {

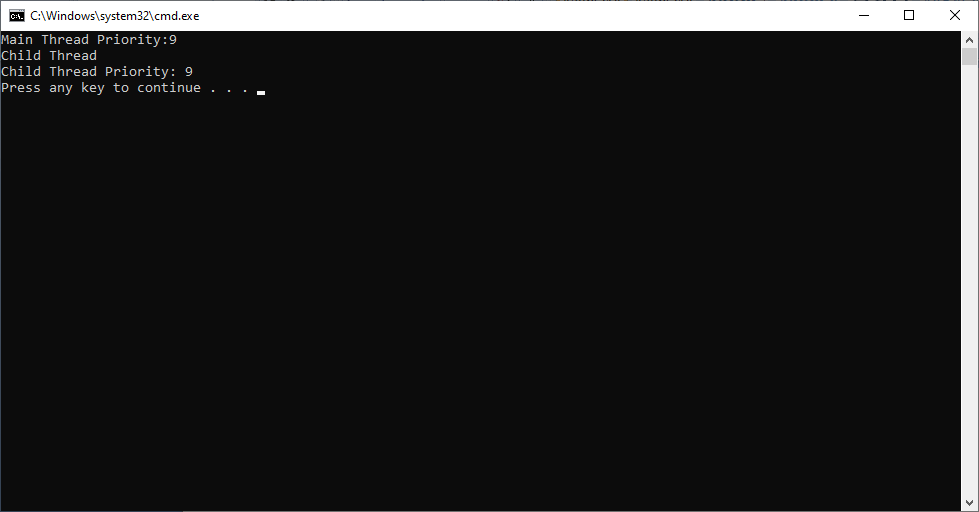
System.out.println("Child Thread");

System.out.println("Child Thread Priority: "+ this.getPriority());

}

}

Output:



**THREAD NAMES**

User Defined thread is created with the default name “Thread”+<index>, where index is the integer number starts with 0. So the first user defined thread name will be Thread-0,second thread name will be Thread-1 , etc …

The default thread name ca be changed by using either

1. Using String parameterized constructor at time of thread object creation.

Syntax:

**Thread(String thread-name)**

1. Using setName method after thread object creation.

Syntax:

public final void setName(String name)

We use following method to get the name of thread in which run method is being executed.

Syntax:

Public final string getName()

Example:

public class Test14 {

public static void main(String[] args) {

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

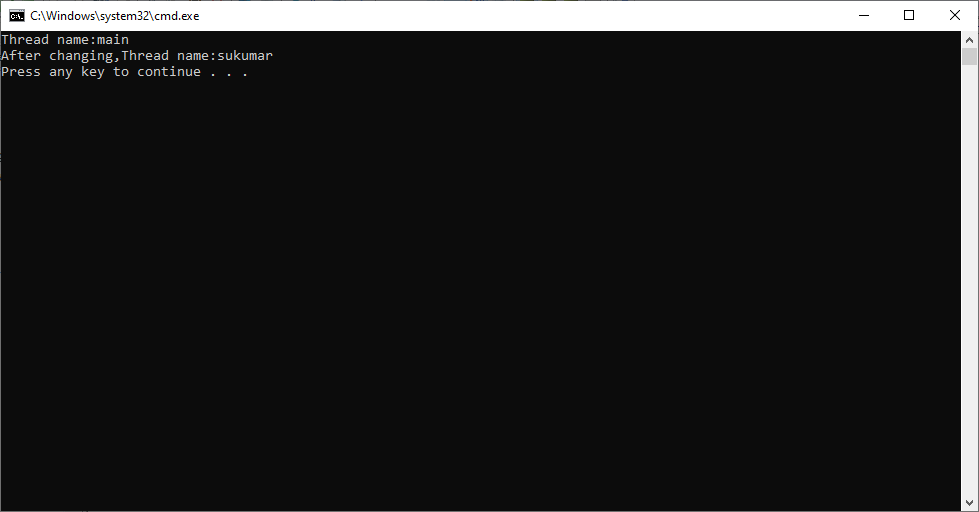
Thread.currentThread().setName("sukumar");

System.out.println("After changing,Thread name:"+ Thread.currentThread().getName());

}

}

Output:



Example:2

public class Test14 {

public static void main(String[] args) {

MThread m1 = new MThread();

m1.start();

}

}

class MThread extends Thread

{

public void run(){

System.out.println("Child Thread Name:"+ this.getName());

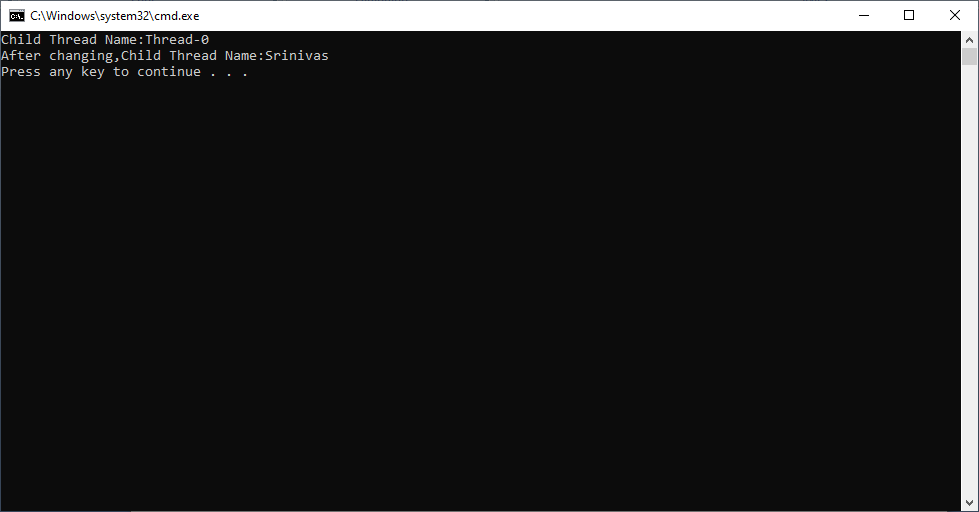
this.setName("Srinivas");

System.out.println("After changing,Child Thread Name:"+ this.getName());

}

}

Output:



Example:3

public class Test14 {

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

new MThread().start();

Thread.sleep(2000);

new Example().m1();

}

}

class Example extends Thread

{

static{

Thread th=Thread.currentThread();

System.out.println("Example class is loaded through the thread:"+ th.getName());

}

static void m1(){

Thread th=Thread.currentThread();

System.out.println("Example class m1 is executing in the thread:"+ th.getName());

}

}

class MThread extends Thread

{

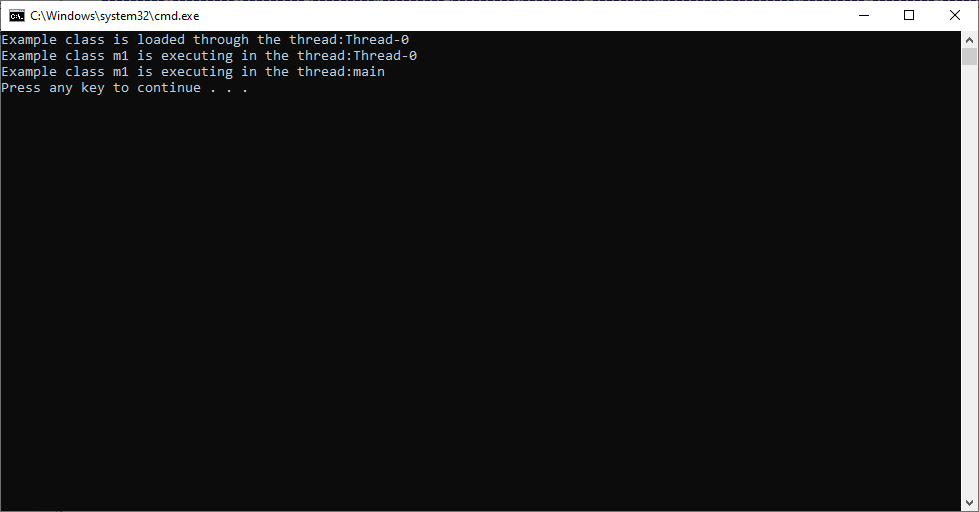
public void run(){

Example.m1();

}

}

Output:



**GETTING THREAD OBJECT REFERENCE**

The following method is used to retrieve reference of the currently running thrad object.

Syntax:

Public static native Thread currentThread().

Currently running thread is thread object in which out method logic is loaded and executing.

The main thread object reference is not available with out application, so we can not change its name and priority if we want, .but using this Thread.currentThread() method we can obtain main thread object reference,and further we can change its name and priority.

Example:

public class Test14 {

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

System.out.println("Main Thread name:"+ Thread.currentThread().getName());

System.out.println("Main Thread Priority:"+ Thread.currentThread().getPriority());

}

}

class MThread extends Thread

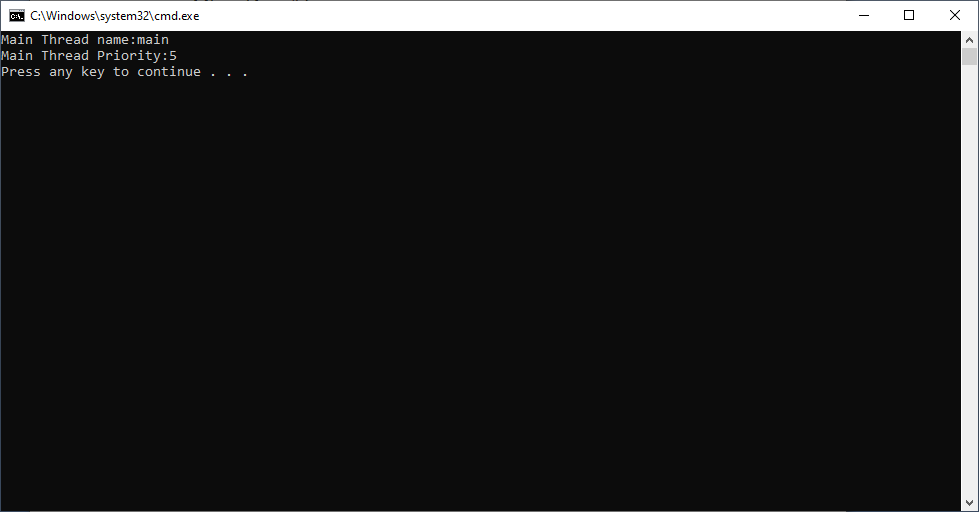
{

public void run(){

}

}

Output:



**THREAD GROUP**

Java provides a convenient way to group multiple threads in single object. A threadGroup is set of threads. A thread group can also have other thread group. A thread group can be represented as tree. In tree, every thread and other thread group has parent except thread group.

ThreadGroup class constructors:

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

Methods:

1. String getName() :- This method returns parent of the thread group.
2. ThreadGroup getParent():- This method returns the parent of the thread group object reference.
3. Void interrupt() :- this method interrupts all threads in the thread group.
4. Boolean isDaemon():- This method tests if the thread group is a daemon thread group.
5. Void setDaemon(boolean daemon):- This method changes the daemon status of the thread group.
6. Void list():- this method prints informaton about the thread group to the standard output.
7. Void suspend() :- this method is used to suspend all threads in the thread group.
8. Void resume():- This method isused to resume all threads in the thread group which was suspended.
9. setMaxPriority(int pri) : This method sets the maximum priority of the group.
10. Void stop():- This method is used to stop all threads in the thread group.
11. Int activeCount():- This method returns an estimate of number of active threads in thread group and its subgroups.

Every custom thread become member of default thread group. The default thread grop name is “main”.

In Thread class we have below method to retrieve current thread’s ThreadGroup object reference.

**Public final ThreadGroup getThreadGroup()**

**Example:1**

public class Test14 {

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

MThread m1=new MThread();

MThread m2=new MThread();

m1.start();

m2.start();

}

}

class MThread extends Thread

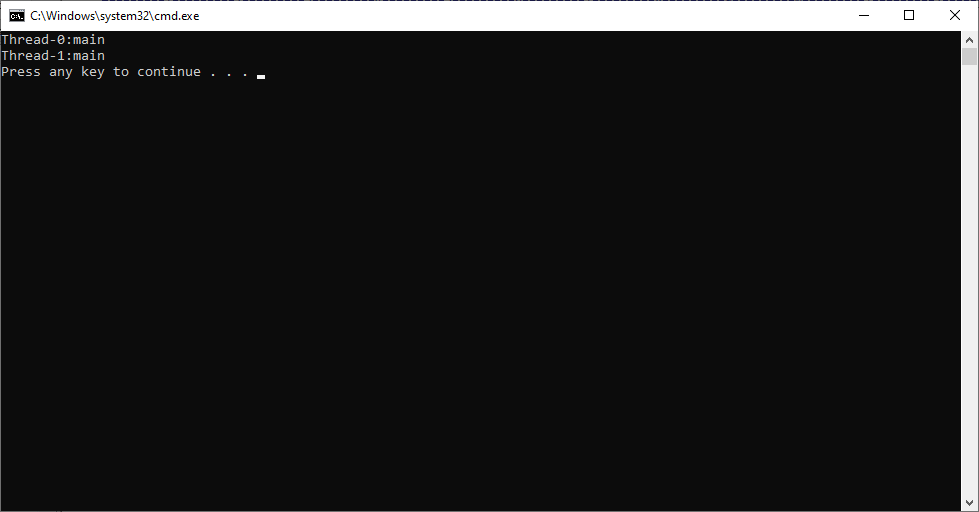
{

public void run(){

System.out.println(this.getName()+":"+ this.getThreadGroup().getName());

}

}



Steps to add custom thread to custom thread group.

Step1: create class which extends the “Thread” class.

Step2: In that class, write parameterized constructor. This constructor has to receive the two arguments. First one is “thread name” and second one is “threadgroup reference”.

Step3: write a super() method inside a parameterized constructor and pass thread name and threadgroup reference as arguments to super method.

Example:2 This program demonstrates that how to add custom thread to custom thread group.

public class Test14 {

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

ThreadGroup tg=new ThreadGroup("sukumar");

MThread m1=new MThread("sambaveena",tg);

MThread m2=new MThread("sulamaha",tg);

m1.start();

m2.start();

}

}

class MThread extends Thread

{

MThread(String a, ThreadGroup b){

super(b,a);

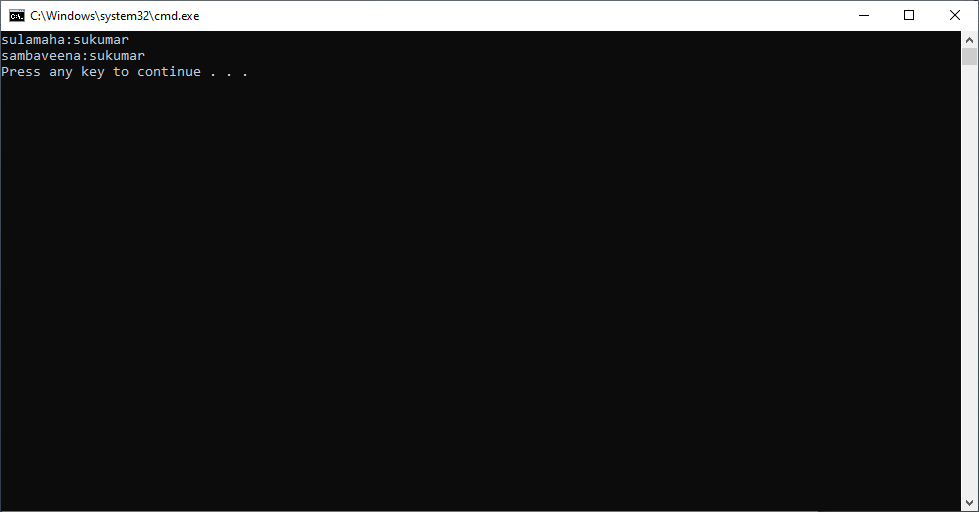
}

public void run(){

System.out.println(this.getName()+":"+ this.getThreadGroup().getName());

}

}



**THREAD TYPES**

Java allows us to create two types of threads. They are

1. Non-Daemon Thread
2. Daemon Thread

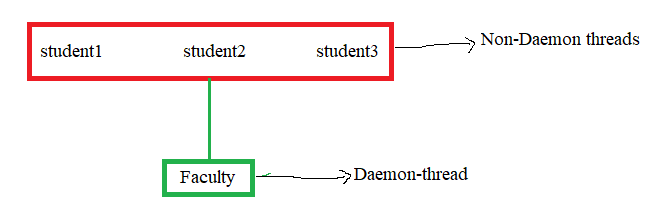
**i.Non-Daemon Thread:-** The thread which executes main logic of the project is called non-daemon thread. Every custom thread is non-daemon thread by default because daemon property is also in inherited from paeent thread(main thread).

**ii. Daemon Thread:-** The thread which provides service to non-daemon thread and which is running in the background is called daemon thread. The daemon thread is also said to be as service thread.

when all non-daemon threads execution is completed, JVM forcely stops daemon thread execution.The daemon thread has low priority therefore we can not guarantee its execution.

**Example:** The garbage collector is daemon thread. Since garbage collector provides service-destroing unreferenced objects from heap memory.

General Example to Non-daemon and Daemon thread



The faculty provides services to students.

**1.Daemon thread creation:** The thread class has below method to create user defined thread as daemon thread.

**Public final void setDaemon( boolean on)**

If on value is true – thread is created as daemon.

Else it is created as non-daemon thread. So the daemon property default value is false.

Thread class has below method to check thread is daemon or non-daemon.

**Public final boolean isDaemon()**

Returns true if thread is daemon, else returns false.

Note:- setDaemon() method should not be called after start() method call, it leads to **RE**:

**Java.lang.illegalThreadStateException.**

Because once thread is created as non-daemon thread, It can not be converted as daemon.

**Example:1**

public class Test14 {

public static void main(String[] args) {

MThread m1=new MThread();

m1.start();

m1.setDaemon(true);

}

}

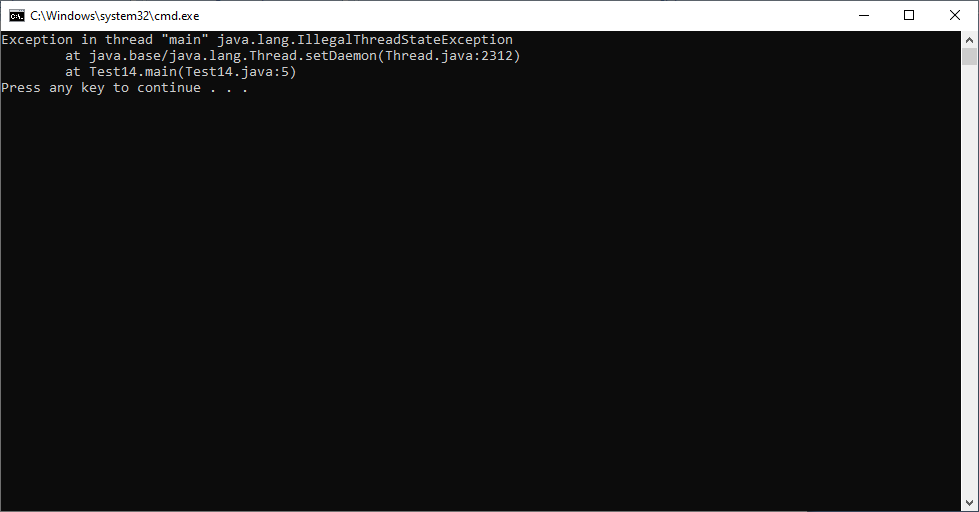
class MThread extends Thread

{

public void run(){

}

}



Example:2

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.setDaemon(true);

m1.start();

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

{

public void run(){

System.out.println("Child Thread & Daemon Thread Started");

for(int i=1;i<20;i++)

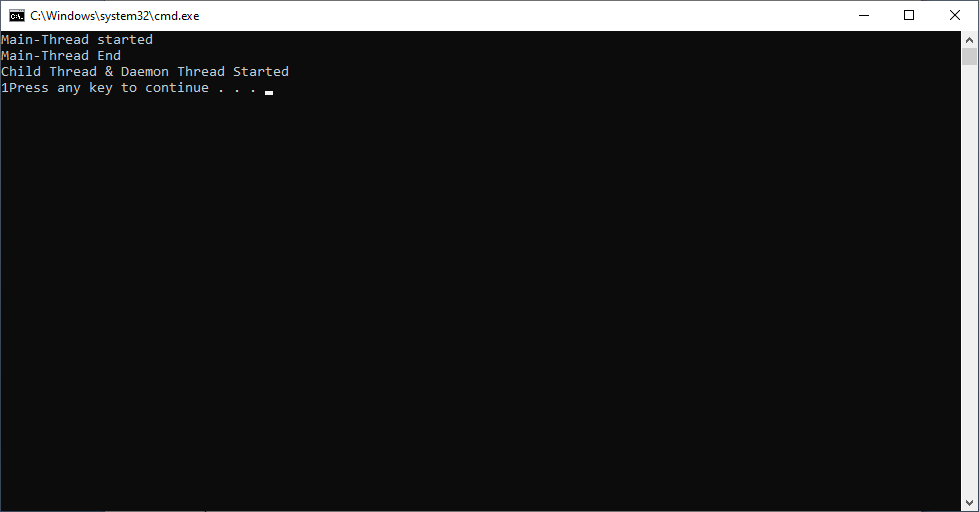
{

System.out.print(i);

}

}

}



Note:- Daemon thread execution is terminated in middle, all iterations output will not be printed.

**CONTROLLING THE THREAD EXECUTION**

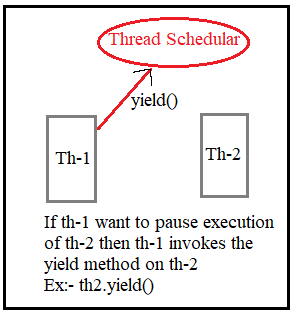
The custom threads execution is usually controlled by Thread schedular. To control thread execution by application, The thread class provided several methods.

1. **Yield:-** The yield() method request thread schedular for pausing thread execution. The thread schedular may ignore or accept this request.

If thread schedular accept request then it pause the specified thread execution.

Syntax:

**Public void yield();**

****

Example:-1

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.start();

m1.yield();

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

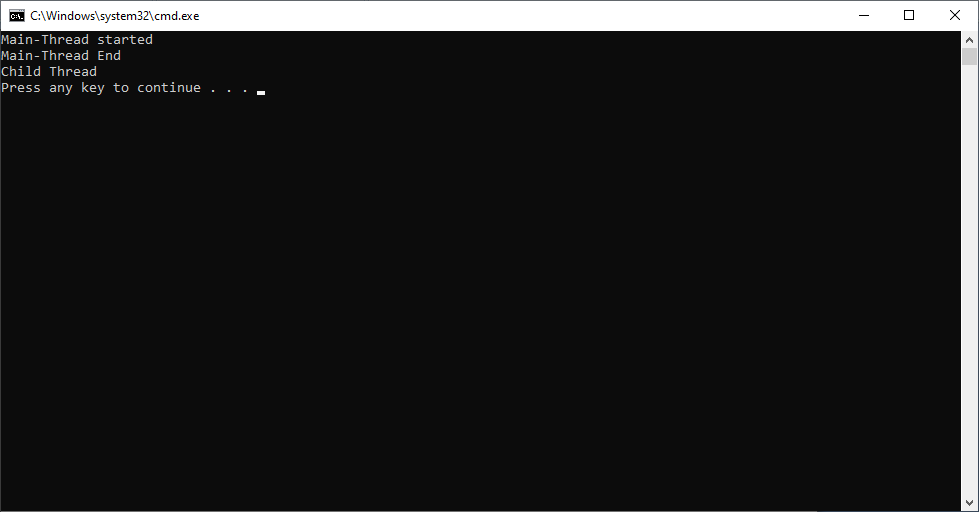
{

public void run(){

System.out.println("Child Thread ");

}

}



In above example, Thread schedular accepted yield() request.

Example:2

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.start();

m1.yield();

for(int i=13;i<20;i++)

{

System.out.print(i);

}

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

{

public void run(){

System.out.println("Child Thread ");

for(int i=1;i<12;i++)

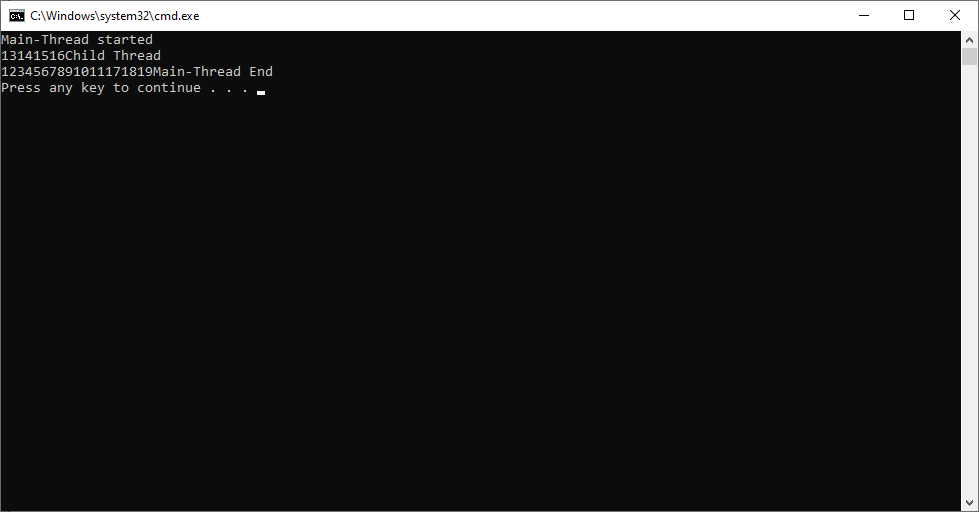
{

System.out.print(i);

}

}

}



In above example, Thread schedular ignored the yield() request.

**2.sleep():-**

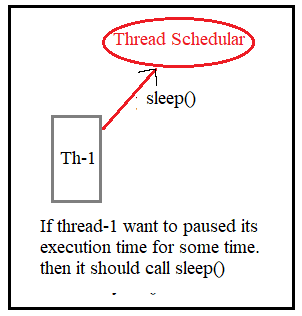
Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread for specified Time.

Syntax:

**Public static void sleep(long mls) throws InterruptedException.**

**Public static void sleep(long mls, int s) throws InterruptedException**.

Note:- The thread schedular should accept the sleep() method request.



Example:

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.start();

for(int i=13;i<20;i++)

{

System.out.print(i);

}

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

{

public void run(){

try{

Thread.sleep(2000);

}

catch(InterruptedException e){

System.out.println("RunTime Error");

}

System.out.println("Child Thread ");

for(int i=1;i<12;i++)

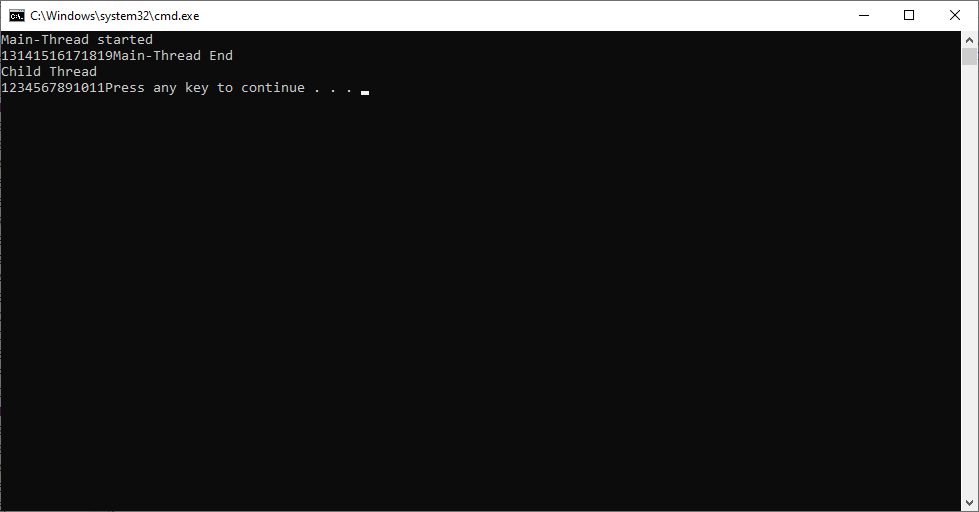
{

System.out.print(i);

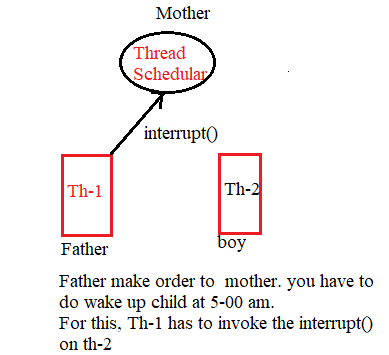
}

}

}



**3.Interrupt:-** This method brings thread out from paused state.



Example:

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.start();

m1.interrupt();

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

{

public void run(){

try{

Thread.sleep(2000);

}

catch(InterruptedException e){

System.out.println("Child Thread ");

for(int i=1;i<12;i++)

{

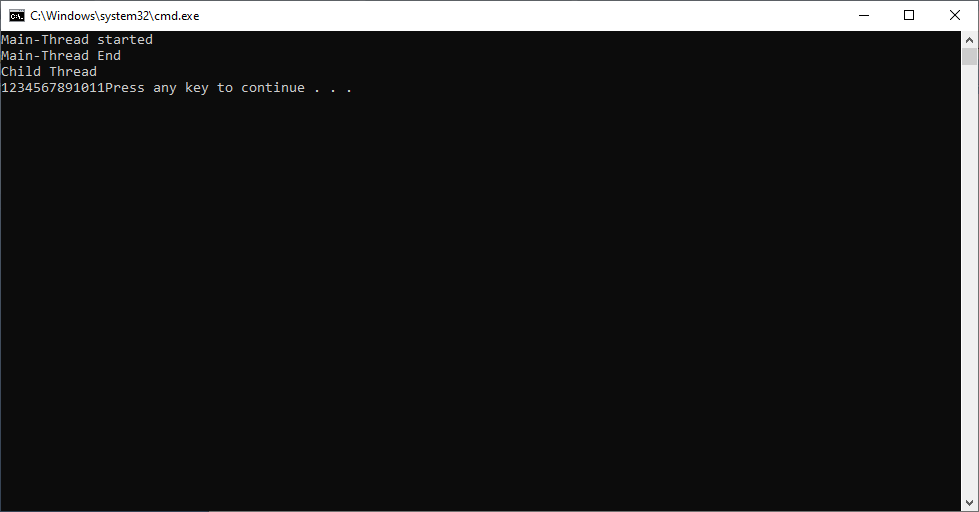
System.out.print(i);

}

}

}

}



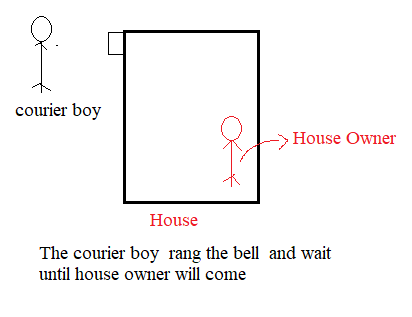
**4.Join:-**

**Syntax:1**

**Public final void join() throws InterruptedException**

The join method pauses the thread execution until completion of other thread execution. If other thread execution is blocked forever, this thread execution is also blocked forever.

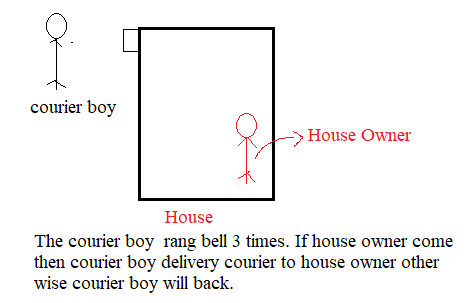
General Example:



Syntax:-2

**Public final void join(long mills) throws InterruptedException**

Join method does not block thread execution until completion of other thread execution. Its execution is resumed after completion of given time.



Example:1

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.start();

try{

m1.join();

}

catch(InterruptedException e){ System.out.println("RunTimeError");}

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

{

public void run(){

System.out.println("Child Thread ");

for(int i=1;i<12;i++)

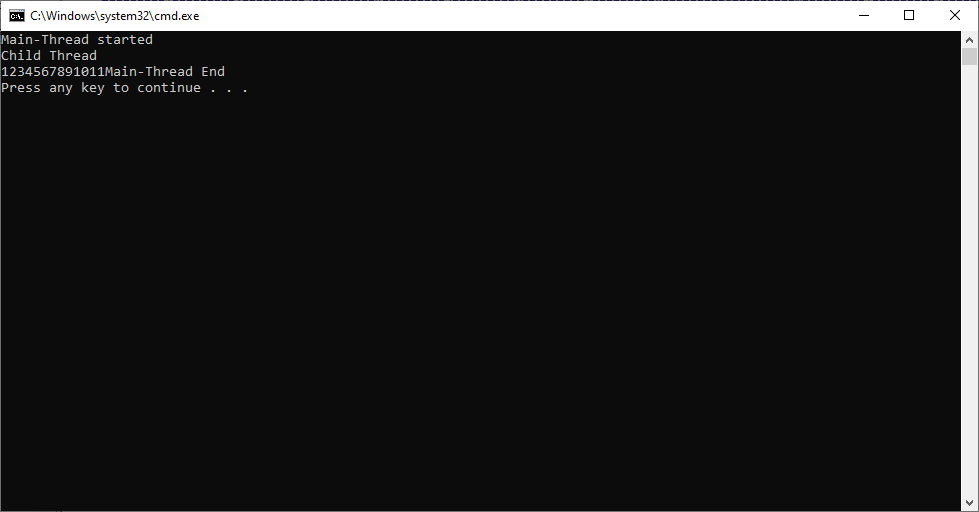
{

System.out.print(i);

}

}

}



Exception:2

public class Test14 {

public static void main(String[] args) {

System.out.println("Main-Thread started");

MThread m1=new MThread();

m1.start();

try{

m1.join(1000);

}

catch(InterruptedException e){ System.out.println("RunTimeError");}

System.out.println("Main-Thread End");

}

}

class MThread extends Thread

{

public void run(){

try{

Thread.sleep(2000);

}

catch(InterruptedException e){

System.out.println("Run Time Error in child");

}

System.out.println("Child Thread ");

for(int i=1;i<12;i++)

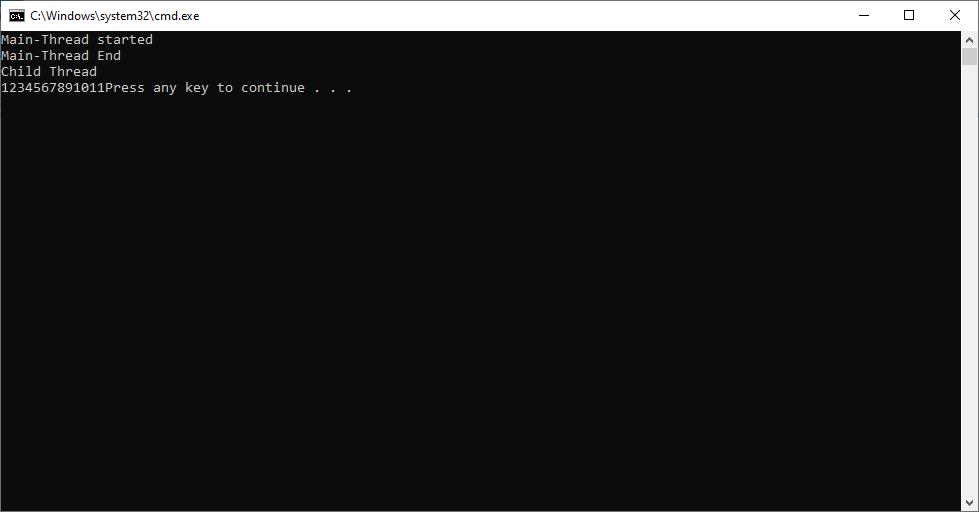
{

System.out.print(i);

}

}

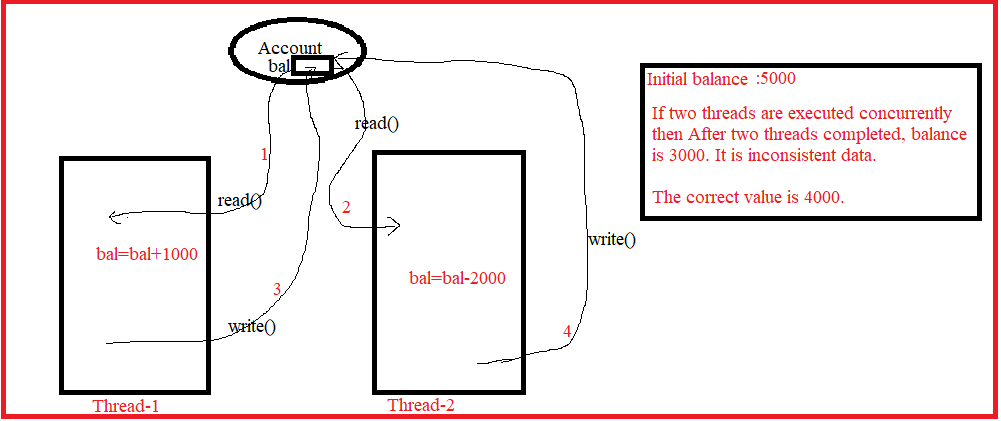
}



**INNERTHREAD COMMUNICATION-1**

Threads are executed concurrently. In this case whe first thread execution is paused, second thread modifies object daa in the middle of first thread execution, after resuming first thread it uses the second thread modified values, but not its actual values. This is called data inconsistent modification problem and it leads to wrong results.

Example:-



The solution to data inconsistency is synchronization.

**1.Synchronization:- The process of executing multiple threads in sequence when they are sharing |using same object is called synchronization. When multiple threads ae modifying same object we must develop synchronization.**

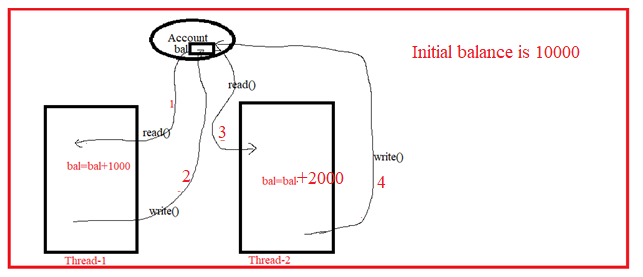
In Java synchronization is developed,

1. By using synchronized methods
2. By using synchronized blocks.

**2. Synchronized Method:-** If the method is declared as synchronized then It is known as Synchronized method. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

If another thread already got lock for the object , invoking thread execution is paused.

Example:1



public class Test14 {

public static void main(String[] args) {

Account a=new Account();

A a1=new A(a);

B a2=new B(a);

a1.start();

a2.start();

}

}

class Account

{

int balance=10000;

public synchronized void deposit(int x)

{

try{ Thread.sleep(2000);}

catch(Exception e){ System.out.println("RunTime Error");}

balance=balance+x;

}

}

class A extends Thread

{ Account acc;

A(Account x){ this.acc=x; }

public void run(){

acc.deposit(1000);

System.out.println(this.getName()+"-"+"Balance: "+acc.balance);

}

}

class B extends Thread

{ Account acc;

B(Account x){ this.acc=x; }

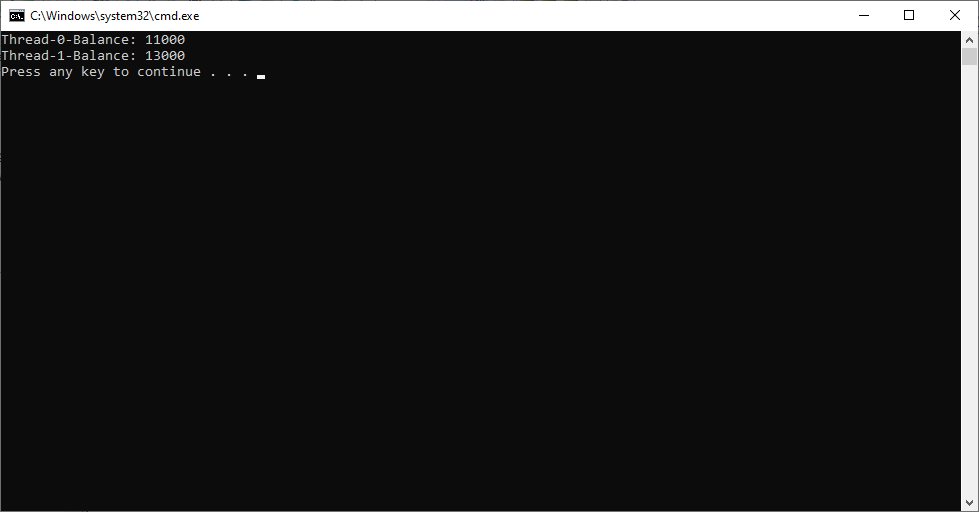
public void run(){

acc.deposit(2000);

System.out.println(this.getName()+"-"+"Balance: "+acc.balance);

}

}



Note:- in above example, If we declare non-static method as synchronized its current object is locked, so that in this method non-static variables of this object are modified sequentially by multiple threads.

Example:2

public class Test14 {

public static void main(String[] args) {

Account a=new Account();

A a1=new A(a);

B a2=new B(a);

a1.start();

a2.start();

}

}

class Account

{

static int balance=10000;

public synchronized static void deposit(int x)

{

try{ Thread.sleep(2000);}

catch(Exception e){ System.out.println("RunTime Error");}

balance=balance+x;

}

}

class A extends Thread

{ Account acc;

A(Account x){ this.acc=x; }

public void run(){

acc.deposit(1000);

System.out.println(this.getName()+"-"+"Balance: "+acc.balance);

}

}

class B extends Thread

{ Account acc;

B(Account x){ this.acc=x; }

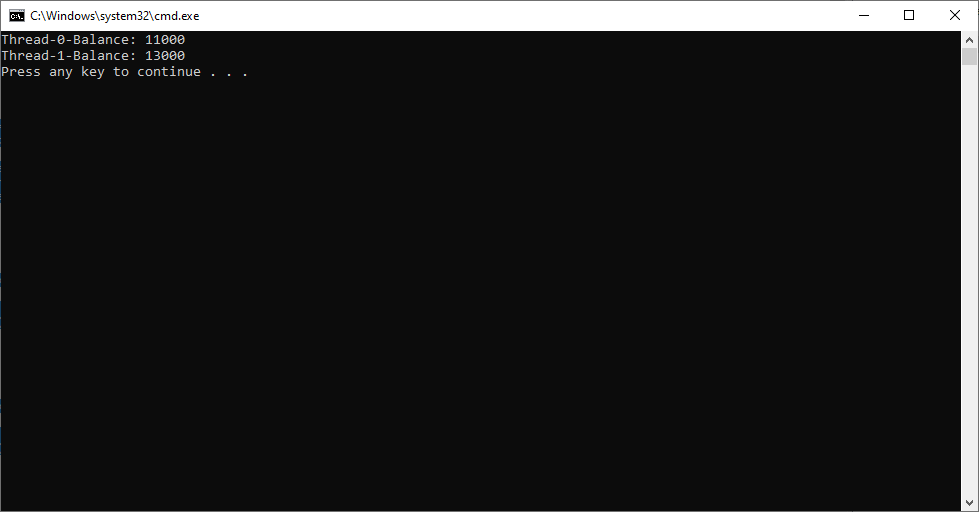
public void run(){

acc.deposit(2000);

System.out.println(this.getName()+"-"+"Balance: "+acc.balance);

}

}



Note:- In above example, we declared static method as synchronized its class’s java.lang.class object is locked, so that in this method Stativ variables are modified sequentially by multiple threads.

**3.Synchronized Block:-** If we declare block as synchronized, only the statements written inside that block are executed sequentially not complete method logic.

Syntax:

Synchronized (obj){}

Examples:

Synchronized(this) // It lock the current object.

Synchronized(a) // It lock the argument object.

Example:

public class Test14 {

public static void main(String[] args) {

Account a=new Account();

A a1=new A(a);

B a2=new B(a);

a1.start();

a2.start();

}

}

class Account

{

int balance=10000;

String tn;

public void deposit(int x,String y)

{ tn=y;

System.out.println("started:"+y);

synchronized(this){

try{ Thread.sleep(2000);}

catch(Exception e){ System.out.println("RunTime Error");}

balance=balance+x;

}

}

}

class A extends Thread

{ Account acc;

A(Account x){ this.acc=x; }

public void run(){

acc.deposit(1000,Thread.currentThread().getName());

System.out.println(this.getName()+"-"+"Balance: "+acc.balance);

}

}

class B extends Thread

{ Account acc;

B(Account x){ this.acc=x; }

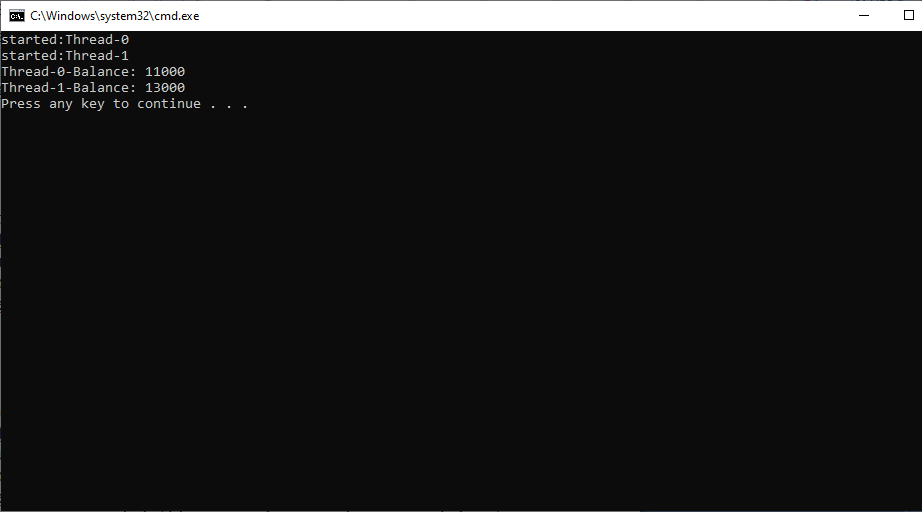
public void run(){

acc.deposit(2000,Thread.currentThread().getName());

System.out.println(this.getName()+"-"+"Balance: "+acc.balance);

}

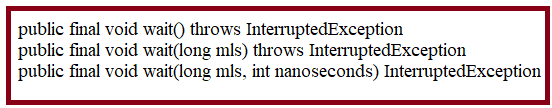
}



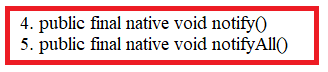
**4.Thread safe Object:-** If an object is not accessible for multiple threads concurrently for modifying its value, or one thread modification on the object is not affected to another thread , then that object is called thread safe object.

**INTERTHREAD COMMUNICATION-2**

To develop inter-thread communication, we must use following methods along with synchronized keyword. The methods are defined in java.lang.Object class

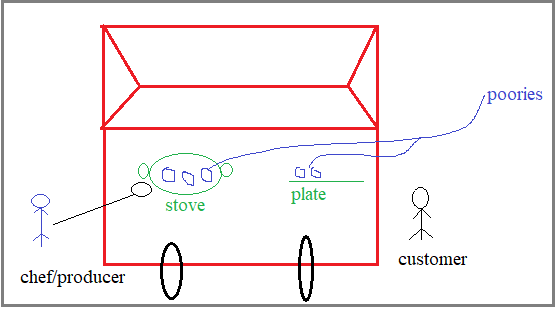


The wait() releasing lock of current object and pusing currently executing thread.



The functionality of notify() is notifying to waiting to waiting thread[s] about lock availability of object. Then waiting thread is moved from “wait” state to “wait for lock acquisition” state. Once lock is obtained then that waiting thread is moved to runnable state.

Example: Producer and consumer Example.



class Factory{

int items;

boolean itemsInFactory;

synchronized void produce(int items){

if (itemsInFactory)

{

try{wait();}

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

}

this.items=items;

itemsInFactory=true;

System.out.println("Produced Items:"+ items);

notify();

}

synchronized int consume()

{

if(!itemsInFactory){

try{wait();}

catch(InterruptedException e)

{ e.printStackTrace();}

}

System.out.println("Items Consumed:"+ items);

itemsInFactory=false;

notify();

return items;

}

}

class producer implements Runnable

{

Factory fa;

producer(Factory fa)

{

this.fa=fa;

new Thread(this,"producer").start();

}

public void run()

{

int i=1;

while(i<=10)

{

fa.produce(i++);

}

}

}

class Consumer implements Runnable

{

Factory fa;

Consumer(Factory fa)

{

this.fa=fa;

new Thread(this,"consumer");

}

public void run()

{

int i=1;

while(i<=10)

{

fa.consume();

i++;

}

}

}

public class sample

{

public static void main(String[] args)

{

Factory bajaj=new Factory();

new producer(bajaj);

new Consumer(bajaj);

}

}