**Thread:**

* Thread is a lightweight sub process.
* It is the smallest independent unit of a program.
* It contains a separate path of execution.
* Every java program contains at least one thread.
* A thread created & controlled by the java.lang.Thread class.
* If there occurs exception in one thread, it does not affect the other threads.
* It uses a shared memory area. One process can have multiple threads.
* At a time one thread is executed only.

Thread life cycle:

Example:

**public** **class** ThreadDemo {

//main method represents main thread

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

//whatever we write in here will be executed by main thread.

//threads always execute the jobs in a sequence.

//job1

System.***out***.println("--Application Started--");

//job2

**For**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document : "+doc);

}

//job3

System.***out***.println("--Application Finished--");

}

}

Output:

--Application Started--

Printing Document: 1

Printing Document: 2

Printing Document: 3

Printing Document: 4

Printing Document: 5

--Application Finished--

Eg2:

**class** Mytask{

**public** **void** executeTask() {

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in Mytask: "+doc);

}

}

}

**public** **class** ThreadDemo {

//main method represents main thread

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

//job1

System.***out***.println("--Application Started--");

//job2

Mytask mt=**new** Mytask();

mt.executeTask();

//Till job2 is not finished, below written jobs are waiting and are not executed.

//In case job2 is a long running operation, i.e several documents are supposed to be printed.

//In such use case OS/ JVM shall give a message that ThreadDemo is not responding.

//job3

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in main: "+doc);

}

//job4

System.***out***.println("--Application Finished--");

}

}

Output: all are executed in a sequence

--Application Started--

Printing Document in Mytask: 1

Printing Document in Mytask: 2

Printing Document in Mytask: 3

Printing Document in Mytask: 4

Printing Document in Mytask: 5

Printing Document in main: 1

Printing Document in main: 2

Printing Document in main: 3

Printing Document in main: 4

Printing Document in main: 5

--Application Finished--

Eg3: Using the thread class

**class** Mytask **extends** Thread{

@Override

**public** **void** run() {

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in Mytask: "+doc);

}

}

}

**public** **class** ThreadDemo {

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

//job1

System.***out***.println("--Application Started--");

//job2

Thread t=**new** Mytask();

t.start();

//start() shall internally execute run() method.

//Now, main and Mytask are executed parallelly or concurrently.

//job3

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in main: "+doc);

}

//job4

System.***out***.println("--Application Finished--");

}

}

Output:

--Application Started--

Printing Document in main: 1

Printing Document in main: 2

Printing Document in main: 3

Printing Document in main: 4

Printing Document in main: 5

Printing Document in Mytask: 1

Printing Document in Mytask: 2

Printing Document in Mytask: 3

--Application Finished--

Printing Document in Mytask: 4

Printing Document in Mytask: 5

Eg4: runnable interface

**class** Mytask **extends** Thread **implements** Runnable{

@Override

**public** **void** run() {

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in Mytask: "+doc);

}

}

}

**public** **class** ThreadDemo {

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

//job1

System.***out***.println("--Application Started--");

//job2

Runnable r=**new** Mytask(); //interface

Thread t=**new** Thread(r); //parent class

t.start();

//job3

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in main: "+doc);

}

//job4

System.***out***.println("--Application Finished--");

}

}

Output:

--Application Started--

Printing Document in main: 1

Printing Document in main: 2

Printing Document in main: 3

Printing Document in main: 4

Printing Document in Mytask: 1

Printing Document in Mytask: 2

Printing Document in Mytask: 3

Printing Document in main: 5

--Application Finished--

Printing Document in Mytask: 4

Printing Document in Mytask: 5

Thread class VS Runnable Interface:

-----------------------------------------------

|  |  |
| --- | --- |
| Thread class | Runnable Interface |
| Each thread creates its unique object. | Each thread creates its unique object. |
| More memory consumption. | More memory consumption. |
| A class extending thread class cannot extend any other class. | Along runnable a class can implement any other interface. |
| Thread class is extended only if there is a need of overriding other methods of Thread class. | Runnable is implemented only if there is a need of special run method. |
| Enables tight coupling. | Enables loose coupling. |

Java Main Thread:

* Main thread is the most important thread of a java program.
* It is executed whenever a java program starts.
* Every program must contain this thread for its execution to take place.
* Java main thread is needed because of the following reasons.

1.From this other child threads are spawned.

2.It must be the last thread to finish execution i.e when the main thread stops program terminates.

Runnable r=**new** Mytask(); //interface

Thread t=**new** Thread(r); //parent class

t.start();

--we can simplify the above statements as:

Thread t=new Thread (new Mytask ());

t.start();

--We can further simplified as:

new Thread (new Mytask ()). Start ();

\*We can set any thread as Daemon thread:

t.setDaemon(true);

Daemon thread: It is a thread which is going to be executed by the JVM whenever the application starts.

Daemon threads are useful for background supporting tasks such as garbage collection, releasing memory of unused objects and removing unwanted entries from the cache. Most of the JVM threads are daemon threads.

Eg: Garbage Collector, finilizer()

Eg: Runnable r=**new** Mytask(); //interface

Thread t=**new** Thread(r); //parent class

t.setDaemon(**true**);

t.start();

Multi - Threading:

------------------------

Multithreading is the ability of a program to run two or more threads concurrently, where each thread can handle a different task at the same time making optimal use of the available resources.

--Each thread is by default joined with as a last statement of the main method, this ensure that after completion of the threads then only main thread will terminate.

**Multithreading:**

Suspending a thread based on time:

------------------------------------------------

Eg1:

**class** MyThread **implements** Runnable{

@Override

**public** **void** run() {

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

String str=Thread.*currentThread*().getName();

System.***out***.println(str+" is running "+i);

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**public** **class** ThreadDemo {

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

MyThread r=**new** MyThread();

Thread t=**new** Thread(r);

t.setName("siri");

t.start();

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

String str=Thread.*currentThread*().getName();

System.***out***.println(str+" is running "+i);

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

} }

Suspending a thread conditionally:

----------------------------------------------

Join () method

If we want to suspend a running thread conditionally then we should use join () method of the thread class.

The join () method is a non – static method.

If a thread wants to wait until completion of other thread then we should use join () method.

Thread1 thread2 Thread3

t1.join (); t2.join ();

Here t2 thread will wait until completion of t1 thread and t3 thread will wait until the completion of t2 thread.

Example: sum of n numbers

**class** MyThread **implements** Runnable{

**int** sum;

@Override

**public** **void** run() {

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

{

sum=sum+i;

}

}

}

**public** **class** ThreadDemo {

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

MyThread mt=**new** MyThread();

Thread t=**new** Thread(mt);

t.start();

t.join(); //here main thread will wait until t thread completes.

//If we comment t.join() then we will get incorrect value.

**int** result=mt.sum;

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

{

System.***out***.println("Inside the main...");

System.***out***.println(result);

}

}

}

Output:

Inside the main...

45

Inside the main...

45

Inside the main...

45

Inside the main...

45

Inside the main...

45

Thread safety in java:

The concept of avoiding multiple threads acting upon the same functionality simultaneously is known as Thread safety and data inconsistency problem.

Race Condition:

A race condition is a condition in which the critical section (shared memory accessed) is concurrently executed by two or more threads. It leads to incorrect behaviour of a program.

Race condition is avoided by the help of synchronized keyword.

Note: The synchronized keyword applicable only for methods and blocks but not for variables and classes.

Advantage of the synchronized keyword is we can resolve data inconsistency problem.

Disadvantage of the synchronized keyword it increases waiting time of the threads and creates performance problem on it.

Example:

checking seat availability method should be non – synchronized, whereas book seat method should be synchronized.

Any method that changes the state of an object I.e add/ update/ delete/ replace method we should use as synchronized.

Eg:

**class** Printer{

//synchronized after the completion one thread it will go to next thread.

**synchronized** **void** printDocument(**int** numOfCopies,String docName) {

**for**(**int** doc=1;doc<=numOfCopies;doc++) {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println("Printing Document in printer class: "+docName+" "+doc);

}

}

}

**class** MyThread **extends** Thread{

Printer p;

**public** MyThread(Printer p) {

**this**.p=p;

}

@Override

**public** **void** run() {

//synchronized block

**synchronized**(p) {

p.printDocument(5, "SiriProfile.pdf");

}

}

}

**class** YourThread **extends** Thread{

Printer p;

**public** YourThread(Printer p) {

**this**.p=p;

}

@Override

**public** **void** run() {

p.printDocument(5, "DhaanviProfile.pdf");

}

}

**public** **class** ThreadDemo {

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

//job1

System.***out***.println("--Application Started--");

//job2

Printer p=**new** Printer();

MyThread m=**new** MyThread(p);

YourThread y=**new** YourThread(p);

m.start();

// try {

// m.join();

// } catch (InterruptedException e) {

// // **TODO** Auto-generated catch block

// e.printStackTrace();

// }

y.start();

//job3

**for**(**int** doc=1;doc<=5;doc++) {

System.***out***.println("Printing Document in main: "+doc);

}

//job4

System.***out***.println("--Application Finished--");

}

}

Eg2:

**class** Common{

//synchronized: It will allow only one thread at a time.

**public** **synchronized** **void** fun(String name) {

System.***out***.print("Welcome ");

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.***out***.println(name);

}

}

**class** ThreadA **extends** Thread{

Common c;

String name;

**public** ThreadA(Common c,String name) {

**this**.c=c;

**this**.name=name;

}

@Override

**public** **void** run() {

c.fun(name);

}

}

**class** ThreadB **extends** Thread{

Common c;

String name;

**public** ThreadB(Common c,String name) {

**this**.c=c;

**this**.name=name;

}

@Override

**public** **void** run() {

c.fun(name);

}

}

**public** **class** ThreadDemo {

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

Common c=**new** Common();

ThreadA t1=**new** ThreadA(c,"Ram");

ThreadB t2=**new** ThreadB(c,"Abi");

t1.start();

t2.start();

}

}

Output:

Welcome Abi

Welcome Ram

Synchronization concept:

* Internally synchronization concept is implemented by using lock concept.
* Every object in a java has a unique lock. Most of the time the lock is unlocked.
* When an object has one or more synchronized methods, a thread can enter into a synchronized method or block only when if that thread have the lock of that object.
* The locks are not per methods basis, instead they are per object basis.
* The thread won’t release the lock until it completes the synchronized methods, so while that thread is holding the lock of that object. Once a synchronized method execution completed then thread releases the lock automatically.
* Until the lock is released (completion of synchronized method) no other threads can enter any of the synchronized methods or blocks of that object.
* So if an object has synchronized methods or blocks, a thread can enter any one of the synchronized methods or block only if the lock of that object is available.
* Acquiring and releasing the lock internally taken care by JVM, programmer are not responsible for this activity.

Class level lock:

---------------------

In java as there is a unique for each object of a class, similarly there is a unique lock for each class also. So there are two types of lock in java:

1. Object level lock (It is unique for each object of a class).
2. Class level lock (It is unique for each class)

If a thread tries to execute a static synchronized method then it required class level lock.

Object lock and class level lock both are independent and there is no link between them.

Eg:

Class A {

Static synchronized funA () {} –class level lock

Static synchronized funB () {} –class level lock

Synchronized funC () {} –object level lock

funD () {} – It can execute simultaneously

static funE () {} – It can execute simultaneously

}

A a1=new A ();

A1.funA ();

While a thread is executing a static synchronized method, the remaining threads are not allowed to execute any other static synchronized method of that class simultaneously (even on the multiple object of that class also).

But remaining threads are allowed to execute normal static and synchronized non-static methods and normal non static method simultaneously.

Synchronized block:

If very few lines of the code requires synchronization then it is not recommended to declare entire method as synchronized. We have to enclose those few lines of the code in synchronized block.

In a method, assume 10000 lines code is there, and in the middle somewhere few line need some database operation like (update/ delete/ add) then declaring entire method as synchronized is a worst kind of programming here it degrades the performance.

Example:

**public** **void** fun(String name) {

**synchronized** (**this**) {

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(name);

}

}

Eg2: class

**synchronized** (Student.**class**) {

//if a thread gets the class level lock of class Student

//then only it is alloed to execute following block.

System.***out***.println(name);

}

--with the help of synchronized method we can access only one lock (either object level or class level).

--but with the synchronized block we can access multiple locks also.

Note: A thread acquire multiple locks.

Eg:

A a1=new A ();

A a2=new A ();

Public void fun1 () {

Synchronized (A.class) {//class

Synchronized (B.class) {

Synchronized (a1) {//object

Synchronized(a2) {

----------------- //critical section

}

}

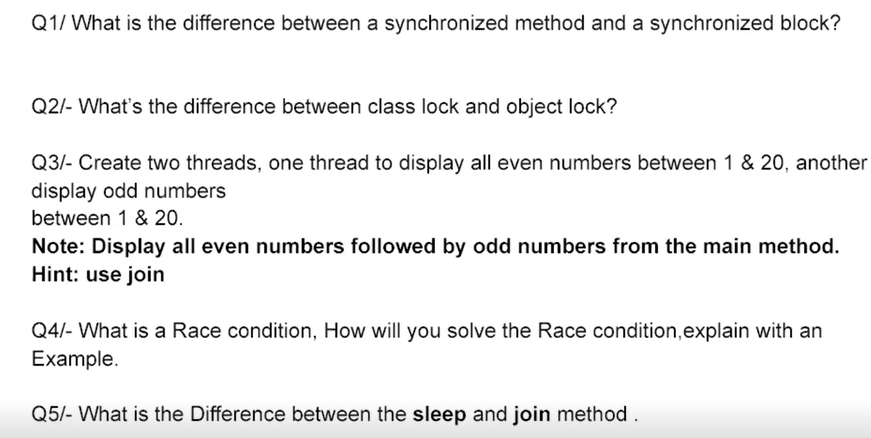
}

}

}

--this critical section can be executed by only that thread which is having 4 locks

Problems:



Question3:

**class** ThreadEven **extends** Thread{

@Override

**public** **void** run() {

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

**if**(i%2==0)

System.***out***.println("Even Number:"+i);

}

}

}

**class** ThreadOdd **extends** Thread{

Thread tr;

**public** ThreadOdd(ThreadEven tr) {

**this**.tr=tr;

}

@Override

**public** **void** run() {

**try** {

tr.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

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

**if**(i%2!=0)

System.***out***.println("Odd Number:"+i);

}

}

}

**public** **class** ThreadEx {

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

ThreadEven te=**new** ThreadEven();

ThreadOdd to=**new** ThreadOdd(te);

te.start();

to.start();

}

}

Output:

Even Number:2

Even Number:4

Even Number:6

Even Number:8

Even Number:10

Even Number:12

Even Number:14

Even Number:16

Even Number:18

Even Number:20

Odd Number:1

Odd Number:3

Odd Number:5

Odd Number:7

Odd Number:9

Odd Number:11

Odd Number:13

Odd Number:15

Odd Number:17

Odd Number:19

Inter thread communication or thread synchronization:

It means two synchronized threads communicate each other.

Two synchronized threads can communicate each other by using methods present in object class, those methods are wait (), notify (), notifyAll ().

By using above methods we can gain partial control on the scheduling mechanism which is supervised by the thread-scheduler.

Whenever we need to suspend a synchronized thread unconditionally then we use wait () method.

Whenever we need to resume a suspended (waiting) thread then we use notify () method.

This is known as thread-synchronization.

In the inter - thread communication the thread which require updation it has to call wait () method.

The thread which performing updation it will call notify () method, so that waiting thread will gets the notification and it continues its execution with those updation.

NOTE: we can call wait (), notify (), notifyAll () only in the synchronized block or synchronized methods. Otherwise we will get a runtime exception called IllegalMonitorStateException.

Once thread calls wait () method on any object, first it releases the lock immediately of that particular object and then it enters into the waiting state immediately.

Once thread calls notify () method on any object it also releases the lock of that object but not immediately.

Wait (): without specifying time limit thread will go to the waiting state forever. Wait (5000) 5 sec.

Notify ():

Join (): we can also use join () method but it will take more time to execute and main thread will wait long time.

Eg:

**class** MyThread **extends** Thread{

**int** num=0;

@Override

**public** **void** run() {

**synchronized**(**this**) {

System.***out***.println("child thread performing calculation");

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

num+=i;

}

System.***out***.println("child thread giving the notification");

**this**.notify();

}

}

}

**public** **class** ThreadDemo {

**public** **static** **void** main(String[] args)

{

MyThread mt=**new** MyThread();

mt.start();

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**synchronized**(mt) {

**try** {

System.***out***.println("main thread calls the wait method");

mt.wait(5000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

System.***out***.println("Result:"+mt.num);

}

}