**MULTITHREADING ASSIGNMENT**

**[ RAGHAV GUPTA ]**

**-------------------------------------------------------**

1. **Write a program do to demonstrate the use of volatile keyword.**

**package** multithreadingVideos;

**import** java.util.Scanner;

**class** Processor **extends** Thread{

**private volatile boolean running** = **true**;

**public void** run(){

**while**(**running**){

System.***out***.println(**"HELLO"**);

**try**{

Thread.*sleep*(100);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

}

}

**public void** shutdown(){

**running** = **false**;

System.***out***.println(**"processor terminated"**);

}

}

**public class** Volatile {

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

Processor p1 = **new** Processor();

p1.start();

System.***out***.println(**"press enter to stop."**);

Scanner sc = **new** Scanner(System.***in***);

sc.nextLine();

p1.shutdown();

}

}

*/\* the concept here is that, here the while loop will run till the value of running is true.*

*\* we call shutdown function to change its value and stop the thead.*

*\* but in some scenarios where java tries to optimise the execution of threads because of*

*\* some special situation, or may be in some distibutions of java, or where there are multiple*

*\* threads. so, the thread caches the value of the running variable. and never sees the updated*

*\* value. so never stops.*

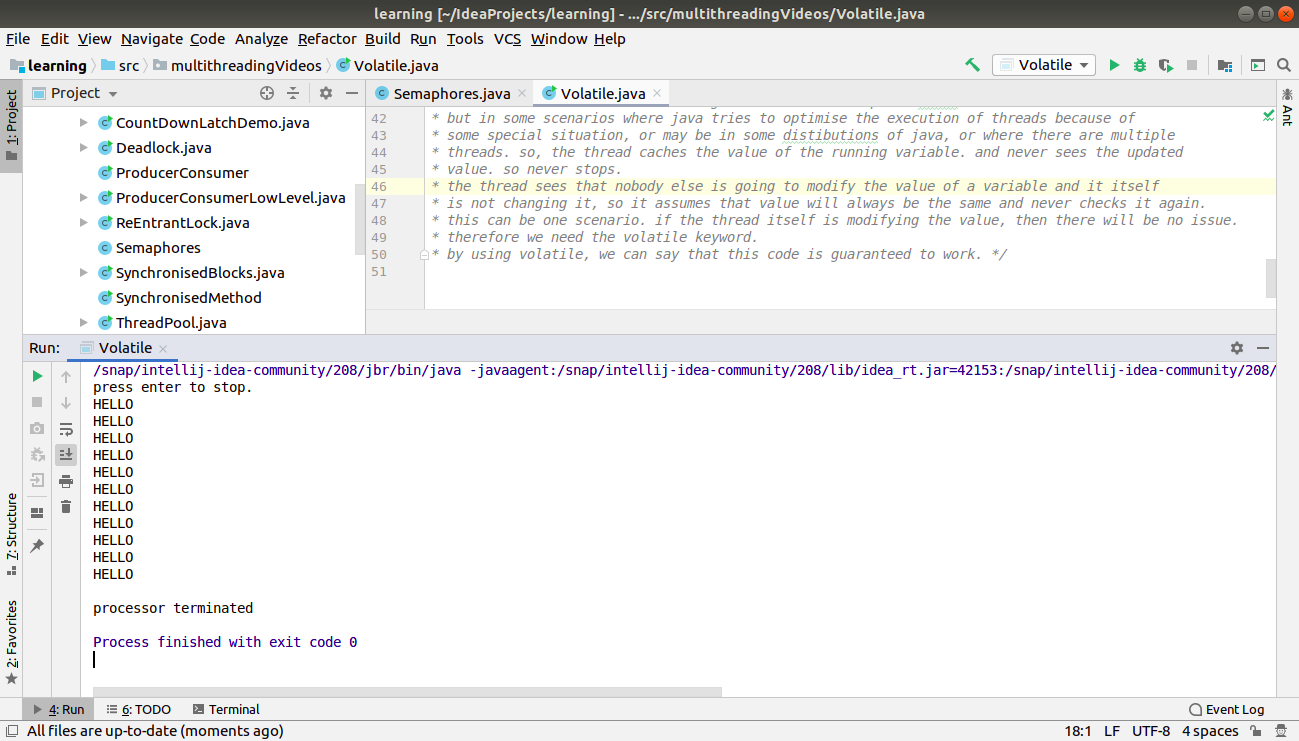
*\* the thread sees that nobody else is going to modify the value of a variable and it itself*

*\* is not changing it, so it assumes that value will always be the same and never checks it again.*

*\* this can be one scenario. if the thread itself is modifying the value, then there will be no issue.*

*\* therefore we need the volatile keyword.*

*\* by using volatile, we can say that this code is guaranteed to work. \*/*



1. **Write a program to create a thread using Thread class and Runnable interface each.**

=============== THREAD CLASS ===============

**package** multithreadingVideos;

**import** javax.management.relation.RoleUnresolved;

**class** Runner **extends** Thread{

**static int** *count* = 0;

**int id**;

Runner(){

**id** = ++*count*;

}

**public void** run(){

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

System.***out***.println(**"HELLO "** + i + **" -- from thread "** + **id**);

**try**{

Thread.*sleep*(500);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

}

}

}

**public class** ThreadUsingThreadClass {

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

Runner r1 = **new** Runner();

Runner r2 = **new** Runner();

*// we call start method because it runs the run method in a nee thread*

*// if we call run directly, it calls the run method in the main thread itself.*

r1.start();

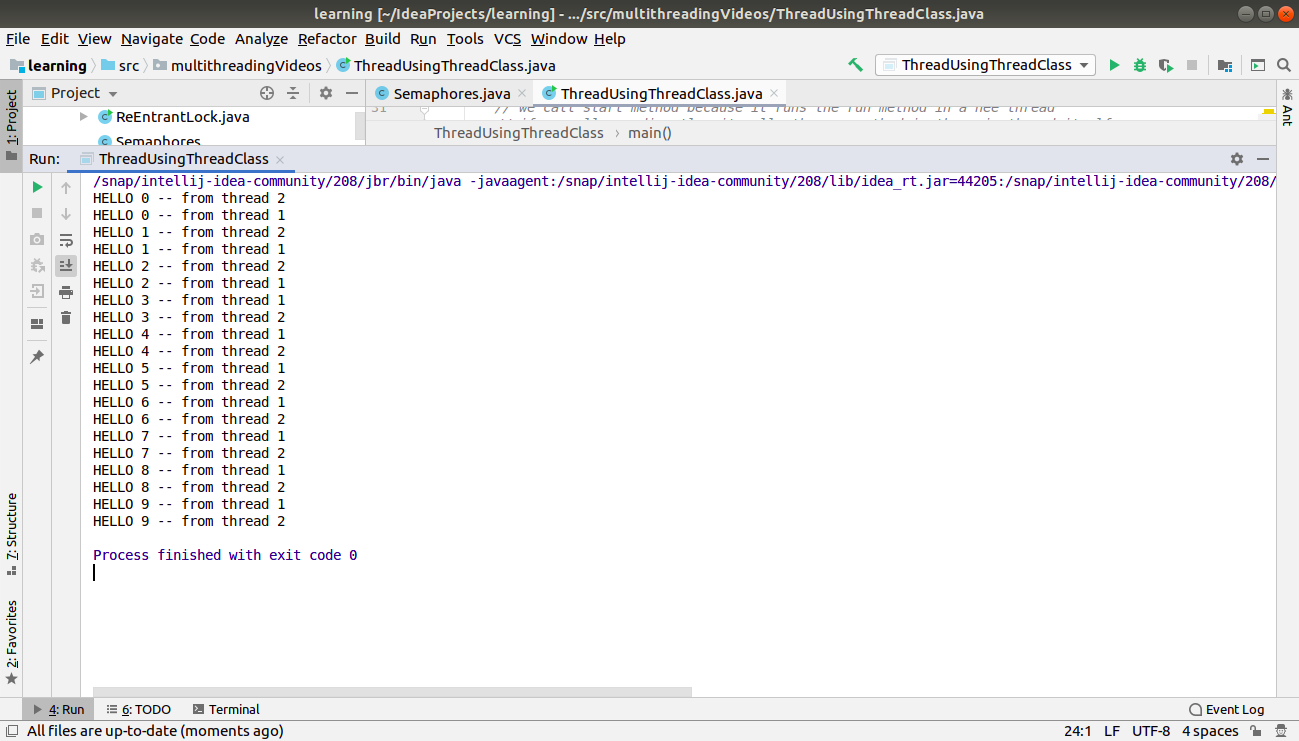
r2.start();

*// i observed that, i started r1 first but still r2 can start early.*

*// this is awesome.*

}

}



============ RUNNABLE INTERFACE ===============

**package** multithreadingVideos;

**class** RunnerUp **implements** Runnable{

**static int** *count* = 0;

**int id**;

RunnerUp(){

**id** = ++*count*;

}

**public void** run(){

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

System.***out***.println(**"HELLO "** + i + **" -- from thread "** + **id**);

**try**{

Thread.*sleep*(500);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

}

}

}

**public class** ThreadUsingRunnable {

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

*// start is a method oof thread class, not runnable.*

*// we cant call it on RunnerUp object here.*

*// so we need to create an obj of Thread class always if we use Runnable.*

Thread t1 = **new** Thread(**new** RunnerUp());

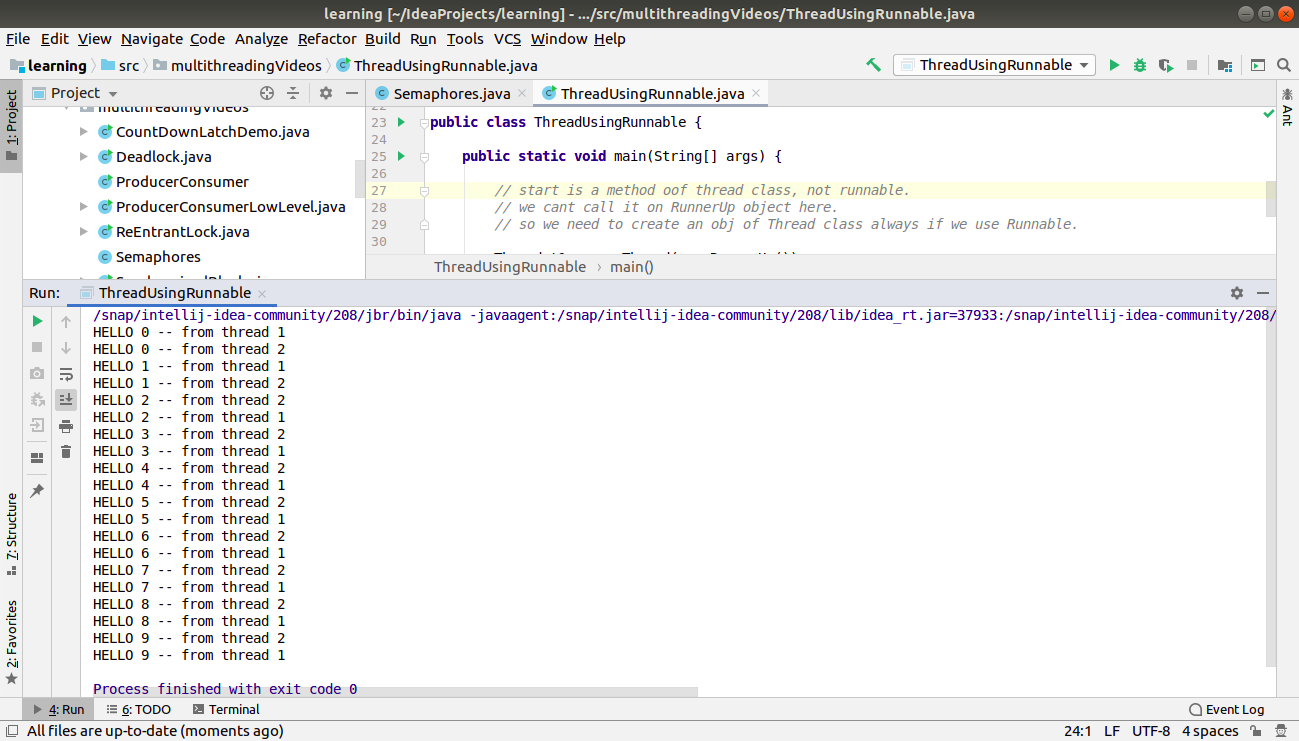
Thread t2 = **new** Thread(**new** RunnerUp());

t1.start();

t2.start();

}

}



1. **Write a program using synchronization block and synchronization method**

=========== SYNCHRONISATION BLOCKS ============

**package** multithreadingVideos;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Random;

**class** Worker {

**private** Random **random** = **new** Random();

**private** List<Integer> **list1** = **new** ArrayList<Integer>();

**private** List<Integer> **list2** = **new** ArrayList<Integer>();

Object **lock1** = **new** Object();

Object **lock2** = **new** Object();

*// rather than locking the shared object itself, its better to create a different lock object and use that.*

*// its a good practice. otherwise u may get confusion or java may create problem while optimising also.*

*// thats why we make synchronised block.*

**public void** process(){

**for**(**int** i=0; i<1000; i++){

stageOne();

stageTwo();

}

}

*// public synchronized void stageOne(){*

**public void** stageOne(){

**synchronized**(**lock1**){

**try**{

Thread.*sleep*(1);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

**list1**.add(**random**.nextInt());

}

}

*// public synchronized void stageTwo(){*

**public void** stageTwo(){

**synchronized**(**lock2**){

**try**{

Thread.*sleep*(1);

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

**list2**.add(**random**.nextInt());

}

}

**public void** main() {

System.***out***.println(**"Starting ..."**);

**long** start = System.*currentTimeMillis*();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

process();

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

process();

}

});

t1.start();

t2.start();

**try** {

t1.join();

t2.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**long** end = System.*currentTimeMillis*();

System.***out***.println(**"time taken - "** + (end - start));

System.***out***.println(**"list 1 size - "** + **list1**.size());

System.***out***.println(**"list 2 size - "** + **list2**.size());

}

}

*/\**

*\* an awesome thing is that, if i make the 2 methods synchronised here, then the thread will acquire*

*\* intrinsic lock on the whole worker object. so, if one thread is using the stageOne() then other*

*\* thread cant use stageTwo() also even though they are using the different data objects - list1-2.*

*\**

*\* so, that's why we provide different locks to both the methods by using synchronised block.*

*\* now, no 2 threads can use the same function simultaneously but they can access the 2 different*

*\* functions together.*

*\**

*\**

*\* its really awesome!!!!........*

*\* \*/*

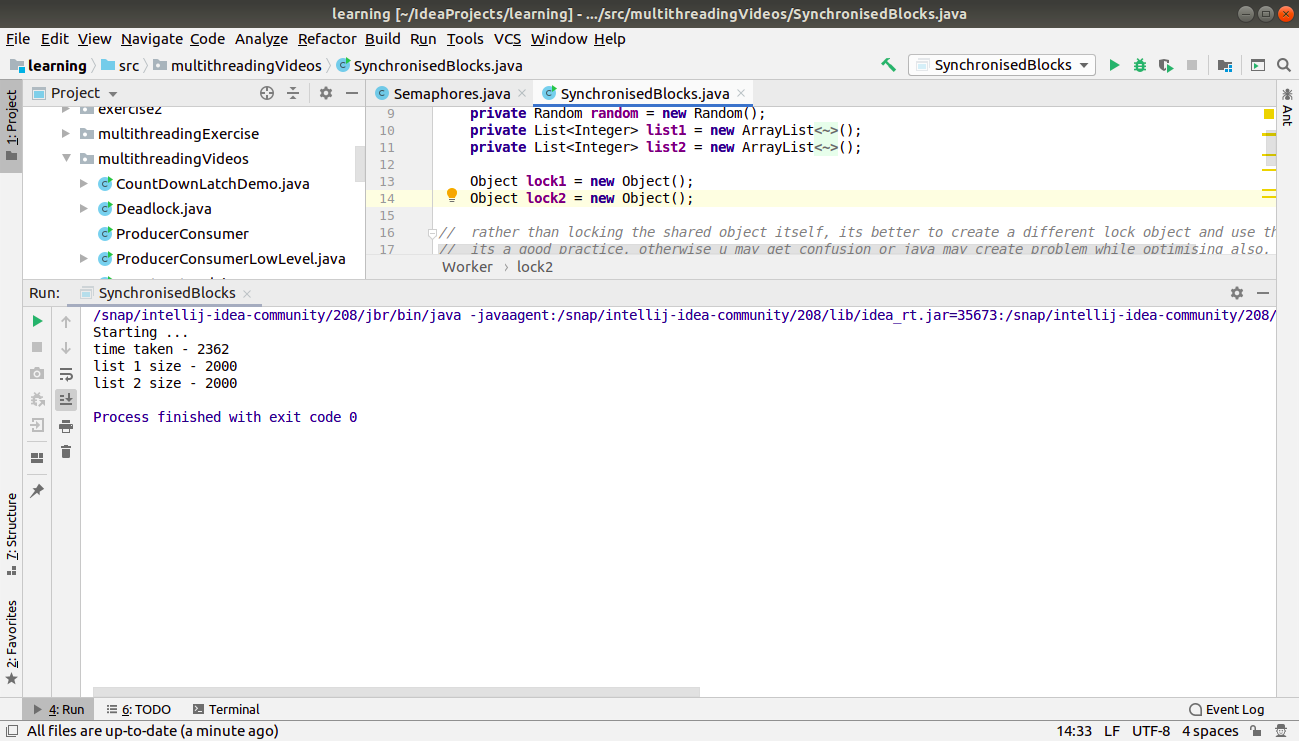
**public class** SynchronisedBlocks {

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

**new** Worker().main();

}

}



================ SYNCHRONIZED METHOD ===========

**package** multithreadingVideos;

**public class** SynchronisedMethod {

**private int count** = 0;

**public synchronized void** increment(){

**count**++;

}

**public void** doWork(){

Thread t1 = **new** Thread( **new** Runnable(){

**public void** run()

{

**for**(**int** i=0; i<10000; i++){

increment();

}

}

});

Thread t2 = **new** Thread( **new** Runnable(){

**public void** run()

{

**for**(**int** i=0; i<10000; i++){

increment();

}

}

});

t1.start();

t2.start();

**try**{

t1.join();

t2.join();

}**catch**(InterruptedException ie){

ie.printStackTrace();

}

System.***out***.println(**"count = "** + **count**);

}

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

SynchronisedMethod sv = **new** SynchronisedMethod();

sv.doWork();

}

}

*/\**

*here there are 2 threads running simultaneously and accessing the same value.*

*the step count++ actually is count = count+1 which involves 3 steps -*

*see the current value of count*

*increment it*

*and then write it again.*

*but, one thread reads the value 100 and increments it. before it writes it, another thread also reads it 100*

*several times and make it to be 110. now the previous thread makes it 101 again when its turn comes.*

*so, they overwrite each other's work. so we dont get the proper output. sometimes, we get.*

*so, the solution is to make this count++ operation atomic.*

*we can do this by using synchronised keyword with a function.*

*volatile will not work here, because caching is not the problem. the problem is interleaving of threads.*

*we convert the count++ statement into a function and makes that synchronised.*

*so, while one thread is executing the function, another thread could not and will*

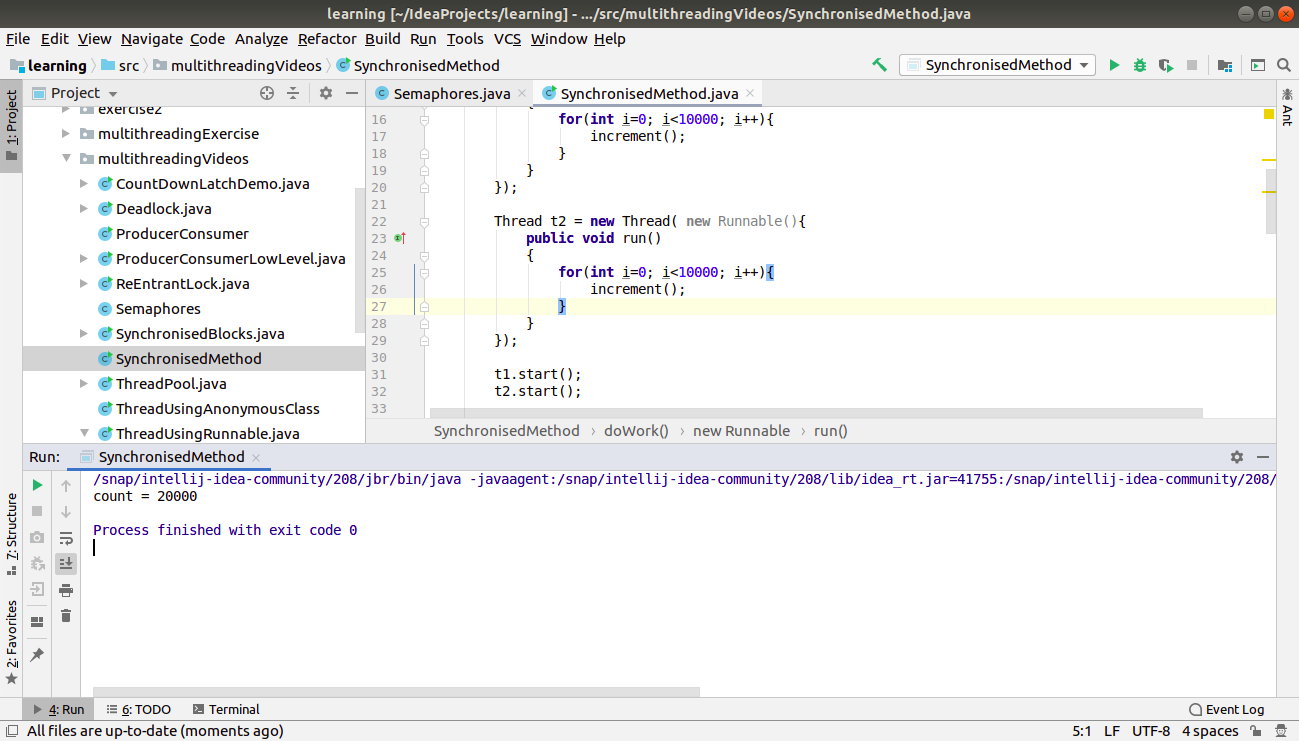
*have to wait for that.*

*every object in java has an intrinsic lock or mutex. synchronised acquires that lock and*

*releases on exiting the method increment. so, its done.*

*so, synchronised method can be called by only one thread at a time.*

*\*/*



1. **Write a program to create a Thread pool of 2 threads where one Thread will print even numbers and other will print odd numbers.**

**package** multithreadingExercise;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

**class** EvenThread **implements** Runnable{

**public void** run(){

System.***out***.println(**"Starting an even Thread "**);

**for**(**int** i=0; i<=20; i+=2){

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println(**"Completed an even Thread "**);

}

}

**class** OddThread **implements** Runnable{

**public void** run(){

System.***out***.println(**"Starting an odd Thread "**);

**for**(**int** i=1; i<20; i+=2){

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

System.***out***.println(**"Completed an odd Thread "**);

}

}

**public class** Question4threadPool {

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

ExecutorService executor = Executors.*newFixedThreadPool*(2);

*// executor starts his own separate thread to manage the thread pool.*

**for**(**int** i=0; i<3; i++){

executor.submit(**new** OddThread());

executor.submit(**new** EvenThread());

}

executor.shutdown();

*// this method prevents the submission of new tasks and allows the already submitted ones to*

*// complete.*

System.***out***.println(**"all threads submitted "**);

**try** {

executor.awaitTermination(1, TimeUnit.***DAYS***);

*// blocks until all the threads have completed execution or the timeout occurs.*

*// or if any thread is interrupted.*

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**"all threads completed"**);

}

}

*/\**

*the use of thread pool is to save the overhead. starting a new thread gives an overhead.*

*To minimize this overhead, we limit the number of threads that can run simultaneously.*

*this is used by .. for ex - amazon during mobile sale.*

*every user gets his own thread but during sale the number of threads will go to lakhs and the*

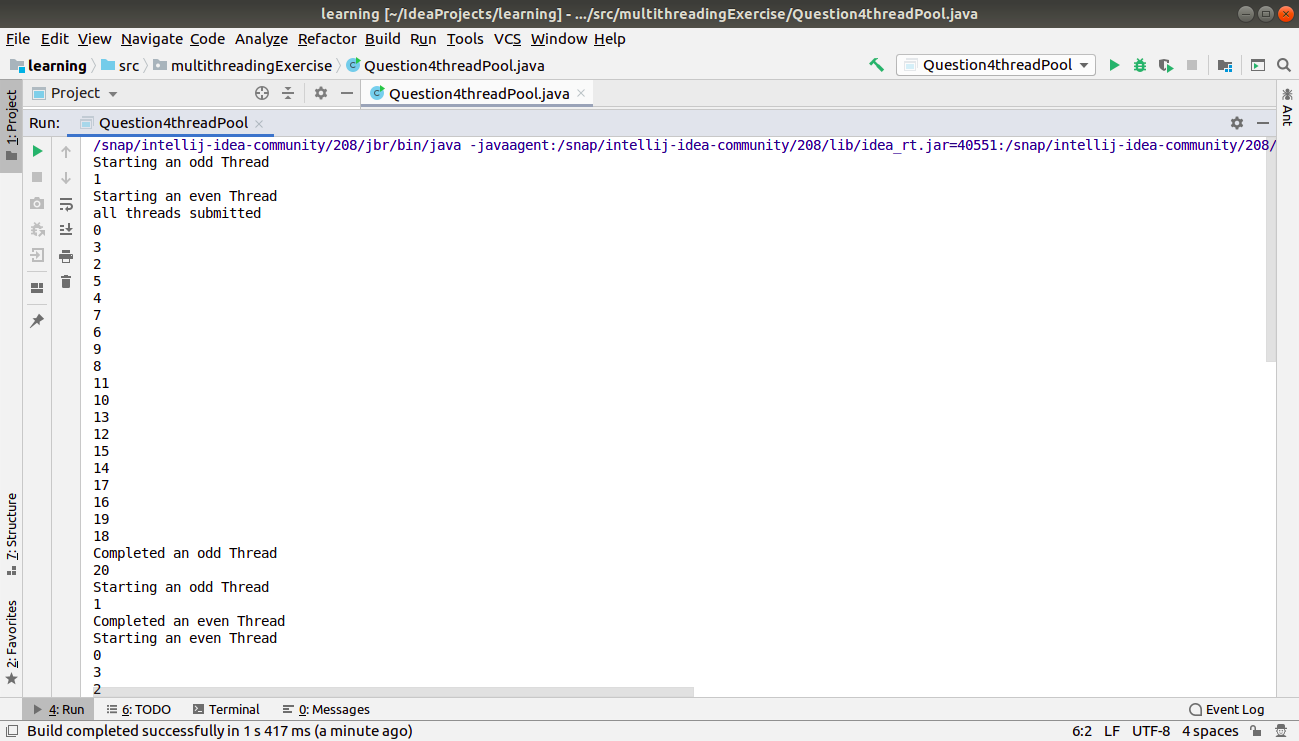
*server would choke down. so they limit the number of threads by using a thread pool. and now*

*only a fixed number of threads are created and executed on first come first serve basis.*

*rest keep waiting for their turn.*

*this is awesome.*

*\*/*



1. **Write a program to demonstrate wait and notify methods.**

**package** multithreadingVideos;

**import** java.util.Scanner;

**class** Processor4{

**public void** producer() **throws** InterruptedException {

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

System.***out***.println(**"producer running"**);

wait();

*// wait method is a method of the Object class. every object has this method.*

*// it can be called only from inside a synchronised block.*

*// we can give the time interval also.*

System.***out***.println(**"producer resumed"**);

}

}

**public void** consumer() **throws** InterruptedException {

Scanner sc = **new** Scanner(System.***in***);

Thread.*sleep*(2000);

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

System.***out***.println(**"consumer running"**);

System.***out***.println(**"waiting for return key pressed"**);

sc.nextLine();

System.***out***.println(**"return key pressed"**);

notify();

*// notify can also be called only from inside a synchronised block.*

*// notify informs the other thread that it can start now but actually it cant start untill*

*// this block is completed and we release the lock. so, it will wait for another 5 sec for*

*// the lock and then other thread acquires the lock and starts.*

Thread.*sleep*(5000);

}

}

}

**public class** WaitAndNotify {

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

Processor4 pc = **new** Processor4();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

**try** {

pc.producer();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

**try** {

pc.consumer();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

t1.start();

t2.start();

**try** {

t1.join();

t2.join();

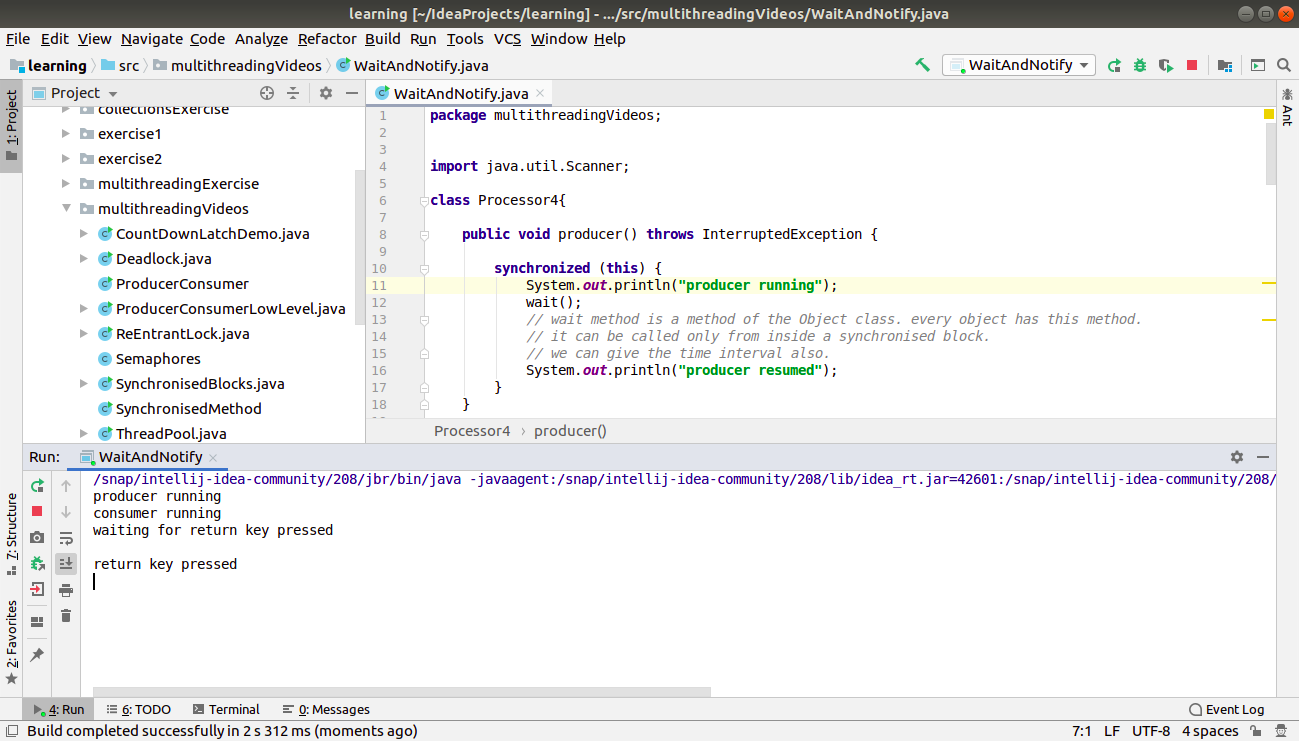
} **catch** (InterruptedException e) {

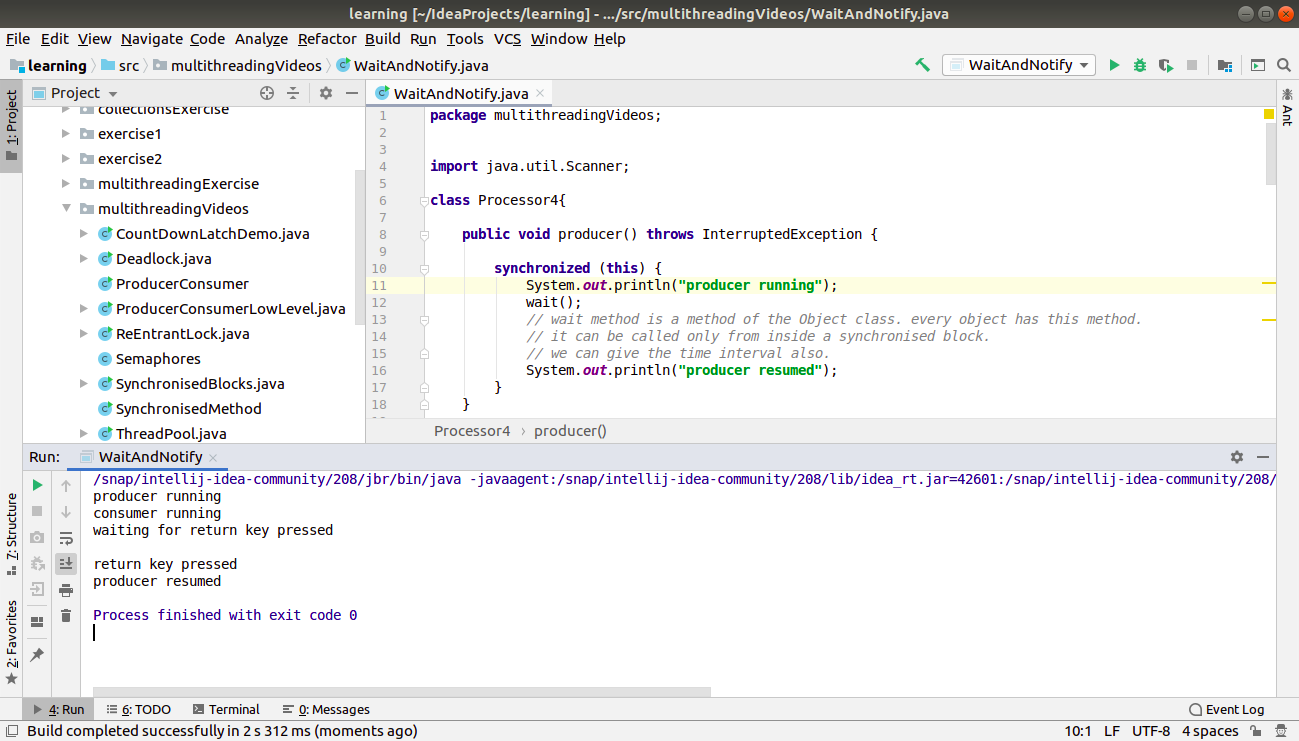
e.printStackTrace();

}

}

}





1. **Write a program to demonstrate sleep and join methods.**

**package** multithreadingExercise;

**public class** SleepAndJoin {

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

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"started runnning 1"**);

**try** {

Thread.*sleep*(5000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**"ended thread 1"**);

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

System.***out***.println(**"started runnning 2"**);

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**"ended thread 2"**);

}

});

t1.start();

t2.start();

**try** {

t1.join();

System.***out***.println(**"finished thread 1"**);

t2.join();

System.***out***.println(**"finished thread 2"**);

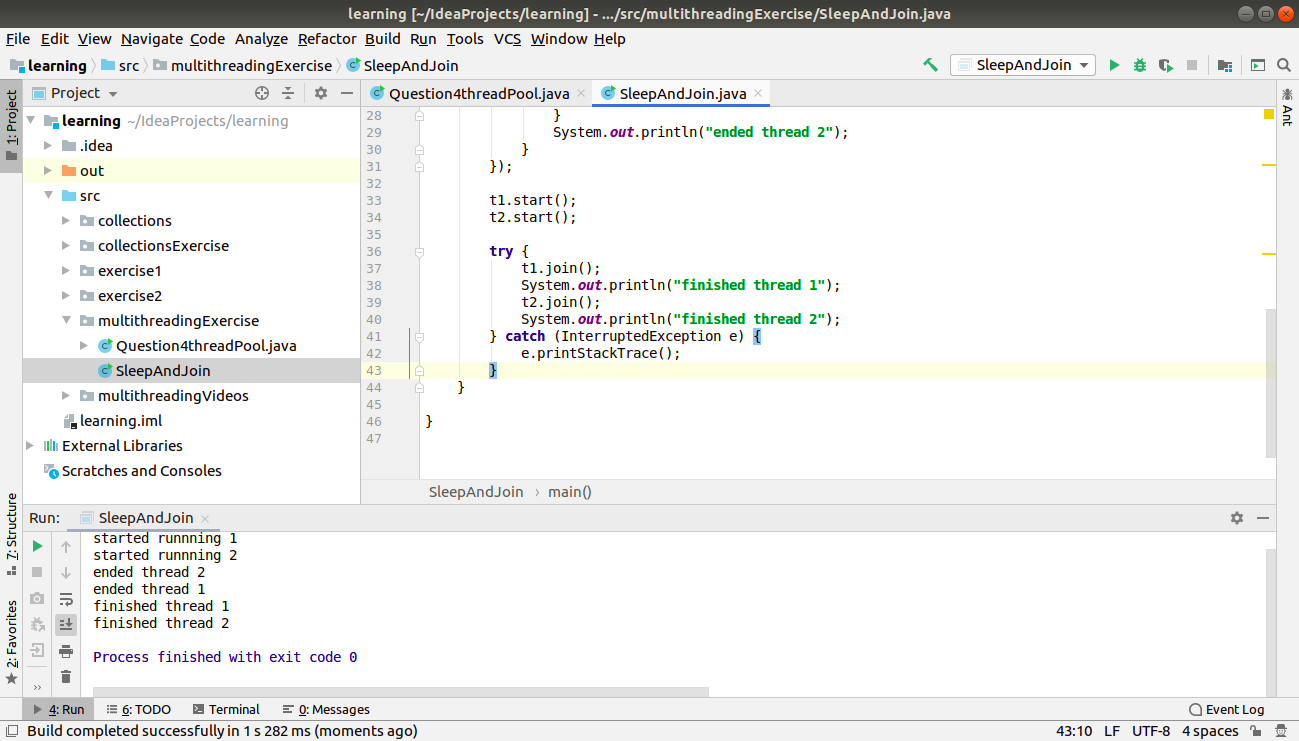
} **catch** (InterruptedException e) {

e.printStackTrace();

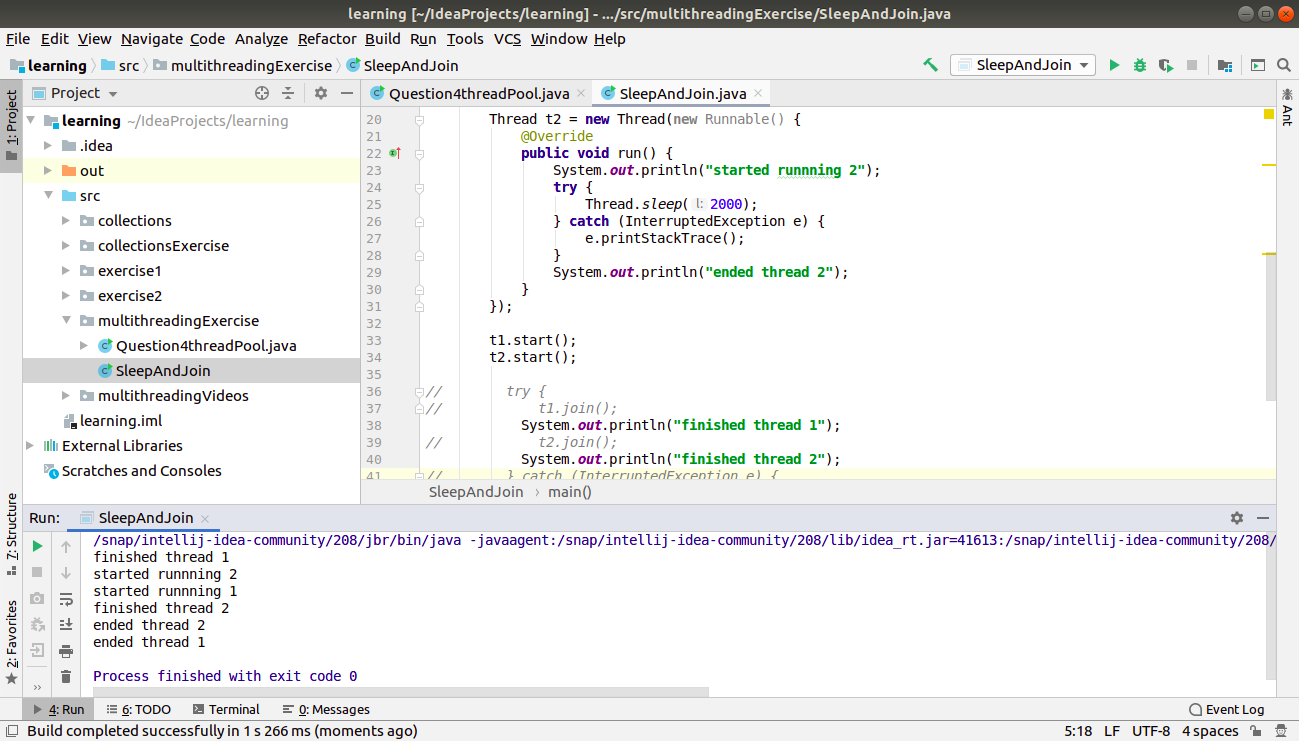
}

}

}



============== OUTPUT WITHOUT JOIN METHOD ============



1. **Run a task with the help of callable and store it's result in the Future.**

**package** multithreadingVideos;

**import** java.io.IOException;

**import** java.util.Random;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public class** CallableAndFuture {

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

ExecutorService executor = Executors.*newCachedThreadPool*();

Future<Integer> future = executor.submit(**new** Callable<Integer>() {

@Override

**public** Integer call() **throws** Exception {

System.***out***.println(**"thread started "**);

*// doing some work*

Random random = **new** Random();

**int** duration = random.nextInt(2000);

**try** {

Thread.*sleep*(duration);

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

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

**return** duration;

}

});

executor.shutdown();

**try** {

System.***out***.println(**"Result is: "** + future.get());

} **catch** (InterruptedException e) {

e.printStackTrace();

} **catch** (ExecutionException e) {

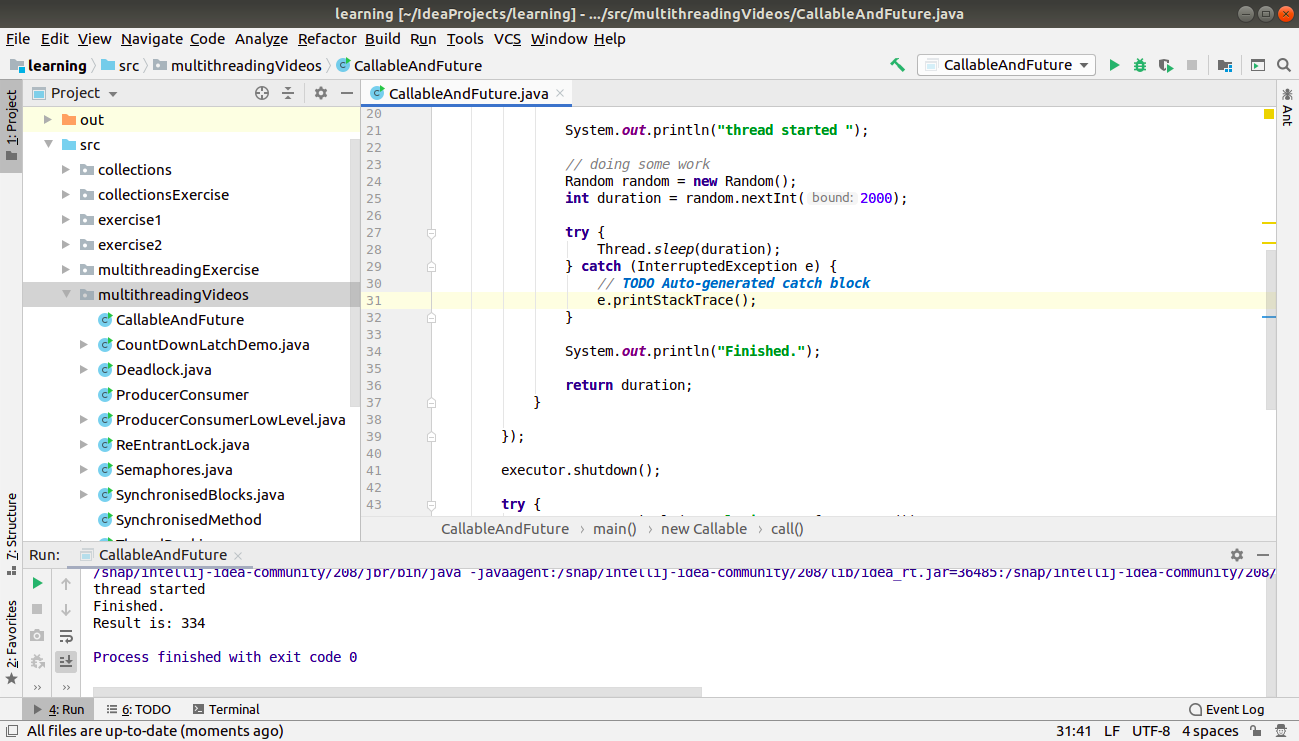
IOException ex = (IOException) e.getCause();

System.***out***.println(ex.getMessage());

}

}

}



1. **Write a program to demonstrate the use of semaphore**

**package** multithreadingVideos;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Semaphore;

**import** java.util.concurrent.TimeUnit;

**class** Connection{

**private static** Connection *instance* = **new** Connection();

**private int connections** = 0;

*// we want only 10 connections to be handles at a time.*

**private static** Semaphore *sem* = **new** Semaphore(10);

**private** Connection(){

*// private contsructor*

}

**public static** Connection getInstance(){

**return** *instance*;

}

**public void** connect(){

**try** {

*// if eligible to make connection.*

*sem*.acquire();

} **catch** (InterruptedException e1) {

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

e1.printStackTrace();

}

**try** {

*// make the connection*

doConnect();

} **finally** {

*// work completed, release the connection.*

*sem*.release();

}

}

**public void** doConnect(){

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

**connections**++;

System.***out***.println(**"Current connections: "** + **connections**);

}

*// doing some work with the connection....*

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

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

**connections**--;

System.***out***.println(**"released lock on connection"**);

}

}

}

**public class** Semaphores {

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

*// Semaphore sem = new Semaphore(1);*

ExecutorService executor = Executors.*newCachedThreadPool*();

**for**(**int** i=0; i<200; i++){

executor.submit(**new** Runnable() {

@Override

**public void** run() {

Connection.*getInstance*().connect();

}

});

}

executor.shutdown();

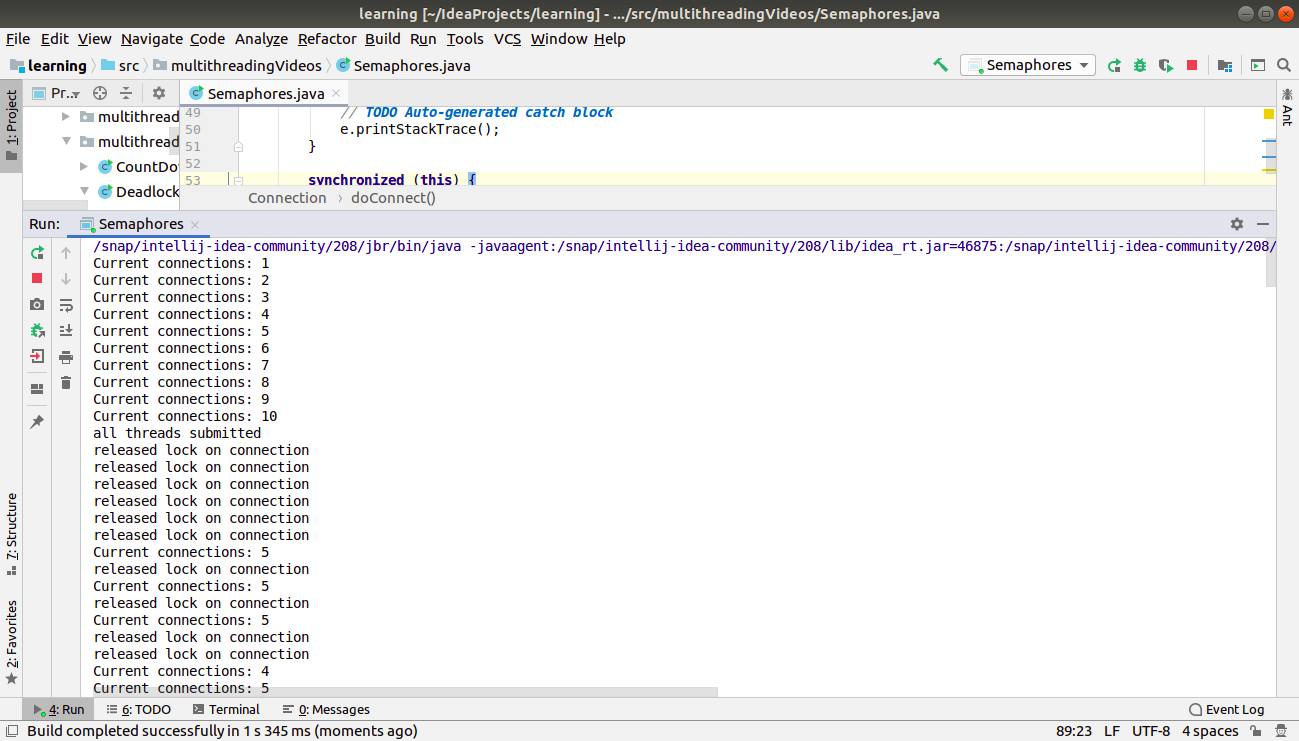
System.***out***.println(**"all threads submitted"**);

executor.awaitTermination(1, TimeUnit.***DAYS***);

System.***out***.println(**"all threads completed"**);

}

}



Please run to see full output ========================

1. **Write a program to demonstrate the use of CountDownLatch**

**package** multithreadingVideos;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**class** Processor2 **implements** Runnable{

**private int id**;

**private** CountDownLatch **latch**;

Processor2(CountDownLatch latch, **int** id){

**this**.**latch** = latch;

**this**.**id** = id;

}

**public void** run(){

System.***out***.println(**"Starting Thread "** + **id**);

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**"Completed Thread "**+ **id**);

**latch**.countDown(); *// decrement the value of latch*

}

}

**public class** CountDownLatchDemo {

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

ExecutorService executor = Executors.*newFixedThreadPool*(3);

CountDownLatch latch = **new** CountDownLatch(3);

**for**(**int** i=0; i<3; i++){

executor.submit(**new** Processor2(latch, i));

}

System.***out***.println(**"all threads submitted"**);

**try** {

latch.await();

*// Causes the current thread to wait until the latch has counted down to zero,*

*// unless the thread is interrupted, or the specified waiting time elapses.*

} **catch** (InterruptedException e) {

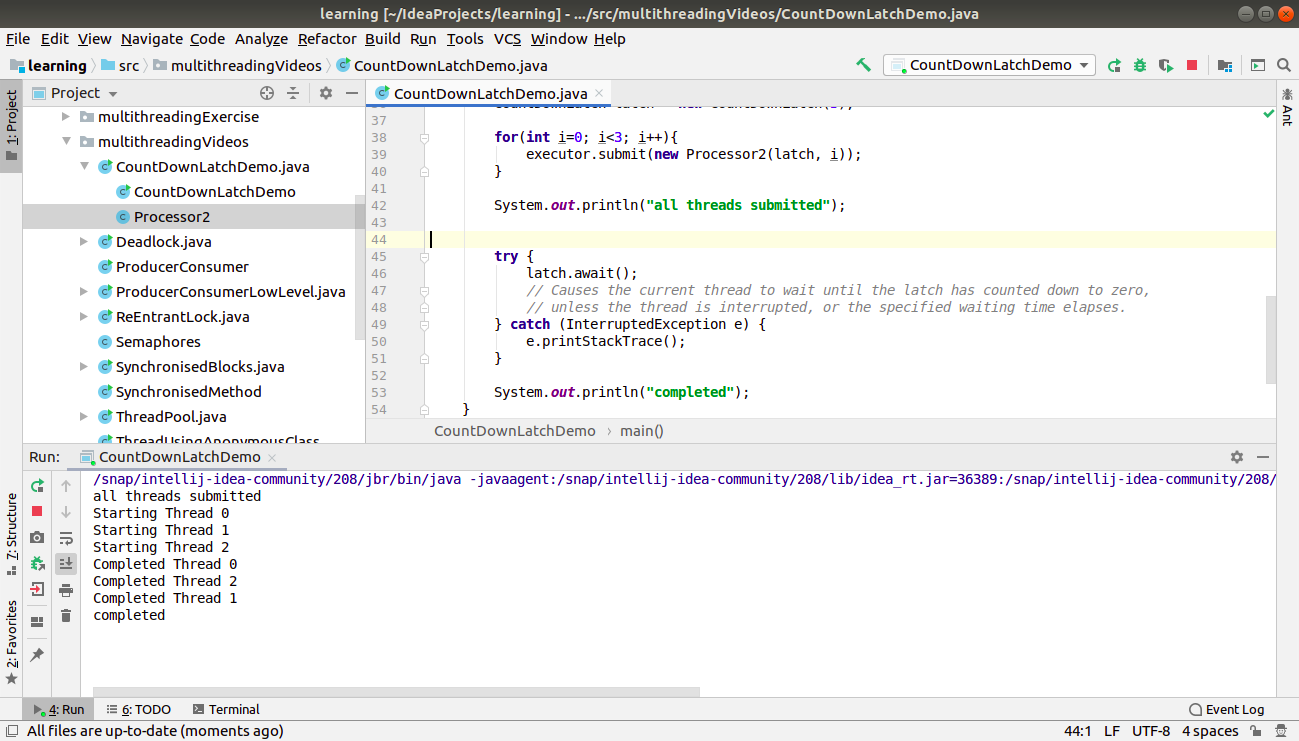
e.printStackTrace();

}

System.***out***.println(**"completed"**);

}

}



1. **Write a program which creates deadlock between 2 threads**

**package** multithreadingVideos;

**import** java.util.Random;

**import** java.util.Scanner;

**import** java.util.concurrent.locks.Condition;

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

**class** Account{

**private int balance** = 10000;

**public void** deposit(**int** amount){

**balance** += amount;

}

**public void** withdraw(**int** amount){

**balance** -= amount;

}

**public int** getBalance(){

**return balance**;

}

**public static void** transfer(Account a1, Account a2, **int** amount){

a1.withdraw(amount);

a2.deposit(amount);

}

}

**class** Runner2{

**private** Account **acc1** = **new** Account();

**private** Account **acc2** = **new** Account();

**private** Lock **lock1** = **new** ReentrantLock();

**private** Lock **lock2** = **new** ReentrantLock();

**public void** acquireLocks(Lock firstLock, Lock secondLock) **throws** InterruptedException {

**while**(**true**){

**boolean** gotFirstLock = **false**;

**boolean** gotSecondLock = **false**;

**try**{

gotFirstLock = firstLock.tryLock();

gotSecondLock = secondLock.tryLock();

}**finally**{

**if**(gotFirstLock && gotSecondLock)

**return**;

**else if**(gotFirstLock)

firstLock.unlock();

**else if**(gotSecondLock)

secondLock.unlock();

}

*// locks not acquired*

Thread.*sleep*(1);

}

}

**public void** firstThread() **throws** InterruptedException{

*// transfer from acc1 to acc2*

*// lock1.lock();*

*// lock2.lock();*

acquireLocks(**lock1**,**lock2**);

**try**{

Random random = **new** Random();

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

Account.*transfer*(**acc1**, **acc2**, random.nextInt(100));

}**finally** {

**lock2**.unlock();

**lock1**.unlock();

}

}

**public void** secondThread() **throws** InterruptedException{

*// transfer from acc2 to acc1*

*// ======= TO CRETE DEADLOCK, USE THIS CODE =======*

*// lock2.lock();*

*// lock1.lock();*

*// WE NEED TO ACQUIRE THE LOCKS IN SAME ORDER IN BOTH THREADS TO AVOID DEADLOCK.*

*// lock1.lock();*

*// lock2.lock();*

acquireLocks(**lock2**, **lock1**);

**try**{

Random random = **new** Random();

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

Account.*transfer*(**acc2**, **acc1**, random.nextInt(100));

}**finally** {

**lock2**.unlock();

**lock1**.unlock();

}

}

**public void** finished(){

System.***out***.println(**"Account 1 balance - "** + **acc1**.getBalance());

System.***out***.println(**"Account 2 balance - "** + **acc2**.getBalance());

System.***out***.println(**"Total balance - "** + (**acc1**.getBalance() + **acc2**.getBalance()));

}

}

**public class** Deadlock {

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

**final** Runner2 runner = **new** Runner2();

Thread t1 = **new** Thread(**new** Runnable() {

**public void** run() {

**try** {

runner.firstThread();

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

**public void** run() {

**try** {

runner.secondThread();

} **catch** (InterruptedException e) {

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

e.printStackTrace();

}

}

});

t1.start();

t2.start();

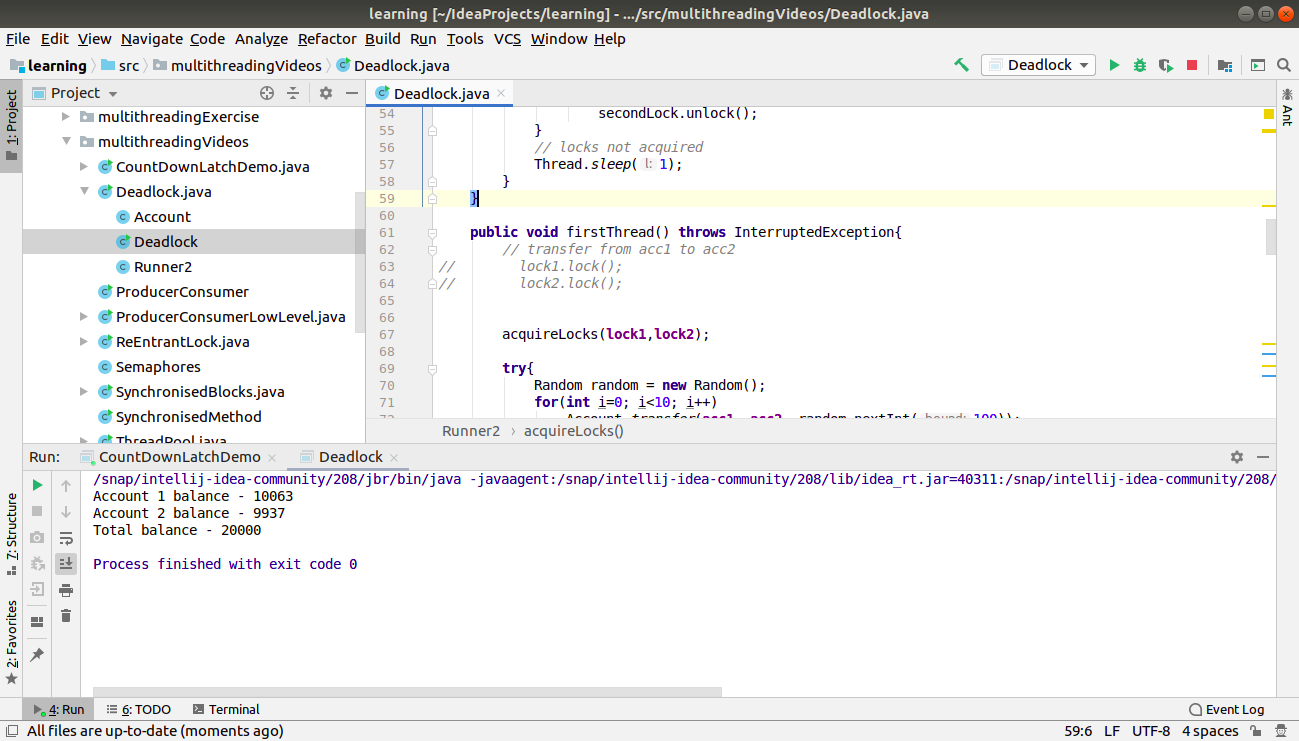
t1.join();

t2.join();

runner.finished();

}

}



===========================================================

===========================================================