|  |  |
| --- | --- |
|  | 1. Implement three classes: Storage, Counter, and Printer.   The Storage class should store an integer.  The Counter class should create a thread that starts counting from 0 (0, 1, 2, 3 ...) and stores each value in the Storage class.  The Printer class should create a thread that keeps reading the value in the Storage class and printing it.  Write a program that creates an instance of the Storage class and sets up a Counter and a Printer object to operate on it. |
|  | Modify the program from the previous exercise to ensure that each number is printed exactly once, by adding suitable synchronization.  **Storage.java**  public class Storage{        private int  i;        private boolean printed = true;        public void setValue(int i){              this.i = i;        }        public int getValue(){              return this.i;        }        public boolean isPrinted() {              return printed;        }        public void setPrinted(boolean p) {              printed = p;        }  }  **Counnter.java**  public class Counter implements  Runnable{          private Storage st;        public Counter(Storage store){              st = store;        }        @Override        public void run() {              synchronized(st) {                    for(int i = 0 ; i < 10; i++){                          while(!st.isPrinted()) {              //loop to take care of spontaneous wake-ups                             try {                                st.wait();                             } catch(Exception e) {  }                          }                          st.setValue(i);                          st.setPrinted(false);                          st.notify();                    }              }        }    }  **Printer.java**  public class Printer implements Runnable{    private Storage st;      public Printer(Storage st){              this.st = st;        }     @Override   public void run() {              synchronized(st) {                    for(int i = 0; i < 10; i++) {                          while(st.isPrinted()) {              //loop to take care of spontaneous wake-ups                                try {                                      st.wait();                                } catch(Exception e) {  }                          }                          System.out.println(Thread.currentThread().getName() + " " + st.getValue());                          st.setPrinted(true);                          st.notify();                    }              }        }    }  **Test.java**  public class Test {        public static void main(String[] args) {              Storage st = new Storage();              Counter c = new Counter(st);              Printer p = new Printer(st);              new Thread(c,"Counter").start();  //start the counter              new Thread(p,"Printer").start();   //start the printer        }  }   1. Write a program TestThreadMany.java that takes a positive integer n and creates exactly n threads that print out their own name. Here is a sample execution.   Enter number of threads you want? 4  Hello, I am Thread #1  Hello, I am Thread #2  Hello, I am Thread #3  Hello, I am Thread #4   1. Write a program to print "Good morning" and "Welcome" continuously on the screen in Java using threads.   class Thread1 extends Thread{  public void run(){  while (true){  System.out.println("Welcome");  }  }  }  class Thread2 extends Thread {  public void run() {  while (true) {  System.out.println("Good morning");  }  }  }  public class CWH {  public static void main(String[] args) {  Thread1 t1= new Thread1();  Thread2 t2= new Thread2();  t1.start();  t2.start();  }  }   1. Add a step method in the welcome thread of question 3 to delay its execution for 200ms.   class Thread1 extends Thread{  public void run(){  try {  Thread.sleep(200);  }  catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("Welcome");  }  }  class Thread2 extends Thread {  public void run() {  System.out.println("Good morning");  }  }  public class CWH {  public static void main(String[] args) {  Thread1 t1= new Thread1();  Thread2 t2= new Thread2();  t1.start();  t2.start();  }  }   1. Demonstrate gerPriority() and setPriority() methods in Java threads.   class Thread1 extends Thread{  public void run(){  System.out.println("Welcome");  }  }  class Thread2 extends Thread {  public void run() {  System.out.println("Good morning");  }  }  public class CWH {  public static void main(String[] args) {  Thread1 t1= new Thread1();  Thread2 t2= new Thread2();  t1.start();  t2.start();  t1.setPriority(5);  t2.setPriority(1);  System.out.println(t1.getPriority());  System.out.println(t2.getPriority());  }  }   1. How do you get the state of a given thread in Java?   class Practice13 extends Thread{  public void run(){  while(true){  System.out.println("Good Morning!");  }  }  }  class Practice13b extends Thread{  public void run(){  // while(false){  // try {  // Thread.sleep(200);  // }  // catch (Exception e){  // System.out.println(e);  // }  // System.out.println("Welcome");  // }  }  }  public class cwh\_76\_practice13 {  public static void main(String[] args) {  Practice13 p1 = new Practice13();  Practice13b p2 = new Practice13b();  // p1.setPriority(6);  // p2.setPriority(9);  System.out.println(p1.getPriority());  System.out.println(p2.getPriority());  System.out.println(p2.getState());  // p1.start();  p2.start();  System.out.println(p2.getState());  System.out.println(Thread.currentThread().getState());  }  }   1. Kind of hospital simulation in which we have to control the accesses to each singular room.   Doctors can enter the room one at the time, and can enter only if no visitors are in.  A visitor, instead, can only access the room if no doctors are in it and a max of 4 more visitors are in.  public class Room {  private int visitors=0; //number of visitors in the room  private int doctors=0; //number of doctors in the room  public Room(){  }  public synchronized void friendVisit() throws InterruptedException{  if(visitors>4 || doctors>0)  //wait();  Sleep(5000);  visitors++;  }  public synchronized void exitFriend(){  visitors--;  notify();  }  public synchronized void doctorVisit() throws InterruptedException{  if(doctors>0 || visitors>0)  wait();  doctors++;  }  public synchronized void exitDoctor(){  --doctors;  notify();  }  public int getVisitors(){  return visitors;  }  public int getDoctors(){  return doctors;  }  }  Doctors and Visitors(the class it's called Friend) are threads  public class Friend extends Thread{  private Room room; //reference to the room  public Friend(Room room\_reference){  room=room\_reference;  }  public void run(){  try {  sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  room.friendVisit();  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  room.exitFriend();  }  }  Here's the doctor thread:  public class Doctor extends Thread{  private Room room; //reference to the room  public Doctor(Room room\_reference){  room=room\_reference;  }  public void run(){  try {  sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  room.doctorVisit();  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  room.exitDoctor();  }  }  Here's a Display thread to keep trace of the number of visitors and doctors:  public class Display extends Thread{  private Room room;  public Display(Room room\_reference){  room=room\_reference;  }  public void run(){  while(true)  {  try {  sleep(300);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  System.out.println("The room contains "+room.getDoctors()+  " doctors and "+room.getVisitors()+" visitors.");  }  }  }  And here's my main:  public class Demo {  public static void main(String[]args){  Room room=new Room();  Friend friend=new Friend(room);  Doctor doctor=new Doctor(room);  Display display=new Display(room);  display.start();  while(true){  if(new Random().nextBoolean()==true){  friend=new Friend(room);  friend.start();  }  if(new Random().nextInt(5)==3){  doctor=new Doctor(room);  doctor.start();  }  }  }  Solution  I think one of your mistakes is assuming that wait() will return only when the condition above it is satisfied:  if(doctors>0 || visitors>0)  wait();  You may return from this call to wait() with the condition in your if statement false. Perhaps try a while loop:  while (doctors>0 || visitors>0) {  wait();  }  (adding brackets, of course, because you know a lack of brackets is evillll.....)  There may be other problems - I've not yet fired up your code.   1. Synchronized code segment synchronized(this)   public class SyncThread implements Runnable {  private static int *count* ;  public SyncThread () {  *count* = 0 ;  }  @Override  public void run () {  */\*\**  *\* The synchronized(this) code segment is equivalent to synchronously locking the object of the method, equivalent to synchronized modifying non-static methods*  *\*/*  synchronized ( this ) {  for ( int i = 5 ; i > 0 ; i--) {  System. *out* .println(Thread. currentThread ().getName() + ":" + ( *count* ++)) ;  try {  Thread. sleep ( 500 ) ;  } catch (InterruptedException e) {  e.printStackTrace() ;  }  }  }  }  public static void main (String[] args) {  SyncThread st = new SyncThread() ;  Thread t1 = new Thread(st , "A" ) ;  Thread t2 = new Thread(st , "B" ) ;  t1.start() ;  t2.start() ;  }  }  Analysis: The synchronization lock is added to this. When multiple threads holding the object execute to the syncronized code segment at the same time, the lock mechanism will work at this time to ensure that only one thread can execute the synchronized code segment at the same time. If two threads hold two different objects, then this is two different objects respectively, which is not restricted, as shown below: public static void main(String[] args) { SyncThread st1 = new SyncThread(); SyncThread st2 = new SyncThread();  Thread t1 = new Thread(st1, "A"); Thread t2 = new Thread(st2, "B"); t1.start(); t2.start(); }   1. Synchronized code segment synchronized (A.class) and synchronized modified static method   /\*\*  \* In the design scenario, I hope that the value of count will be increased by 1, and then decreased by 1, in a method, and finally unchanged.\*/  public class SyncThread2 implements Runnable {  private static int count ;  public SyncThread2 () {  count = 0 ;  }  public static void addCount () {  System. out .println(Thread. currentThread ().getName() + ":" + (++ count )) ;  try {  Thread. sleep ( 500 ) ;  } catch (InterruptedException e) {  e.printStackTrace() ;  }  System. out .println(Thread. currentThread ().getName() + ":" + (-- count )) ;  }  /\*\*  \* When synchronized modifies a static method, it is equivalent to synchronously locking the class of this class, so that all objects of this class hold the lock of this method  \*/  public static synchronized void addCount2 () {  System. out .println(Thread. currentThread ().getName() + ":" + (++ count )) ;  try {  Thread. sleep ( 500 ) ;  } catch (InterruptedException e) {  e.printStackTrace() ;  }  System. out .println(Thread. currentThread ().getName() + ":" + (-- count )) ;  }  /\*\*  \* Synchronous modification of class is equivalent to locking the class of this class synchronously, so that all objects of this class hold the lock of this method  \*/  public static void addCount3 () {  synchronized (SyncThread2. class ) {  System. out .println(Thread. currentThread ().getName() + ":" + (++ count )) ;  try {  Thread. sleep ( 500 ) ;  } catch (InterruptedException e) {  e.printStackTrace() ;  }  System. out .println(Thread. currentThread ().getName() + ":" + (-- count )) ;  }  }  @Override  public void run () {  for ( int i = 5 ; i> 0 ; i--) {  //Asynchronous safety, the value of count does not change after each method execution cannot be completed in a multi-threaded concurrent environment  //addCount();  //Sync security  addCount2 () ;  }  }  public static void main (String[] args) {  SyncThread2 st1 = new SyncThread2() ;  SyncThread2 st2 = new SyncThread2() ;  Thread t1 = new Thread(st1 , "A" ) ;  Thread t2 = new Thread(st2 , "B" ) ;  t1.start() ;  t2. start() ;  }  }   1. Comprehensive exercise, simulating concurrent bank deposit and withdrawal   */\*\**  *\* Personal bank account*  *\**  *\*/*  public class Account {  private String accountNo ;  private double balance ;  Private Boolean In Flag = to false;  public the Account (String \_accountNo , Double \_balance) {  the this . AccountNo = \_accountNo ;  the this . balance = \_balance ;  }  public String getAccountNo () {  return accountNo ;  }  public void setAccountNo (String accountNo) {  this . accountNo = accountNo ;  }  public double getBalance () {  return balance ;  }  */\*\**  *\* Synchronized modification of a non-static method is equivalent to locking the object of the method synchronously, which is equivalent to the synchronized(this) code segment*  *\**  *\** ***@param*** *drawBalance*  *\*/*  public synchronized void drawBalance ( double drawBalance) {  if ( flag ) {  if ( balance > drawBalance) {  System. *out* .println(Thread. currentThread ().getName() + "The withdrawal is successful, and the money is being ejected :" + drawBalance) ;  balance -= drawBalance ;  flag = false;  this .notifyAll() ;  System. *out* .println (Thread. currentThread ().getName() + "The balance is:" + getBalance()) ;  } else {  System. *out* .println(Thread. currentThread ().getName() + "Failed to withdraw money, insufficient balance" ) ;  }  } else {  try {  wait() ;  } catch (InterruptedException e) {  e.printStackTrace() ;  }  }  }  */\*\**  *\* Synchronized modification of a non-static method is equivalent to synchronously locking the object of the method, equivalent to the synchronized(this) code segment*  *\**  *\** ***@param*** *depositBalance*  *\*/*  public synchronized void deposit ( double depositBalance) {  if ( flag ) {  try {  wait() ;  } catch (InterruptedException e) {  e.printStackTrace() ;  }  } else {  . The System *OUT* .println (. The Thread currentThread () .getName () + "deposits:" + depositBalance) ;  the this . Balance + = depositBalance ;  In Flag = to true;  the this .notifyAll () ;  the System. *OUT* .println (the Thread. CurrentThread ().getName() + "The balance is:" + getBalance()) ;  }  }  }    */\*\**  *\** Deposit *money thread*  *\** ***@author*** *andy*  *\**  *\*/*  public class StorerThread implements Runnable {  private Account account ;  private double depositBalance ;  public Account getAccount () {  return account ;  }  public void setAccount (Account account) {  this . account = account ;  }  public double getDepositBalance () {  return depositBalance ;  }  public void setDepositBalance ( double depositBalance) {  this . depositBalance = depositBalance ;  }  @Override  public void run () {  for ( int i = 0 ; i < 100 ; i++) {  account .deposit( depositBalance ) ;  }  }  }    */\*\**  *\* Withdraw money thread*  *\** ***@author*** *andy*  *\**  *\*/*  public class FetcherThread implements Runnable {  private Account account ;  private double drawBalance ;  public Account getAccount () {  return account ;  }  public void setAccount (Account account) {  this . account = account ;  }  public double getDrawBalance () {  return drawBalance ;  }  public void setDrawBalance ( double drawBalance) {  this . drawBalance = drawBalance ;  }  @Override  public void run () {  for ( int i = 0 ; i < 100 ; i++) {  account .drawBalance( drawBalance ) ;  }  }  }    */\*\**  *\* Simulated bank account concurrent deposit and withdrawal*  *\** ***@author*** *alex*  *\**  *\*/*  public class AccountSyncTest {  public static void main (String[] args) {  Account acc = new Account( "001" , 1000 ) ;  FetcherThread mt1 = new FetcherThread() ;  mt1.setAccount(acc) ;  mt1.setDrawBalance( 20 ) ;  Thread t1 = new Thread(mt1 , " Withdrawer 1" ) ;  Thread t2 = new Thread(mt1 Thread(mt2 , "money saver 1" ) ;  Thread t4 = new Thread(mt2 , , " Withdrawer 2" ) ;  StorerThread mt2 = new StorerThread() ;  mt2.setAccount(acc) ;  mt2.setDepositBalance( 20 ) ;  Thread t3 = new t4.start() ;  } " 2" ) ;  t1.start() ;  t2.start() ;  t3.start() ;  }   1. Several state switching threads   import java.util.ArrayList;import java.util.List; import java.util.concurrent.CountDownLatch; /\*\*  \* @author alienware  \*  \*/ public class ListAdd2 { private volatile static List list = new ArrayList();  public void add(){ list.add("bjsxt"); } public int size(){ return list.size(); }  public static void main(String[] args) {  final ListAdd2 list2 = new ListAdd2(); final Object lock = new Object(); Thread t1 = new Thread(new Runnable() { @Override public void run() { try { synchronized (lock) { System.out.println("t1 start.."); for(int i = 0; i <10; i++){ list2.add(); System.out.println("Current thread:" + Thread.currentThread().getName() + "Added an element..."); Thread.sleep(500); if(list2.size() == 5){ System.out.println("Notification has been issued..."); //Because notify will not release the lock lock, after execution, although thread t2 is notified, t2 will not immediately obtain the lock lock, so t2 must wait for t1 to completely release the lock lock before t2 can continue execution lock.notify(); } } } } catch (InterruptedException e) { e.printStackTrace(); } } }, "t1");  Thread t2 = new Thread(new Runnable() { @Override public void run() { synchronized (lock) { System.out.println("t2 start.."); if(list2.size() != 5){ try { //Because wait will release the lock lock immediately, the t1 thread will immediately acquire the lock lock after the wait is executed. The main difference between sleep and wait is that sleep will not release the object lock, and wait will release the object lock. lock.wait(); } catch (InterruptedException e) { e.printStackTrace(); } } System.out.println("Current thread:" + Thread.currentThread().getName() + "Thread stopped receiving notification.."); throw new RuntimeException(); } } }, "t2"); t2.start(); t1.start();  }  }  Upgraded version to solve the problem that notify cannot release the object lock immediately  import java.util.ArrayList;import java.util.List; import java.util.concurrent.CountDownLatch; /\*\*  \* @author alienware  \*  \*/ public class ListAdd3 { private volatile static List list = new ArrayList();  public void add(){ list.add("bjsxt"); } public int size(){ return list.size(); }  public static void main(String[] args) {  final ListAdd3 list2 = new ListAdd3(); final Object lock = new Object(); final CountDownLatch cdl = new CountDownLatch(1); Thread t1 = new Thread(new Runnable() { @Override public void run() { try { //synchronized (lock) { System.out.println("t1 start.."); for(int i = 0; i <10; i++){ list2.add(); System.out.println("Current thread:" + Thread.currentThread().getName() + "Added an element..."); Thread.sleep(500); if(list2.size() == 5){ System.out.println("Notification has been issued..."); //Because notify will not release the lock lock, after execution, although thread t2 is notified, t2 will not immediately obtain the lock lock, so t2 must wait for t1 to completely release the lock lock before t2 can continue execution //If you want to notify thread t2 immediately and release the object lock, you can use CountDownLatch //lock.notify(); cdl.countDown(); } } //} } catch (InterruptedException e) { e.printStackTrace(); } } }, "t1");  Thread t2 = new Thread(new Runnable() { @Override public void run() { //synchronized (lock) { System.out.println("t2 start.."); if(list2.size() != 5){ try { //Because wait will release the lock lock immediately, the t1 thread will immediately acquire the lock lock after the wait is executed. The main difference between sleep and wait is that sleep will not release the object lock, and wait will release the object lock. //lock.wait(); //Use CountDownLatch to achieve cdl.await(); } catch (InterruptedException e) { e.printStackTrace(); } } System.out.println("Current thread:" + Thread.currentThread().getName() + "Thread stopped receiving notification.."); throw new RuntimeException(); } //} }, "t2"); t2.start(); t1.start();  }  } |
|  |  |