**Lista 12**

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

public **class** DinigRoom {

**private** Semaphore doorman;

**private** Semaphore[] chopstick;

**private** Philosopher[] philosopher;

**private** **final** int N;

public DinigRoom(int N) {

**this**.N = N;

doorman = **new** Semaphore(N - 1);

}

**private** void createForks() {

// Create the chopsticks, 1 per philosopher

chopstick = **new** Semaphore[N];

**for** (int f = 0; f < N; f++) {

// each chopstick is a single resource

chopstick[f] = **new** Semaphore(1, **true**);

}

}

**private** void invitePhilosophers() {

// Create the philosophers, pass in their chopsticks

philosopher = **new** Philosopher[N];

**for** (int ID = N - 1; ID >= 0; ID--) {

// determine my left neighbor's ID

int myneighbor = ID - 1;

**if** (myneighbor == -1) myneighbor = N - 1;

// Initialize each philosopher (no pun intended)

philosopher[ID] = **new** Philosopher(ID, chopstick[ID], chopstick[myneighbor]);

}

Philosopher.doorman = doorman;

}

**private** void startFeast() {

createForks();

invitePhilosophers();

System.out.println("Start dining!");

// Start the philosophers

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

philosopher[i].start();

}

// Wait for them to finish

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

**try** {

philosopher[i].join();

}

**catch** (InterruptedException ex) {}

}

System.out.println("Done eating.");

}

/\*\*

\* Dining simulations - puts philosophers by the table with their chopsticks

\*/

public static void main(String[] args) {

**new** DinigRoom(5).startFeast(); // five philosophers, five chopsticks

}

}

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

public **class** Philosopher **extends** Thread {

// Shared by each instance

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

**private** static int event = 0; // counting events during feast

public static Semaphore doorman;

// Philosopher private data

**private** int ID; // Unique philosopher ID

**private** Semaphore chopstickLeft; // Own chopstick

**private** Semaphore chopstickRight; // Nieghbour's chopstick

**private** int meals = 10; // Max meals to eat during banquet

public Philosopher(int ID, Semaphore chopstickLeft, Semaphore chopstickRight) {

**this**.ID = ID;

**this**.chopstickLeft = chopstickLeft;

**this**.chopstickRight = chopstickRight;

}

**private** void updateStatus(String str) {

System.out.printf("Event no:%3d | Philosopher of ID:%d %s\n",

++event, ID, str);

}

**private** void pause() {

**try** { sleep(random.nextInt(1000)); }

**catch** (InterruptedException e) {}

}

**private** void think() {

updateStatus("is thinking");

pause();

}

**private** void trytoeat() {

**if** (isAllowedToEat()) {

updateStatus("is hungry and is trying to pick up his chopsticks");

**try** { // Semaphore - waits on his own chopstick if necessary

chopstickLeft.acquire();

// He's picked up his own chopstick, now try and grab his neighbor's chopstick

**if** (!chopstickRight.tryAcquire()) { // Unsuccessful, guess he's fasting at the moment

updateStatus(" was not able to get neighbor's chopstick");

}

**else** { // Success! begins to eat

updateStatus("picked up his chopsticks and is eating meal no:" + (10 - --meals));

pause();

// Now put down the chopsticks

updateStatus("puts down his chopsticks");

}

}

**catch** (InterruptedException e) { updateStatus("was interrupted while waiting for his chopstick");

}

**finally** { // always puts his own chopstick back down

chopstickLeft.release();

chopstickRight.release();

doorman.release();

}

}

}

**private** boolean isAllowedToEat() {

**if** (doorman.tryAcquire()) {

updateStatus("has just gone into dinner room");

**return** **true**;

}

**else** {

updateStatus("can't go into dinner room");

**return** **false**;

}

}

/\*\*

\* Philosophize between all meals are consumed

\*/

@Override public void run() {

**while** (meals > 0) {

think();

trytoeat();

}

}

}

3.  
a)  
**class** IntCell {

**private** int n = 0;

public int getN() { **return** n; }

public void setN(int n) { **this**.n = n; }

public synchronized void increment() { setN(getN() + 1); }

}

**class** Count1 **extends** Thread {

static IntCell n = **new** IntCell();

@Override public void run() { **for** (int i = 0; i < 200000; i++) n.increment(); }

public static void main(String[] args) {

Count1 p = **new** Count1();

Count1 q = **new** Count1();

p.start();

q.start();

**try** {

p.join();

q.join();

}

**catch** (InterruptedException e) {}

System.out.println("The value of n is " + n.getN());

}

}

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

**class** IntCell {

**private** **int** n = 0;

**public** **int** getN() { **return** n; }

**public** **void** setN(**int** n) { **this**.n = n;

}

**class** Count **extends** Thread {

**static** IntCell *n* = **new** IntCell();

**private** **final** Semaphore semaphore = **new** Semaphore(1);

@Override **public** **void** run() {

**int** temp;

**for** (**int** i = 0; i < 200000; i++) {

**try** {

semaphore.acquire();

temp = *n*.getN();

*n*.setN(temp + 1);

semaphore.release();

}

**catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

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

Count p = **new** Count();

Count q = **new** Count();

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

p.start();

q.start();

**try** {

p.join();

q.join();

}

**catch** (InterruptedException e) {}

System.*out*.println("The value of n is " + *n*.getN());

System.*out*.println(System.*currentTimeMillis*() - start);

}

}

4.

BlockingQueue<Integer> blockingQueue = **new** ArrayBlockingQueue<>(N);

**Lista 13**

**import** scala.actors.Actor.\_

**import** scala.actors.Actor

object ClientServer {

def main(args: Array[String]) {

var clientArray = **new** Array[Actor](10)

**for** (i <- 1 to 9) clientArray(i) = **new** (A)SyncClient(i, (A)SyncServer)

(A)SyncServer.start

**for** (i <- 1 to 9) clientArray(i).start

}

val SyncServer = actor {

var queryCounter = 0

loop {

react {

**case** query =>

queryCounter += 1

println("Received message: " + query)

reply("Current query counter: " + queryCounter)

}

}

}

**class** SyncClient(id: Int, server: Actor) **extends** Actor {

override def act() {

val response = server !? "Here is client of ID: " + id

println("Client of ID: " + id + " received response: " + response)

}

}

val AsyncServer = actor {

var queryCounter = 0

loop {

react {

**case** query =>

queryCounter += 1

println("Received message: " + query)

sender ! "Current query counter is " + queryCounter

}

}

}

**class** AsyncClient(id: Int, server: Actor) **extends** Actor {

override def act {

server ! "Here is client of ID: " + id

println("Client of ID " + id + " received response: " + ?)

}

}

}

2.  
**import** scala.actors.Actor

**import** scala.collection.mutable.MutableList

**import** scala.util.Random

object Play {

val players = **new** MutableList[Player]

def main(args: Array[String]) {

val p1 = **new** Player(1)

val p2 = **new** Player(2)

val p3 = **new** Player(3)

players += p1 += p2 += p3

p1.start

p2.start

p3.start

p1 ! Ball(0)

Thread.sleep(5000)

val p4 = **new** Player(4)

p4.start

players += p4

}

**case** **class** Ball(count: Int)

**class** Player(id: Int) **extends** Actor {

val random = **new** Random

def act {

loop {

Thread.sleep(1000)

receive {

**case** Ball(x) =>

println("Throws count " + x + ". Ball received and thrown by user of ID " + id)

players(random.nextInt(players.size)) ! Ball(x + 1)

}

}

}

}

}