1.public class Chopstick {

2.private final int id;

3.public Chopstick(int id) {

4.this.id = id;

5.}

6.

7.// equals, hashcode, and toString() omitted

8.}

1.public interface ChopstickOrder {

2.Chopstick[] getOrder(Chopstick left, Chopstick right);

3.}

01.class Philosopher implements Runnable {

02.public final int id;

03.private final Chopstick[] chopsticks;

04.protected final ChopstickOrder order;

05.

06.public Philosopher(int id, Chopstick[] chopsticks, ChopstickOrder order) {

07.this.id = id;

08.this.chopsticks = chopsticks;

09.this.order = order;

10.}

11.

12.public void run() {

13.while(true) {

14.eat();

15.}

16.}

17.

18.protected void eat() {

19.Chopstick[] chopstickOrder = order.getOrder(getLeft(), getRight());

20.synchronized(chopstickOrder[0]) {

21.synchronized(chopstickOrder[1]) {

22.Util.sleep(1000);

23.}

24.}

25.}

26.

27.Chopstick getLeft() {  return chopsticks[id]; }

28.Chopstick getRight() { return chopsticks[(id+1) % chopsticks.length]; }

29.}

30.}

01.public class GraciousPhilosopher extends Philosopher {

02.

03.private static Map chopstickLocks = new ConcurrentHashMap();

04.

05.public GraciousPhilosopher(int id, Chopstick[] chopsticks, ChopstickOrder order) {

06.super(id, chopsticks, order);

07.

08.// Every philosopher creates a lock for their left chopstick

09.chopstickLocks.put(getLeft(), new ReentrantLock());

10.}

11.

12.protected void eat() {

13.Chopstick[] chopstickOrder = order.getOrder(getLeft(), getRight());

14.Lock firstLock = chopstickLocks.get(chopstickOrder[0]);

15.

16.Lock secondLock = chopstickLocks.get(chopstickOrder[1]);

17.firstLock.lock();

18.try {

19.secondLock.lock();

20.try {

21.Util.sleep(1000);

22.} finally {

23.secondLock.unlock();

24.}

25.} finally {

26.firstLock.unlock();

27.}

28.}

29.}

import java.util.Random;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

/\*\*

\* This is the classic dining philosophers problem. See the textbook for a description of the problem.

\* This version will deadlock.

\*

\* @author Barbara Lerner

\* @version Oct 5, 2010

\*

\*/

public class DiningPhilosophers {

// The number of philosophers

private static final int NUM\_PHILOSOPHERS = 5;

/\*\*

\* Test the dining philosophers solution

\* @param args Not used

\*/

public static void main (String[] args) {

// Model each chopstick with a lock

Lock[] chopsticks = new ReentrantLock[NUM\_PHILOSOPHERS];

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

chopsticks[i] = new ReentrantLock();

}

// Create the philosophers and start each running in its own thread.

Philosopher[] philosophers = new Philosopher[NUM\_PHILOSOPHERS];

for (int i = 0; i < NUM\_PHILOSOPHERS; i++) {

philosophers[i] = new Philosopher(i, chopsticks[i], chopsticks[(i+1)%NUM\_PHILOSOPHERS]);

new Thread(philosophers[i]).start();

}

}

}

/\*\*

\* A philosopher alternates between thinking and eating. To eat, the philosopher needs to pick

\* up the left chopstick and then the right chopstick sequentially. The phillosopher shares

\* chopsticks with its neighbors, so it cannot eat at the same time as either neighbor.

\*

\* @author Barbara Lerner

\* @version Oct 5, 2010

\*

\*/

class Philosopher implements Runnable {

// Used to vary how long a philosopher thinks before eating and how long the

// philosopher eats

private Random numGenerator = new Random();

// The philosopher's unique id

private int id;

// The chopsticks this philosopher may use

private Lock leftChopstick;

private Lock rightChopstick;

/\*\*

\* Constructs a new philosopher

\* @param id the unique id

\* @param leftChopstick chopstick to the left

\* @param rightChopstick chopstick to the right

\*/

public Philosopher (int id, Lock leftChopstick, Lock rightChopstick) {

this.id = id;

this.leftChopstick = leftChopstick;

this.rightChopstick = rightChopstick;

}

/\*\*

\* Repeatedly think, pick up chopsticks, eat and put down chopsticks

\*/

public void run() {

try {

while (true) {

think();

pickUpLeftChopstick();

pickUpRightChopstick();

eat();

putDownChopsticks();

}

} catch (InterruptedException e) {

System.out.println("Philosopher " + id + " was interrupted.\n");

}

}

/\*\*

\* Lets a random amount of time pass to model thinking.

\* @throws InterruptedException

\*/

private void think() throws InterruptedException {

System.out.println("Philosopher " + id + " is thinking.\n");

System.out.flush();

Thread.sleep (numGenerator.nextInt(10));

}

/\*\*

\* Locks the left chopstick to signify that this philosopher is holding it

\*/

private void pickUpLeftChopstick() {

leftChopstick.lock();

System.out.println("Philosopher " + id + " is holding 1 chopstick.\n");

System.out.flush();

}

/\*\*

\* Locks the right chopstick to signify that this philosopher is holding it

\*/

private void pickUpRightChopstick() {

rightChopstick.lock();

}

/\*\*

\* Lets a random amount of time pass to model eating.

\* @throws InterruptedException

\*/

private void eat() throws InterruptedException {

System.out.println("Philosopher " + id + " is eating.\n");

System.out.flush();

Thread.sleep (numGenerator.nextInt(10));

}

/\*\*

\* Releases the locks on both chopsticks to model putting them down so the

\* other philosophers can use them.

\*/

private void putDownChopsticks() {

leftChopstick.unlock();

rightChopstick.unlock();

}

}