Distributed Systems **Spring-Summer 2015**

Assignment 1 **Simon Razniewski**

Threads in Java

Instructions: You are allowed to work alone or in teams of two students.
Submit a single file DiningPhilosophers.java which contains (i) an explanation of your solution to the problem as comment, (ii) the Java code.

Problem Statement: This programming assignment gives you the opportunity to do a small concurrent object-oriented programming using threads. The assignment is to be done in Java.
Dining Philosophers A common problem in concurrent programming is coordinating access to a shared resource by enforcing a synchronization condition between multiple concurrent processes. A classic example is the Dining Philosophers problem, which consists of 5 philosophers seated at a round table thinking and eating concurrently. For this version of the Dining Philosophers problem, they are eating using a pair of chopsticks. Each philosopher has a left chopstick and a right chopstick, so there are 5 chopsticks in total. In order to eat, a philosopher must obtain both left and right chopsticks exclusively. The philosopher must pick up one, then the other chopstick. If a chopstick is not available a philosopher must wait for it to be released by his/her neighbor before being allowed to eat. A philosopher is allowed to eat for no more than 1 second before relinquishing both chopsticks to think again, so as not to starve the other philosophers. After a random number of milliseconds of thinking, a philosopher may try to eat again but has to re-acquire both chopsticks. Below is a fragment of a Philosopher class that inherits the java.lang.Thread class and shows the order in which to think, acquire chopsticks, eat, and release chopsticks:

```
public class Philosopher extends Thread{
  private String name;
  private Chopstick left, right;

  // each Philosopher is assigned an integer id and two chopsticks
  public Philosopher (int id, Chopstick c1, Chopstick c2) {
    name = "Philosopher-" + id;
    left = c1; right = c2;
  }

  // the run method is invoked by calling the start()method on
  // an instance of the Philosopher class, e.g., phil.start()
  public void run() {
    while (true) { // loop forever
      System.out.println(name + " thinking");
```

```
      // TODO: think for random number of milliseconds <=1 sec
      // TODO: acquire both chopsticks
      System.out.println (name + "dining");
      // TODO: eat for random number of milliseconds <= 1sec
      // TODO: release both chopsticks
    }
  }
}
```

In addition to the Philosopher class, you will need a Chopstick class for enforcing synchronized access to each chopstick. A philosopher acquires a chopstick when it is available, or waits. At no time may two philosophers hold the same chopstick. When finished eating, a philosopher releases both chopsticks.
Your main program must create 5 Chopstick objects and 5 Philosopher objects and assign to each Philosopher the correct pair of Chopsticks (modulo 5), one shared with the Philosopher on the left and one with the Philosopher on the right. Create each Philosopher thread and start it running in your main procedure.
Run the program for at least 1 minute verifying that each Philosopher gets to eat several times. On termination, the program must report statistics on a) how many times each philosopher got to eat and b) how long (in milliseconds) each philosopher spent eating and thinking. Use these statistics to verify that neither starvation nor deadlock occurs for any Philosopher.

Update: Having starvation or deadlock occur is unlikely in a run of one minute. So please make sure that your implementation is deadlock and starvation free by design, not only by experience. Explain in your comment which strategy you use to avoid deadlocks/livelocks

Implementation Notes: 1. In Java, a thread sleeps by calling the Thread.sleep() method passing an integer value representing the number of milliseconds to sleep. The easiest thing to do is write a randomSleep() function as follows, which also handles an exception that could be thrown by the Thread.sleep() function. Math.random() returns a double value between 0.0 and 1.0, so it must be multiplied by 1000 and converted to an integer value.

```
void randomSleep () {
  int ms = (int)(1000 * Math.random());
  try { Thread.sleep(ms); }
  catch (InterruptedException e) { /* ignore it */}
}
```

Submission: Friday, 6th of March 2015, 10:00 am via Moodle.