Programming assignment #3 – The Dining Philosophers Problem

The Dining Philosophers Problem – For this lab, we will write a multi-threaded program, in Java, that implements the well-known *Dining Philosophers* problem. This particular problem is often used to demonstrate the use of mutual exclusion algorithms and deadlock prevention in multi-threaded applications. The scenario is that there are some number *N* of oriental philosophers sitting around a round table (for our implementation we will use *N*=5). Each philosopher has one chopstick at his right hand and one chopstick at his left hand. However, the right hand chopstick for philosopher *n* is the left hand chopstick for philosopher *(n+1) mod N*. In other words, there are *N* chopsticks for *N* philosophers (one between each pair of philosophers).

As the philosophers are sitting around the table, each enters a state of deep thinking for a random amount of time. Eventually, each philosopher becomes hungry and picks up the chopstick at his right hand. If the right hand chopstick is not available, he simply waits until it is available. He then picks up the left hand chopstick, again waiting if it is not available. Once he has two chopsticks, he begins eating for a random amount of time. When finished eating, he puts down the chopsticks one at the time, freeing them for use by a neighboring philosopher (ignoring for new any possible hygiene problems with this approach). If you think through this scenario, you should be able to see the potential for a *deadlock*, where each philosopher has a single chopstick and is waiting infinitely for the second one.

Java Implementation. For the Java implementation, you are given a file `Diningjava.zip` posted on the eclass webpage contains the java skeleton `dining.java` and `miscsubs.java`. You must compile the class files individually with Eclipse environment. After the un-zip process, the directory DiningJava will contain the skeleton classes. “dining” is the class that has the main method that is currently empty. You need to fill the main method by editing. `miscsubs.java` contains a set of methods provided by the instructor. Use *Java* threads for each philosopher. Use *Java* `synchronize` to lock the chopsticks. Use the *Java* `wait` and `notify` methods to implement the condition variables.

If your implementation is successful, the output will be:

```
Starting the Dining Philosophers Simulation
EatCount 0 - 100
EatCount 1 - 100
EatCount 2 - 100
EatCount 3 - 100
EatCount 4 – 100
```

Implementation Details. Your implementation should use one *thread* to model the behavior of each philosopher. Each thread should be assigned a unique integer *philosopher identifier*, in the range 0 . . . (N −1). The main program should create N threads, and should insure that each thread has begun processing before creating the next thread (use a *Condition Variable* for this). After all threads have been created, your main method **must not** just burn CPU time waiting for the philosophers (threads) to exit. Instead, you must use a condition variable which the philosophers use to inform the main program they are exiting. You should also ”desk check” your code carefully to be sure that you don’t have a possible deadlock situation, since just running the program for a while with no problems does not mean there are not potential problems. Each philosopher thread should loop in a *thinking*, *hungry*, *eating* cycle until a fixed number of eating states have been processed (we will use 500). There are four methods in `miscsubs.java` that you should use to instrument your program. `RandomDelay` is called to have the philosopher do nothing for a random amount of time (both in thinking state and eating state). **When a philosopher starts eating, the `StartEating` subroutine should be called. When a philosopher stops eating, the `DoneEating` subroutine should be called. Both `StartEating` and `DoneEating` have a single integer parameter, indicating which philosopher the action is for in the range 0 . . . (N − 1)**. The `StartEating` subroutine maintains a global variable, `TotalEats`, that each thread can use to determine when to exit. Also defined is a global constant `MAX_EATS`, which is set to 500. When all threads have exited, the `LogResults` subroutine should be called.

Expected Results. Obviously, the program should not deadlock and should run to completion. Further, there should be some degree of *fairness* among the threads. In other words, each philosopher should eat approximately the same number of times. For our simulation, we have 5 philosophers and MAX EATS of 500, so each philosopher should be able to eat about 100 times. If the number of eating is more than 500, the program is exiting forcibly. If there is a philosopher who cannot eat more at least 20 rounds, the simulation also forcibly exits.

Due to a number of thread scheduling issues, this could vary somewhat, but in no case should one or more philosophers starve.

Grading Criteria:
Proper Creation of Philosopher threads – 15 points
Proper Management of Chopsticks – 40points.
Thread Fairness – 35points.
Code neatness and commenting – 10point.