Operating Systems Project – The Dining Philosophers Monitor

Goal: Implement a Monitor to solve the Dining Philosophers problem.

Description: Section 5.8.2 of the text describes a solution to the dining philosophers problem using monitors. Implement this solution using Java's condition variables.

Input: n/a

Output: Print output to System.out to demonstrate your functioning solution.

Deliverables: Java program in an executable jar file names [YOUR_NAME]DiningPhilos.jar, all sources (please include your entire src folder), a readme.txt, and a post-mortem in a zip file named [YOUR_NAME]DiningPhilos.zip. All source code belongs in a package named edu.frostburg.cosc460.

[YOUR_NAME] = LastNameFirstName

The program will be tested in Ubuntu 16.

Details:

Part 1: Reread section 5.8 – 5.8.2 on Monitors and help you design the basics.

Begin by setting up your project. You should have 3 components: The server, the philosophers, and the driver. Begin by creating these classes. The server should implement the following interface:

```java
public interface DiningServer {

    /* called by philosopher that wishes to eat */
    public void takeChopsticks(int philNumber);

    /* called by philosopher when it has finished eating */
    public void returnChopsticks(int philNumber);
}
```

The driver runs your program, and the Philosophers simply alternate between thinking and eating. You should also include an enumerated type for Philosopher states (HUNGRY, EATING, THINKING).

Part 2: Implement the Philosophers

The philosophers don't do much of anything. They represent units of work in parallel processing, but don't actually do anything useful. A philosopher only needs two pieces of data, its ID (0 – 4) and a reference to the server from where it can collect chopsticks.

Each philosopher runs independently (i.e. are Runnable) and infinitely alternates between thinking and eating. To keep things running at a manageable pace, they should wait with Thread.sleep() for random amounts of time between each step. In other words, a philosopher should wait (think), then takeChopsticks(myID) (hungry), then wait (eat), and then return the chopsticks. Forever. Such is life for the philosopher.

Part 3: Implement the Server using the pseudocode in 5.8.2.

The pseudocode get you far in your solution, but it misses a key component, how to manage the Condition variables. Java attaches condition variables to the ReentrantLock class. (API) You can get a Condition from your lock.

```
Lock lock = new ReentrantLock();
// get condition with lock.newCondition().  You need 5
```

Use Java's condition variables to synchronize the activity of the philosophers and prevent deadlock.

After you have completed the DiningPhilosophersServer, you can create your driver class. Your driver will set up and start the Philosopher threads, then simply loop, printing out the state of the server every now and then.

Part 4: Testing and Wrap-Up

Once you have completed the above, it can be difficult to tell what you have done. Make good use of the debugger and system output to make sure that your philosophers each get a chance to eat. (Debugging threads in Netbeans).

After you are satisfied that your program runs correctly, make sure to complete your documentation and submit your project to Blackboard by the due date.

Checklist

	Item	Value
	Part 1	10%
	Part 2	30%
	Part 3	30%
	Part 4	10%
	Documentation	10%
	Deliverables	5%
	Post-mortem	5%

Avoid

	Item	Value
	Bugs	-x
	Runtime errors	-y
	Unsatisfactory quality	-z
	Compilation errors	- 50+%