Dining Philosophers Assignment

On the next page is an outline of a solution to the dining-philosophers problem using monitors. This assignment requires implementing this solution using Java's condition variables.

Begin by creating five philosophers, each identified by a number 0...4. Each philosopher runs as a separate thread. Philosophers alternate between thinking and eating. When a philosopher wishes to eat, it invokes the method takeChopsticks(philNumber),where philNumber identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, it invokes returnChopsticks(philNumber).

Your solution will implement the following interface:

```
public interface DiningServer
{
  /* Called by a philosopher when it wishes to eat */
  public void takeChopsticks(int philNumber);

  /* Called by a philosopher when it is finished eating */
  public void returnChopsticks(int philNumber);
}
```

The implementation of the interface follows the outline of the solution provided in Figure 1 on page 3. Use Java's condition variables to synchronize the activity of the philosophers and prevent deadlock.

Dining-Philosophers Solution Using Monitors

This is a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structures:

```
enum State { THINKING, HUNGRY, EATING };
State states = new State[5];
```

Philosopher *i* can set the variable `states[i] = State.EATING` only if her two neighbors are not eating. That is, the conditions `(states[(i + 4) % 5] != State.EATING)` and `(states[(i + 1) % 5] != State.EATING)` hold.

We also need to declare:

```
Condition[] self = new Condition[5];
```

where philosopher *i* can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Now the description of the solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor `dp`, which is an instance of the monitor type `DiningPhilosophers`. Figure 1 (on the next page) shows the definition of this monitor type using a Java-like pseudocode. Each philosopher, before starting to eat, must invoke the operation `takeChopsticks()`. This act may result in the suspension of the philosopher thread. After the successful completion of the operation, the philosopher may eat. After she eats, the philosopher invokes the `returnChopsticks()` operation and starts to think. Thus, philosopher *i* must invoke the operations `takeChopsticks()` and `returnChopsticks()` in the following sequence:

```
dp.takeChopsticks(i);
eat();
dp.returnChopsticks(i);
```

It is easy to show that this solution ensures that no two neighboring philosophers are eating simultaneously and that no deadlocks will occur. However, it is possible for a philosopher to starve to death. This assignment does **NOT** require a solution to that problem.

```
monitor DiningPhilosophers
{
  enum State { THINKING, HUNGRY, EATING };
  State[] states = new State[5];
  Condition[] self = new Condition[5];

  public DiningPhilosophers
  {
    for (int i = 0; i < 5; i++) states[i] = State.THINKING;
  }

  public void takeChopsticks(int i)
  {
    states[i] = State.HUNGRY;
    test(i);
    if (states[i] != State.EATING) self[i].wait;
  }

  public void returnChopsticks(int i)
  {
    states[i] = State.THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
  }

  private void test(int i)
  {
    if ((states[(i + 4) % 5] != State.EATING) &&
        (states[i] == State.HUNGRY) &&
        (states[(i + 1) % 5] != State.EATING))
    {
      states[i] = State.EATING;
      self[i].signal;
    }
  }
}
```

Figure 1

Additional Design Requirements

1. The driver class is the class **Assignment2** and it should only contain only one method, main. The main method creates the dining philosopher object and an instance of each philosopher followed by starting the execution of each philosopher.

2. When a philosopher is "thinking" have the philosopher print: "Philosopher ***i*** is thinking." with i being the number of the philosopher who is thinking. After the philosopher prints that it is thinking have the philosopher sleep for a random amount of time. The SleepUtilities class from the Factory.java program provides the ability for a thread to sleep.

3. When a philosopher is "eating" have the philosopher print: "Philosopher ***i*** is eating." with i being the number of the philosopher who is eating. After the philosopher prints that it is eating have the philosopher sleep for a random amount of time. The SleepUtilities class from the Factory.java program provides the ability for a thread to sleep.

4. When a philosopher acquires both of its chopsticks have the DiningPhilosopher object print: "Philosopher ***i*** acquired its left and right chopsticks." with i being the number of the philosopher who acquired its chopsticks. When a philosopher releases both of its chopsticks have the DiningPhilosopher object print: "Philosopher ***i*** released its left and right chopsticks." with i being the number of the philosopher who released its chopsticks.

5. **Tip:** You must make your program as modular as possible, not placing all your code in one .java file. You must create multiple classes as you need. Methods should be reasonably small following the guidance that "A function should do one thing, and do it well." You will lose a lot of points for code readability if you don't make your program as modular as possible. But, do not go overboard on creating classes and methods. Your common sense should guide your creation of classes and methods.

6. Do **NOT** use your own packages in your program. If you see the keyword **package** on the top line of any of your .java files then you created a package. Create every .java file in the **src** folder of your Eclipse project

7. Do **NOT** use any graphical user interface code in your program!

8. Do **NOT** write any comments for your code. If you make your code as modular as possible then anyone reading your code will easily be able to determine the code's task.

Grading Criteria

The total assignment is worth 20 points, broken down as follows:

1. If your code does not implement the task described in this assignment then the grade for the assignment is zero.
2. If your program does not compile successfully then the grade for the assignment is zero.
3. If your program produces runtime errors which prevents the grader from determining if your code works properly then the grade for the assignment is zero.

If the program compiles successfully and executes without significant runtime errors then the grade computes as follows:
Followed proper submission instructions, 4 points:

1. Was the file submitted a zip file.
2. The zip file has the correct filename.
3. The contents of the zip file are in the correct format.
4. The keyword **package** should not appear at the top of any of the .java files.

Program execution, 4 points:

- Program output, the program produces the correct results for the input.

Code implementation, 8 points:

- The driver file has the correct filename, **Assignment2.java** and contains only the method **main** performing the exact tasks as described in the assignment description.
- The code performs all the tasks as described in the assignment description.
- The code is free from logical errors.

Code readability, 4 points:

- Good variable, method, and class names.
- Variables, classes, and methods that have a single small purpose.
- Consistent indentation and formatting style.
- Reduction of the nesting level in code.

Late submission penalty: assignments submitted after the due date are subjected to a 2 point deduction for each day late.

Submission Instructions

Go to the folder containing the .java files of your assignment and select all (and **ONLY**) the .java files which you created for the assignment in order to place them in a Zip file. The file should **NOT** be a **7z** or **rar** file! Then, follow the directions below for creating a zip file depending on the operating system running on the computer containing your assignment's .java files.

Creating a Zip file in Microsoft Windows (any version):

1. Right-click any of the selected .java files to display a pop-up menu.
2. Click on **Send to**.
3. Click on **Compressed (zipped) Folder**.
4. Rename your Zip file as described below.
5. Follow the directions below to submit your assignment.

Creating a Zip file in Mac OS X:

1. Click **File** on the menu bar.
2. Click on **Compress ? Items** where ? is the number of .java files you selected.
3. Mac OS X creates the file **Archive.zip**.
4. Rename **Archive** as described below.
5. Follow the directions below to submit your assignment.

Save the Zip file with the filename having the following format:

your last name,
followed by an underscore _,
followed by your first name,
followed by an underscore _,
followed by the word **Assignment2**.

For example, if your name is John Doe then the filename would be: **Doe_John_Assignment2**

Once you submit your assignment you will not be able to resubmit it!
Make absolutely sure the assignment you want to submit is the assignment you want graded.
There will be **NO** exceptions to this rule!

You will submit your Zip file via your CUNY Blackboard account.
Follow these instructions:

Log onto your CUNY BlackBoard account.
Click on the CSCI 340 course link in the list of courses you're taking this semester.
Click on **Content** in the green area on the left side of the webpage.
You will see the **Assignment 2 – Dining Philosophers Assignment**.
Click on the assignment.
Upload your Zip file and then click the submit button to submit your assignment.

Due Date: Submit this assignment by Thursday, December 13, 2018.