Nimesh Silva

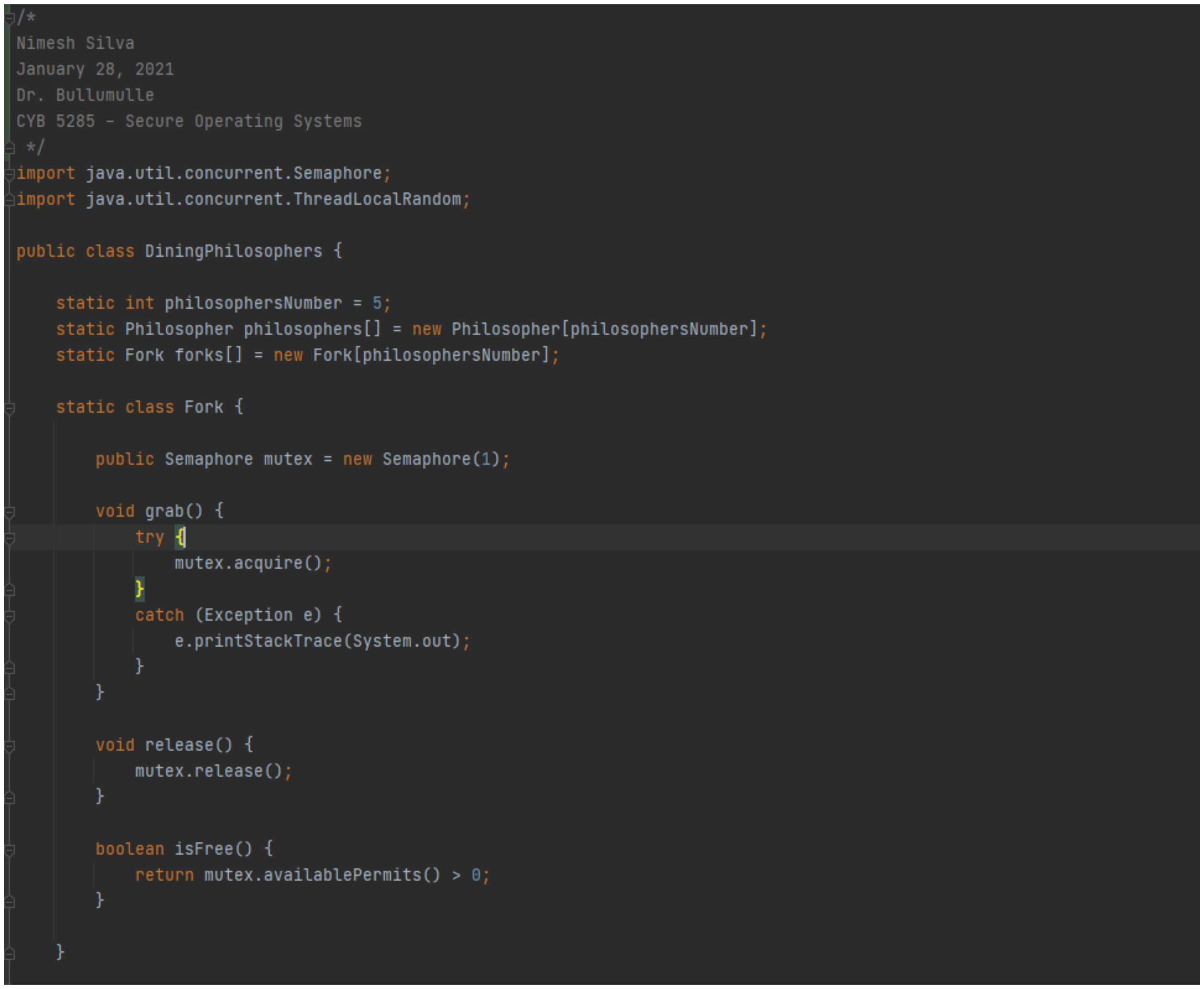
January 28, 2021

Dr. Bullumulle

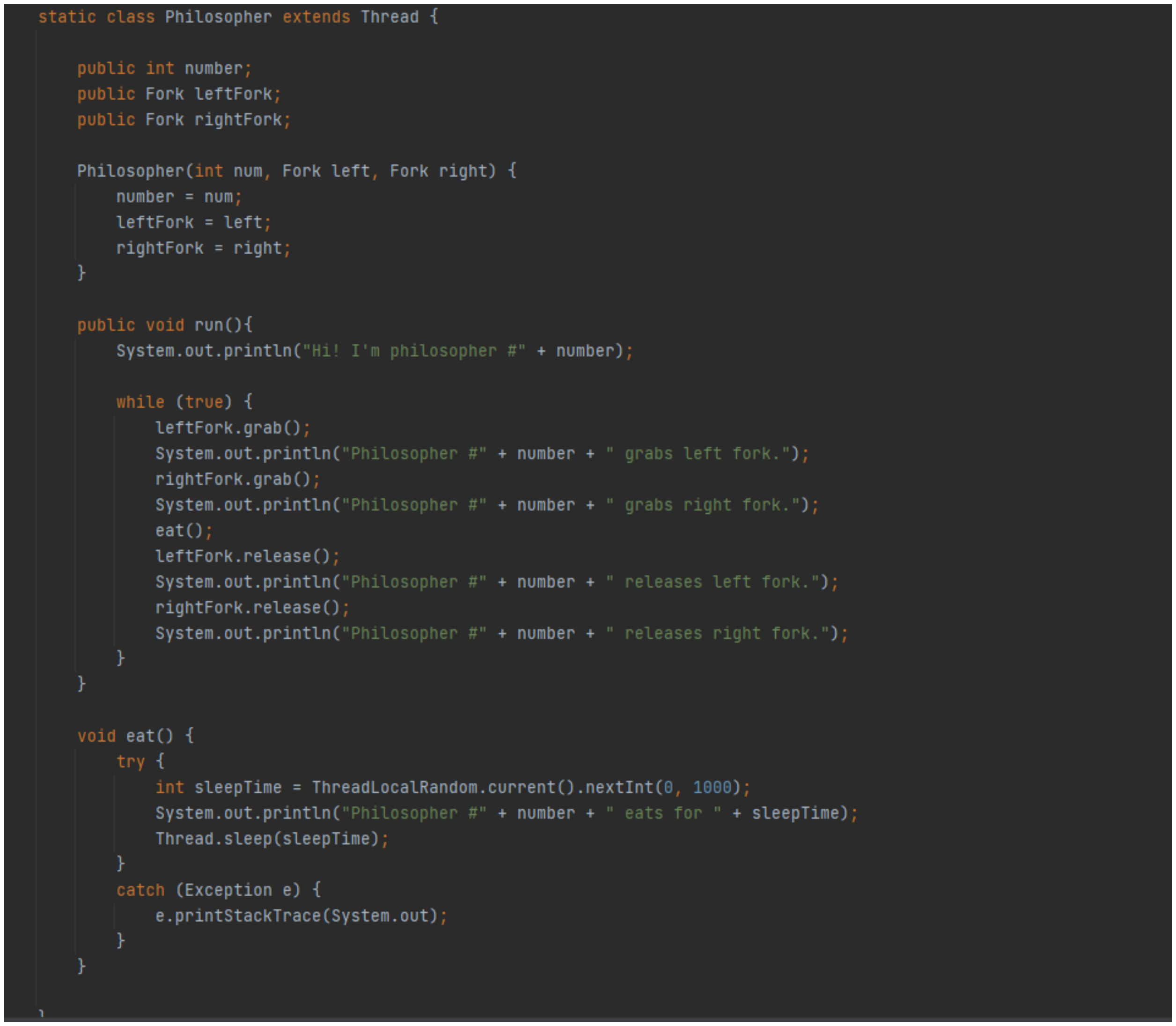
CYB 5285

3.2 Assignment: Dining Philosophers

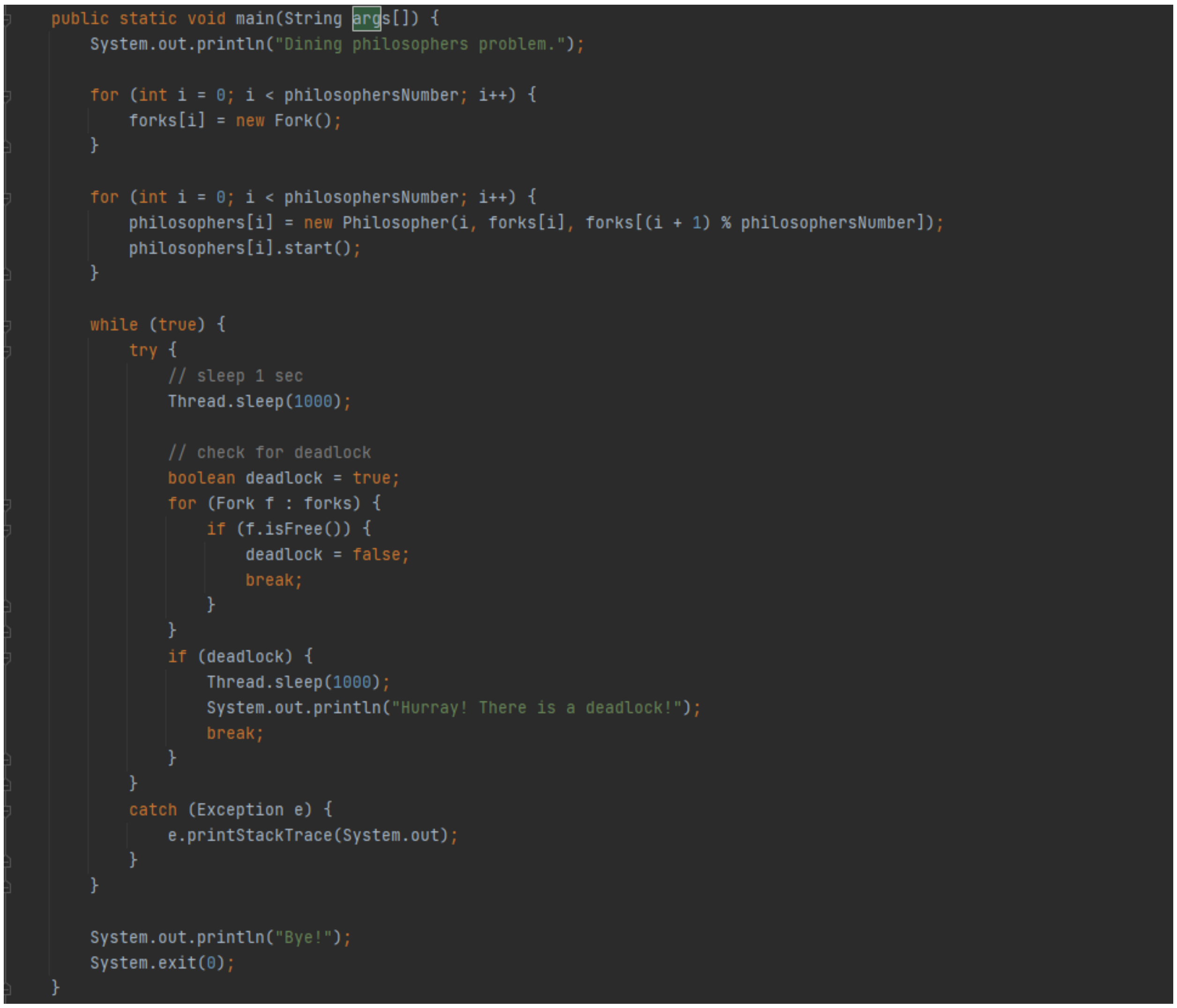
This program was written in Java. The aim is to create a procedure for philosophers to accomplish their goal of eating and dreaming without starving to death. The philosopher gets hungry after some time and wants to feed. The philosopher reaches for the forks on either side of him to do this. Once the philosopher succeeds in getting both people, he starts to feed. If he puts both forks down, his neighbor's philosopher has access to those forks. We need to lock it to simulate acquiring a fork so that no two philosophical threads obtain it at the same time. We use the synchronized keyword to obtain the fork object's internal monitor to accomplish this and prevent other threads from doing the same. This is performed in the process run(). For a moment, a philosopher thinks and then decides to eat. To explain the order in which actions occur, timestamps have also been applied to each operation. As generic Java objects, we model each of the forks and render as many of them as philosophers exist. Using the synchronized keyword, we pass each Philosopher his left and right forks that he tries to lock. Both philosophers, without triggering a deadlock, get their chance to think and eat. We analyzed the popular Dining Philosophers problem using threads in this paper. My code can be found here https://github.com/nimeshsilva1997/JVM-Projects/tree/master/Java/Dining.



```java
/*
Nimesh Silva
January 28, 2021
Dr. Bullumulle
CYB 5285 - Secure Operating Systems
 */
import java.util.concurrent.Semaphore;
import java.util.concurrent.ThreadLocalRandom;

public class DiningPhilosophers {

    static int philosophersNumber = 5;
    static Philosopher philosophers[] = new Philosopher[philosophersNumber];
    static Fork forks[] = new Fork[philosophersNumber];

    static class Fork {

        public Semaphore mutex = new Semaphore(1);

        void grab() {
            try {
                mutex.acquire();
            }
            catch (Exception e) {
                e.printStackTrace(System.out);
            }
        }

        void release() {
            mutex.release();
        }

        boolean isFree() {
            return mutex.availablePermits() > 0;
        }

    }
```



```java
static class Philosopher extends Thread {

    public int number;
    public Fork leftFork;
    public Fork rightFork;

    Philosopher(int num, Fork left, Fork right) {
        number = num;
        leftFork = left;
        rightFork = right;
    }

    public void run(){
        System.out.println("Hi! I'm philosopher #" + number);

        while (true) {
            leftFork.grab();
            System.out.println("Philosopher #" + number + " grabs left fork.");
            rightFork.grab();
            System.out.println("Philosopher #" + number + " grabs right fork.");
            eat();
            leftFork.release();
            System.out.println("Philosopher #" + number + " releases left fork.");
            rightFork.release();
            System.out.println("Philosopher #" + number + " releases right fork.");
        }
    }

    void eat() {
        try {
            int sleepTime = ThreadLocalRandom.current().nextInt(0, 1000);
            System.out.println("Philosopher #" + number + " eats for " + sleepTime);
            Thread.sleep(sleepTime);
        }
        catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }

}
```



```java
public static void main(String args[]) {
    System.out.println("Dining philosophers problem.");

    for (int i = 0; i < philosophersNumber; i++) {
        forks[i] = new Fork();
    }

    for (int i = 0; i < philosophersNumber; i++) {
        philosophers[i] = new Philosopher(i, forks[i], forks[(i + 1) % philosophersNumber]);
        philosophers[i].start();
    }

    while (true) {
        try {
            // sleep 1 sec
            Thread.sleep(1000);

            // check for deadlock
            boolean deadlock = true;
            for (Fork f : forks) {
                if (f.isFree()) {
                    deadlock = false;
                    break;
                }
            }
            if (deadlock) {
                Thread.sleep(1000);
                System.out.println("Hurray! There is a deadlock!");
                break;
            }
        }
        catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }

    System.out.println("Bye!");
    System.exit(0);
}
```