Assignment 1

February 11th, 2021

ECSE 420
Parallel Computing
Winter 2021
Group 26

Rintaro Nomura - 260781007
Edward Latulipe-Kang - 260746475

<u>Question 1</u>

1.1)
See figure 7 in Appendix A for the implementation for sequential matrix multiplication. Please see source code MatrixMultiplication.java for additional information.

The implementation works by storing all elements in a given row in a separate array. This is also true for all elements in a given column. Once this is done, we can take both arrays, and sum the products of the elements at some position along both intermediate arrays. This intermediate step of creating and storing in separate arrays is not entirely necessary, but was the initial thought process. A more efficient algorithm was implemented for the parallel implementation, but was not used for the sequential component. In any case, both implementations operate at O(n^2) due to the nested for-loops. They are roughly equivalent in performance.

1.2)
See figure 8 in Appendix A for the implementation for parallel matrix multiplication. Please see source code MatrixMultiplication.java for additional information.

The implementation works by giving each thread a set of rows with which it will perform the matrix multiplication. This allows for the matrix to be neatly divided between threads without any overlap. For cases where the matrix cannot be neatly divided given the number of threads, we offload the remaining portions of the matrix to recently “finished” threads. Note that a finished thread hasn’t necessarily ended yet. Instead, it has only completed computation on an assigned chunk of the matrix. Each thread will effectively deal with chunks of the matrix that correspond to the appropriate remainder as well. Simply put, this is a modulo operation. In the event that there are too many threads, we simply delegate one row to one thread. Dividing rows and columns beyond a 1-D vector would create much more overhead than is needed and would be computationally tedious.

1.3)
See figure 9 in Appendix A for the code that allows us to measure execution time. Please see source code MatrixMultiplication.java for additional information.

We simply use the built in system timing function to determine the start and end time of the matrix multiplication. The implementation is fairly straightforward, as it only requires that the matrix multiplication be nested between those checkmarks.

1.4)
Please refer to the graph shown in Figure 1 below. These tests were run using a 6-core intel CPU. Each point in the graph was extracted from measurements seen in Table 1 of Appendix A.

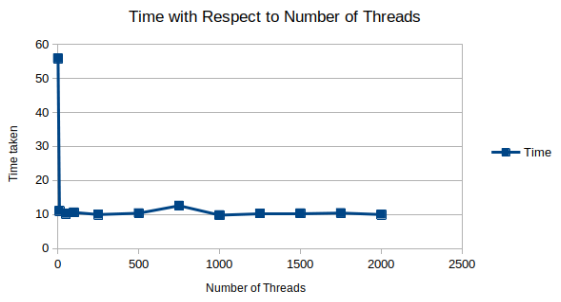


Figure 1. Execution Time vs Number of Threads

1.5)
Please refer to the graph shown in Figure 2 below. Each point in the graph was extracted from measurements seen in Table 2 of Appendix

A. We have used 1000 threads for this part. This will be elaborated upon shortly.

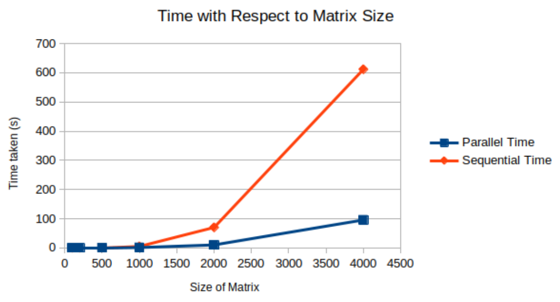


Figure 2. Execution Time vs Size of Matrices

1.6)
From the graph in Figure 1, we can see that it takes the program roughly 1 minute to execute completely. Every additional thread beyond that hovers around the 10s mark. This may be because of the additional overhead that is created for each new thread to run. That is to say, the amount of overhead that is involved is proportional to the number of threads created. In essence, creating additional threads to decrease time is inefficient here as it will also lead to greater overhead. Given that the execution time remains constant, we used 1000 threads for computing the execution time as a function of matrix size.

The graph in Figure 2 illustrates the time required to run our program in parallel versus the time required for a sequential run. We note that there is greater exponential growth for the sequential computation. This makes sense, given that each row and column must be handled sequentially and incrementally. The threaded version allows for parallel execution of many rows and columns. Furthermore, for matrices of size 100 and 200 only , the sequential implementation performed better than the parallel counterpart. Beyond those sizes, parallel execution performed far better, as mentioned earlier.

Question 2

2.1)
Deadlock occurs when multiple threads are competing for resources and are stuck waiting on each other. In the case of our implementation, which can be seen in Figure 11 in the appendix B, 2 threads share 2 locks. More specifically, one thread, while having acquired one lock, will attempt to acquire the other. This is problematic, because that other lock was already acquired by a different thread which is also doing the same thing. Because both threads are restricting resources from each other, neither can proceed. Figure 3 shown below demonstrates the deadlock state as nothing more can occur.

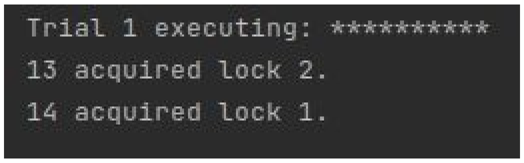
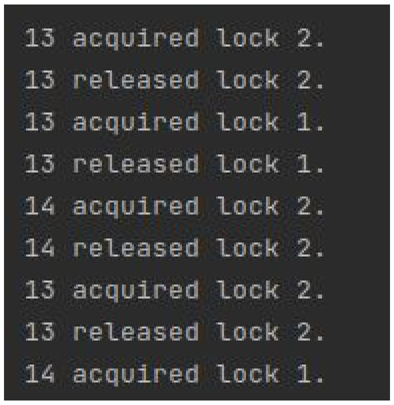

```
Trial 1 executing: **********
13 acquired lock 2.
14 acquired lock 1.
```

Figure 3. Deadlock resulting from execution

2.2)
A possible design solution in our implementation is to simply avoid using "nested" locks. That is to say, a thread should release a lock before proceeding to obtain another one. Essentially, we would like for the threads to yield to each other. We can further elaborate on this by having threads declare themselves as victims to yield. Additional design solutions include a flagging system, such that different threads are aware of who wants access to what. Reordering the locks so that they aren't nested allows for the proper flow of both threads as per Figure 4 shown below.

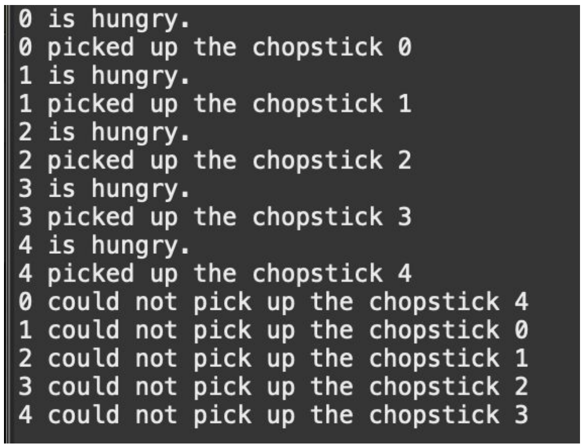


```
13 acquired lock 2.
13 released lock 2.
13 acquired lock 1.
13 released lock 1.
14 acquired lock 2.
14 released lock 2.
13 acquired lock 2.
13 released lock 2.
14 acquired lock 1.
```

Figure 4. Deadlock avoided via reordering

Question 3

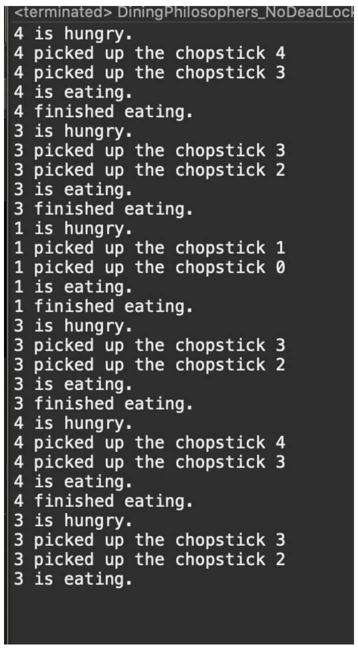
3.1)
The code for thinking philosophers is available as figure 12 in appendix C, as well as the attached diningphilosophers.java. In the code, it demonstrates that if all the philosophers decided to grab a chopstick on their left hand at the same time, they will be stuck waiting at the chopstick to be available for the right hand, such that the deadlock occurs. This is shown in the figure 5, where the system is in deadlock once the 4th philosopher picked up the chopstick on the left hand.



```
0 is hungry.
0 picked up the chopstick 0
1 is hungry.
1 picked up the chopstick 1
2 is hungry.
2 picked up the chopstick 2
3 is hungry.
3 picked up the chopstick 3
4 is hungry.
4 picked up the chopstick 4
0 could not pick up the chopstick 4
1 could not pick up the chopstick 0
2 could not pick up the chopstick 1
3 could not pick up the chopstick 2
4 could not pick up the chopstick 3
```

Figure 5. Deadlock Demo for Question 3

3.2)
The program has been amended in diningphilosophers_nodeadlock_nostarvation.java so that it never reaches a state where philosophers are deadlocked. We have added the restriction that only one philosopher can be eating at the same time, therefore if anyone is hungry, this person has to come back another time.



```
<terminated> DiningPhilosophers_NoDeadLock
4 is hungry.
4 picked up the chopstick 4
4 picked up the chopstick 3
4 is eating.
4 finished eating.
3 is hungry.
3 picked up the chopstick 3
3 picked up the chopstick 2
3 is eating.
3 finished eating.
1 is hungry.
1 picked up the chopstick 1
1 picked up the chopstick 0
1 is eating.
1 finished eating.
3 is hungry.
3 picked up the chopstick 3
3 picked up the chopstick 2
3 is eating.
3 finished eating.
4 is hungry.
4 picked up the chopstick 4
4 picked up the chopstick 3
4 is eating.
4 finished eating.
3 is hungry.
3 picked up the chopstick 3
3 picked up the chopstick 2
3 is eating.
```

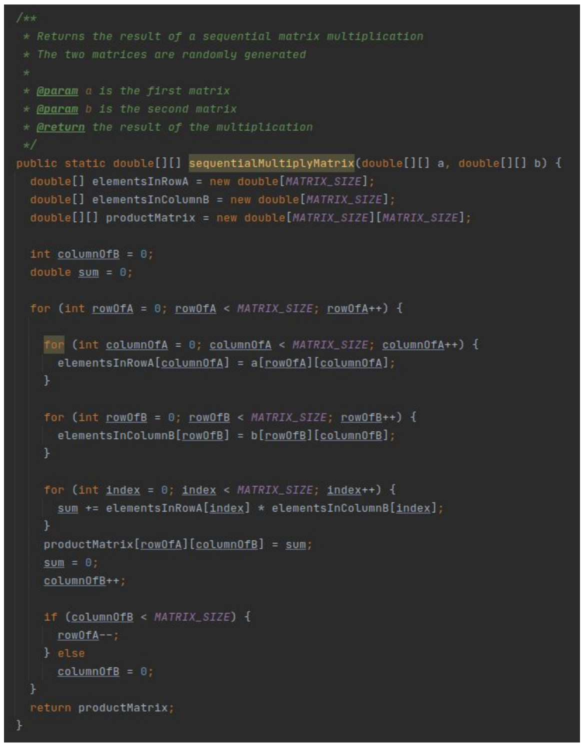
Figure 6. Five Philosophers Problem without Deadlock

3.3)
The program we amended for 3.2 also allows no starvation because it guarantees the eating by first-come first-serve basis, without reservations allowed. Although it does not starve anyone, in order to be fair to the philosophers, a priority queue could be implemented so that if someone failed to eat for a couple consecutive trials then this person could be given a priority to eat when the next slot is available.

Question 4

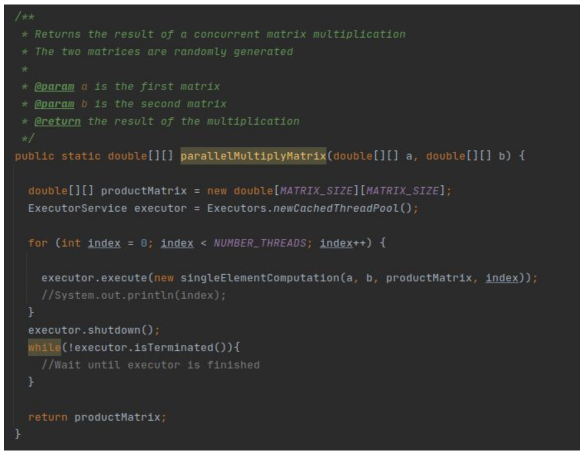
Please see attachment at the very end of this document.

Appendix A:
Question 1 Source Code



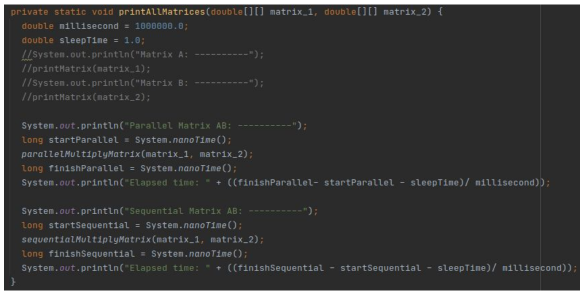
```
/**
 * Returns the result of a sequential matrix multiplication
 * The two matrices are randomly generated
 *
 * @param a is the first matrix
 * @param b is the second matrix
 * @return the result of the multiplication
 */
public static double[][] sequentialMultiplyMatrix(double[][] a, double[][] b) {
  double[] elementsInRowA = new double[MATRIX_SIZE];
  double[] elementsInColumnB = new double[MATRIX_SIZE];
  double[][] productMatrix = new double[MATRIX_SIZE][MATRIX_SIZE];

  int columnOfB = 0;
  double sum = 0;

  for (int rowOfA = 0; rowOfA < MATRIX_SIZE; rowOfA++) {

    for (int columnOfA = 0; columnOfA < MATRIX_SIZE; columnOfA++) {
      elementsInRowA[columnOfA] = a[rowOfA][columnOfA];
    }

    for (int rowOfB = 0; rowOfB < MATRIX_SIZE; rowOfB++) {
      elementsInColumnB[rowOfB] = b[rowOfB][columnOfB];
    }

    for (int index = 0; index < MATRIX_SIZE; index++) {
      sum += elementsInRowA[index] * elementsInColumnB[index];
    }
    productMatrix[rowOfA][columnOfB] = sum;
    sum = 0;
    columnOfB++;

    if (columnOfB < MATRIX_SIZE) {
      rowOfA--;
    } else
      columnOfB = 0;
  }
  return productMatrix;
}
```

Figure 7. Sequential Implementation



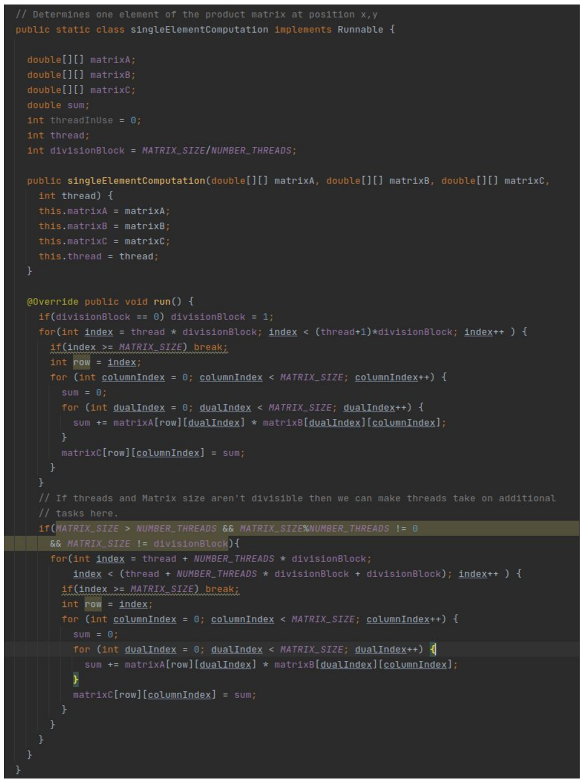
```
/**
 * Returns the result of a concurrent matrix multiplication
 * The two matrices are randomly generated
 *
 * @param a is the first matrix
 * @param b is the second matrix
 * @return the result of the multiplication
 */
public static double[][] parallelMultiplyMatrix(double[][] a, double[][] b) {

  double[][] productMatrix = new double[MATRIX_SIZE][MATRIX_SIZE];
  ExecutorService executor = Executors.newCachedThreadPool();

  for (int index = 0; index < NUMBER_THREADS; index++) {

    executor.execute(new singleElementComputation(a, b, productMatrix, index));
    //System.out.println(index);
  }
  executor.shutdown();
  while(!executor.isTerminated()){
    //Wait until executor is finished
  }

  return productMatrix;
}
```

Figure 8. Parallel Implementation



```
private static void printAllMatrices(double[][] matrix_1, double[][] matrix_2) {
  double millisecond = 1000000.0;
  double sleepTime = 1.0;
  //System.out.println("Matrix A: ----------");
  //printMatrix(matrix_1);
  //System.out.println("Matrix B: ----------");
  //printMatrix(matrix_2);

  System.out.println("Parallel Matrix AB: ----------");
  long startParallel = System.nanoTime();
  parallelMultiplyMatrix(matrix_1, matrix_2);
  long finishParallel = System.nanoTime();
  System.out.println("Elapsed time: " + ((finishParallel- startParallel - sleepTime)/ millisecond));

  System.out.println("Sequential Matrix AB: ----------");
  long startSequential = System.nanoTime();
  sequentialMultiplyMatrix(matrix_1, matrix_2);
  long finishSequential = System.nanoTime();
  System.out.println("Elapsed time: " + ((finishSequential - startSequential - sleepTime)/ millisecond));
}
```

Figure 9. Execution Time Implementation



```
// Determines one element of the product matrix at position x,y
public static class singleElementComputation implements Runnable {

  double[][] matrixA;
  double[][] matrixB;
  double[][] matrixC;
  double sum;
  int threadInUse = 0;
  int thread;
  int divisionBlock = MATRIX_SIZE/NUMBER_THREADS;

  public singleElementComputation(double[][] matrixA, double[][] matrixB, double[][] matrixC,
    int thread) {
    this.matrixA = matrixA;
    this.matrixB = matrixB;
    this.matrixC = matrixC;
    this.thread = thread;
  }

  @Override public void run() {
    if(divisionBlock == 0) divisionBlock = 1;
    for(int index = thread * divisionBlock; index < (thread+1)*divisionBlock; index++ ) {
      if(index >= MATRIX_SIZE) break;
      int row = index;
      for (int columnIndex = 0; columnIndex < MATRIX_SIZE; columnIndex++) {
        sum = 0;
        for (int dualIndex = 0; dualIndex < MATRIX_SIZE; dualIndex++) {
          sum += matrixA[row][dualIndex] * matrixB[dualIndex][columnIndex];
        }
        matrixC[row][columnIndex] = sum;
      }
    }
    // If threads and Matrix size aren't divisible then we can make threads take on additional
    // tasks here.
    if(MATRIX_SIZE > NUMBER_THREADS && MATRIX_SIZE%NUMBER_THREADS != 0
      && MATRIX_SIZE != divisionBlock){
      for(int index = thread + NUMBER_THREADS * divisionBlock;
          index < (thread + NUMBER_THREADS * divisionBlock + divisionBlock); index++ ) {
        if(index >= MATRIX_SIZE) break;
        int row = index;
        for (int columnIndex = 0; columnIndex < MATRIX_SIZE; columnIndex++) {
          sum = 0;
          for (int dualIndex = 0; dualIndex < MATRIX_SIZE; dualIndex++) {
            sum += matrixA[row][dualIndex] * matrixB[dualIndex][columnIndex];
          }
          matrixC[row][columnIndex] = sum;
        }
      }
    }
  }
}
```

Figure 10. Thread Task Implementation

Threads	Time (s)
1	55.81
10	11.04
50	10.24
100	10.61
250	10
500	10.38
750	12.6
1000	9.82
1250	10.29
1500	10.35
1750	10.44
2000	10

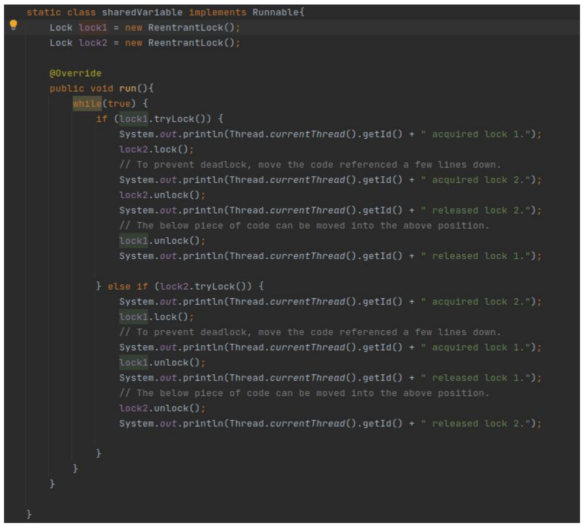
Table 1. Data points for varied threads

Matrix Size	Parallel Time	Sequential Time
100	0.1635	0.0741
200	0.03858	0.02343
500	0.10064	0.42912
1000	1.04087	4.80481
2000	9.98	69.99
4000	95.52	611.85

Table 2. Data points for varied sizes

Appendix B.

Question 2 Implementation

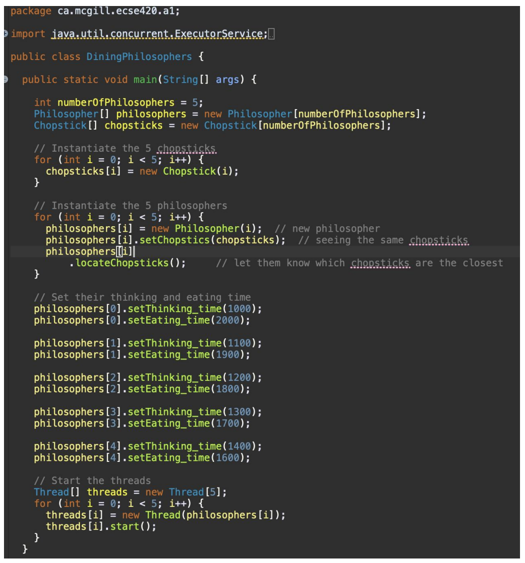


```java
static class sharedVariable implements Runnable{
    Lock lock1 = new ReentrantLock();
    Lock lock2 = new ReentrantLock();

    @Override
    public void run(){
        while(true) {
            if (lock1.tryLock()) {
                System.out.println(Thread.currentThread().getId() + " acquired lock 1.");
                lock2.lock();
                // To prevent deadlock, move the code referenced a few lines down.
                System.out.println(Thread.currentThread().getId() + " acquired lock 2.");
                lock2.unlock();
                System.out.println(Thread.currentThread().getId() + " released lock 2.");
                // The below piece of code can be moved into the above position.
                lock1.unlock();
                System.out.println(Thread.currentThread().getId() + " released lock 1.");

            } else if (lock2.tryLock()) {
                System.out.println(Thread.currentThread().getId() + " acquired lock 2.");
                lock1.lock();
                // To prevent deadlock, move the code referenced a few lines down.
                System.out.println(Thread.currentThread().getId() + " acquired lock 1.");
                lock1.unlock();
                System.out.println(Thread.currentThread().getId() + " released lock 1.");
                // The below piece of code can be moved into the above position.
                lock2.unlock();
                System.out.println(Thread.currentThread().getId() + " released lock 2.");

            }
        }
    }

}
```

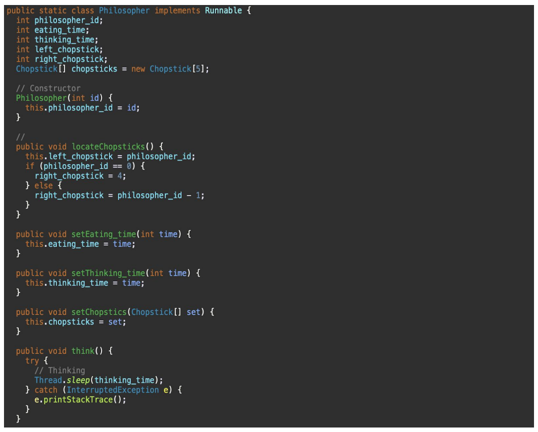
Figure 11. Deadlock Implementation

Appendix C.

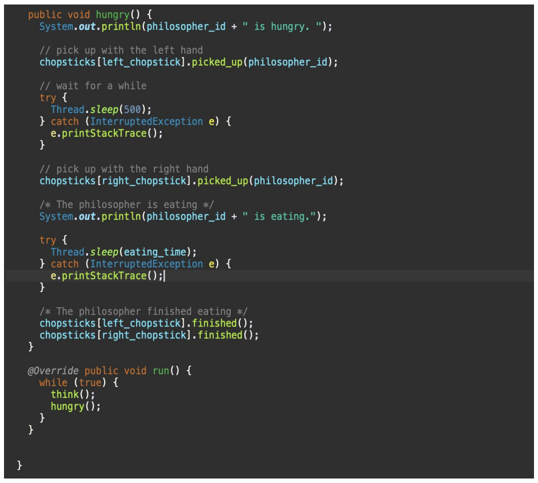
Regarding Question 3



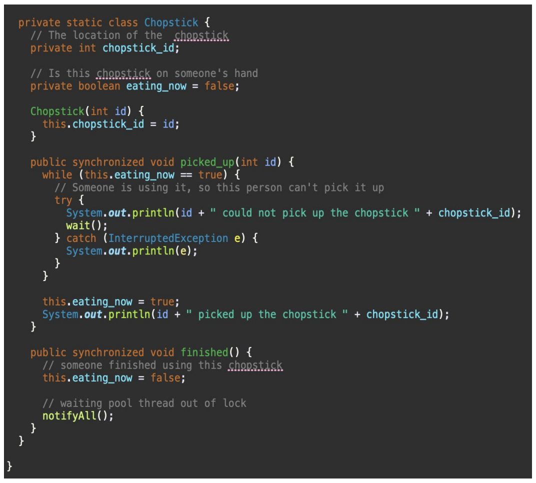
```java
package ca.mcgill.ecse420.a1;

import java.util.concurrent.ExecutorService;

public class DiningPhilosophers {

  public static void main(String[] args) {

    int numberOfPhilosophers = 5;
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    Chopstick[] chopsticks = new Chopstick[numberOfPhilosophers];

    // Instantiate the 5 chopsticks
    for (int i = 0; i < 5; i++) {
      chopsticks[i] = new Chopstick(i);
    }

    // Instantiate the 5 philosophers
    for (int i = 0; i < 5; i++) {
      philosophers[i] = new Philosopher(i);  // new philosopher
      philosophers[i].setChopstics(chopsticks);  // seeing the same chopsticks
      philosophers[i]
          .locateChopsticks();     // let them know which chopsticks are the closest
    }

    // Set their thinking and eating time
    philosophers[0].setThinking_time(1000);
    philosophers[0].setEating_time(2000);

    philosophers[1].setThinking_time(1100);
    philosophers[1].setEating_time(1900);

    philosophers[2].setThinking_time(1200);
    philosophers[2].setEating_time(1800);

    philosophers[3].setThinking_time(1300);
    philosophers[3].setEating_time(1700);

    philosophers[4].setThinking_time(1400);
    philosophers[4].setEating_time(1600);

    // Start the threads
    Thread[] threads = new Thread[5];
    for (int i = 0; i < 5; i++) {
      threads[i] = new Thread(philosophers[i]);
      threads[i].start();
    }
  }
```



```java
public static class Philosopher implements Runnable {
  int philosopher_id;
  int eating_time;
  int thinking_time;
  int left_chopstick;
  int right_chopstick;
  Chopstick[] chopsticks = new Chopstick[5];

  // Constructor
  Philosopher(int id) {
    this.philosopher_id = id;
  }

  //
  public void locateChopsticks() {
    this.left_chopstick = philosopher_id;
    if (philosopher_id == 0) {
      right_chopstick = 4;
    } else {
      right_chopstick = philosopher_id - 1;
    }
  }

  public void setEating_time(int time) {
    this.eating_time = time;
  }

  public void setThinking_time(int time) {
    this.thinking_time = time;
  }

  public void setChopstics(Chopstick[] set) {
    this.chopsticks = set;
  }

  public void think() {
    try {
      // Thinking
      Thread.sleep(thinking_time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
```

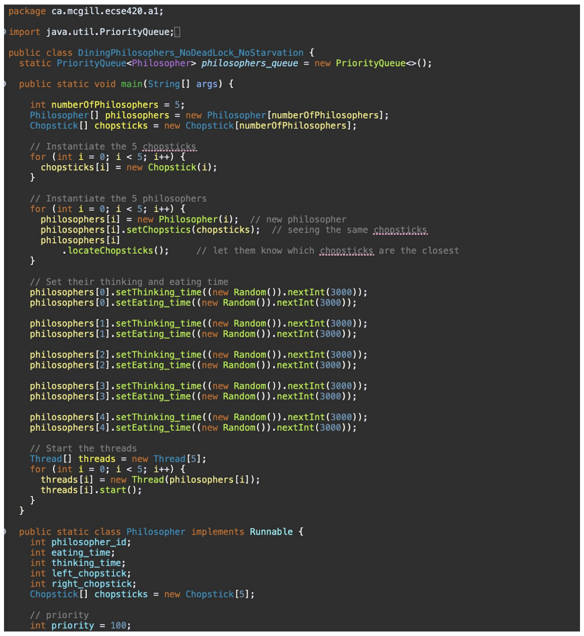


```java
  public void hungry() {
    System.out.println(philosopher_id + " is hungry. ");

    // pick up with the left hand
    chopsticks[left_chopstick].picked_up(philosopher_id);

    // wait for a while
    try {
      Thread.sleep(500);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    // pick up with the right hand
    chopsticks[right_chopstick].picked_up(philosopher_id);

    /* The philosopher is eating */
    System.out.println(philosopher_id + " is eating.");

    try {
      Thread.sleep(eating_time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    /* The philosopher finished eating */
    chopsticks[left_chopstick].finished();
    chopsticks[right_chopstick].finished();
  }

  @Override public void run() {
    while (true) {
      think();
      hungry();
    }
  }

}
```

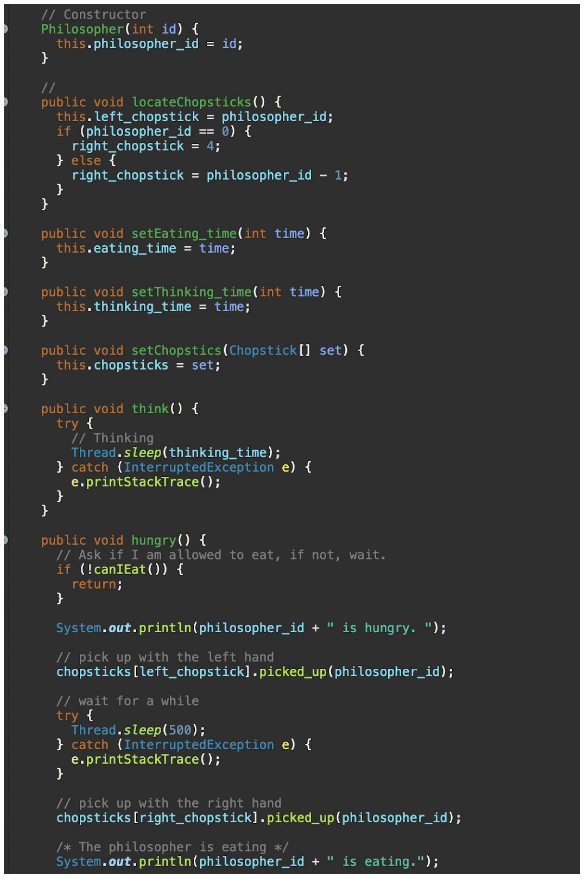


```java
  private static class Chopstick {
    // The location of the  chopstick
    private int chopstick_id;

    // Is this chopstick on someone's hand
    private boolean eating_now = false;

    Chopstick(int id) {
      this.chopstick_id = id;
    }

    public synchronized void picked_up(int id) {
      while (this.eating_now == true) {
        // Someone is using it, so this person can't pick it up
        try {
          System.out.println(id + " could not pick up the chopstick " + chopstick_id);
          wait();
        } catch (InterruptedException e) {
          System.out.println(e);
        }
      }

      this.eating_now = true;
      System.out.println(id + " picked up the chopstick " + chopstick_id);
    }

    public synchronized void finished() {
      // someone finished using this chopstick
      this.eating_now = false;

      // waiting pool thread out of lock
      notifyAll();
    }
  }

}
```

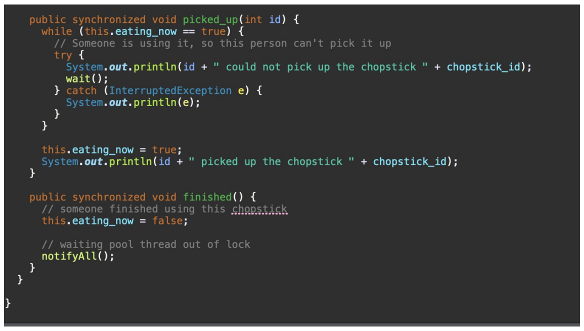
Figure 12. Dining Philosophers Implementation



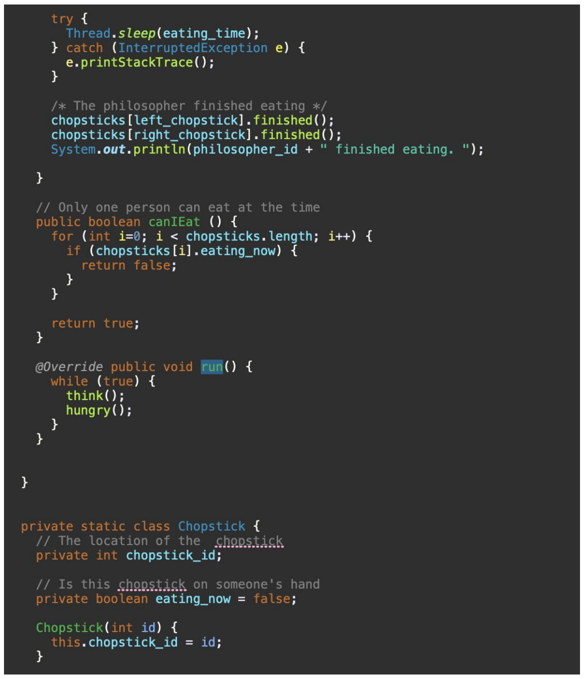
```java
package ca.mcgill.ecse420.a1;

import java.util.PriorityQueue;

public class DiningPhilosophers_NoDeadLock_NoStarvation {
  static PriorityQueue<Philosopher> philosophers_queue = new PriorityQueue<>();

  public static void main(String[] args) {

    int numberOfPhilosophers = 5;
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    Chopstick[] chopsticks = new Chopstick[numberOfPhilosophers];

    // Instantiate the 5 chopsticks
    for (int i = 0; i < 5; i++) {
      chopsticks[i] = new Chopstick(i);
    }

    // Instantiate the 5 philosophers
    for (int i = 0; i < 5; i++) {
      philosophers[i] = new Philosopher(i);  // new philosopher
      philosophers[i].setChopstics(chopsticks);  // seeing the same chopsticks
      philosophers[i]
          .locateChopsticks();     // let them know which chopsticks are the closest
    }

    // Set their thinking and eating time
    philosophers[0].setThinking_time((new Random()).nextInt(3000));
    philosophers[0].setEating_time((new Random()).nextInt(3000));

    philosophers[1].setThinking_time((new Random()).nextInt(3000));
    philosophers[1].setEating_time((new Random()).nextInt(3000));

    philosophers[2].setThinking_time((new Random()).nextInt(3000));
    philosophers[2].setEating_time((new Random()).nextInt(3000));

    philosophers[3].setThinking_time((new Random()).nextInt(3000));
    philosophers[3].setEating_time((new Random()).nextInt(3000));

    philosophers[4].setThinking_time((new Random()).nextInt(3000));
    philosophers[4].setEating_time((new Random()).nextInt(3000));

    // Start the threads
    Thread[] threads = new Thread[5];
    for (int i = 0; i < 5; i++) {
      threads[i] = new Thread(philosophers[i]);
      threads[i].start();
    }
  }

  public static class Philosopher implements Runnable {
    int philosopher_id;
    int eating_time;
    int thinking_time;
    int left_chopstick;
    int right_chopstick;
    Chopstick[] chopsticks = new Chopstick[5];

    // priority
    int priority = 100;
```



```java
    // Constructor
    Philosopher(int id) {
      this.philosopher_id = id;
    }

    //
    public void locateChopsticks() {
      this.left_chopstick = philosopher_id;
      if (philosopher_id == 0) {
        right_chopstick = 4;
      } else {
        right_chopstick = philosopher_id - 1;
      }
    }

    public void setEating_time(int time) {
      this.eating_time = time;
    }

    public void setThinking_time(int time) {
      this.thinking_time = time;
    }

    public void setChopstics(Chopstick[] set) {
      this.chopsticks = set;
    }

    public void think() {
      try {
        // Thinking
        Thread.sleep(thinking_time);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    public void hungry() {
      // Ask if I am allowed to eat, if not, wait.
      if (!canIEat()) {
        return;
      }

      System.out.println(philosopher_id + " is hungry. ");

      // pick up with the left hand
      chopsticks[left_chopstick].picked_up(philosopher_id);

      // wait for a while
      try {
        Thread.sleep(500);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      // pick up with the right hand
      chopsticks[right_chopstick].picked_up(philosopher_id);

      /* The philosopher is eating */
      System.out.println(philosopher_id + " is eating.");
```



```java
    public synchronized void picked_up(int id) {
      while (this.eating_now == true) {
        // Someone is using it, so this person can't pick it up
        try {
          System.out.println(id + " could not pick up the chopstick " + chopstick_id);
          wait();
        } catch (InterruptedException e) {
          System.out.println(e);
        }
      }

      this.eating_now = true;
      System.out.println(id + " picked up the chopstick " + chopstick_id);
    }

    public synchronized void finished() {
      // someone finished using this chopstick
      this.eating_now = false;

      // waiting pool thread out of lock
      notifyAll();
    }
  }
}
```



```
    try {
      Thread.sleep(eating_time);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    /* The philosopher finished eating */
    chopsticks[left_chopstick].finished();
    chopsticks[right_chopstick].finished();
    System.out.println(philosopher_id + " finished eating. ");

  }

  // Only one person can eat at the time
  public boolean canIEat () {
    for (int i=0; i < chopsticks.length; i++) {
      if (chopsticks[i].eating_now) {
        return false;
      }
    }

    return true;
  }

  @Override public void run() {
    while (true) {
      think();
      hungry();
    }
  }


}


private static class Chopstick {
  // The location of the  chopstick
  private int chopstick_id;

  // Is this chopstick on someone's hand
  private boolean eating_now = false;

  Chopstick(int id) {
    this.chopstick_id = id;
  }
```

Figure 13. Dining Philosophers Without Deadlock Implementation