Title

TAs Project

Sleeping Teacher Assistant

الاسم : بكر ابوحسيبه زكي حسن
ID:202000206

الاسم: عبدالله خالد عبدالشافي
ID: 201900459

الاسم: عمر خالد صلاح الدين عبد الجواد
ID: 202000593

الاسم: محمد فتحي فرج احمد
ID:201900717

الاسم : محمد سعيد إبراهيم عبدالغني
ID: 20180498

الاسم : عبدالله ممدوح السيد ابراهيم
ID:202000554

الاسم : محمد ابراهيم عبدالفتاح محمد
ID:202000718

PROBLEM STATEMENT:

The project represents a solution to the sleeping teaching assistant problem using Java and a graphical user interface (GUI). The problem simulates a scenario where students seek help from teaching assistants. The teaching assistant office is just a single room with chairs. When the teaching assistant is busy in assisting a student, the other students have to wait in the waiting area which is outside the office. If the chairs available in the waiting area are empty the student can sit on the available chair and wait for their chance. Also, if all the chairs are not available then the student will leave and come later.
The goal is to manage the synchronization and concurrency of students and teaching assistants using threads, semaphores, and locks.

Synchronization Techniques:

1-Semaphores:

- Two semaphores are utilized: one for the TA (available) and another for students (chairs).
- The available semaphore controls the number of TAs available for assistance.
- The chairs semaphore manages the number of chairs in the waiting area, limiting the number of students who can wait.

2-Mutual Exclusion Locks:

- A Reentrant Lock (entlock) is used to provide mutual exclusion when accessing critical sections of the code.
- Conditions (self) associated with the lock are used to coordinate between threads, allowing them to wait and signal appropriately.

3-Threads:

- Multiple threads are created to represent both students and TAs. Each student and TA run in their respective threads.

Simulation Overview:

1-TA Actions:

- The TA has three actions: checking for a student, sleeping if none are waiting, and waking up when a student arrives.
- The TA uses the take() method to signal and wake up when a student arrives.
- After assisting a student, the TA releases resources (semaphores) and checks for the next student.

2-Student Actions:

- Students perform three actions: arriving, checking for available chairs, and waiting for assistance.
- Students use semaphores (chairs and available) to control access to resources.
- The code handles scenarios where a student leaves if no chairs are available or waits for assistance if a chair is available.

3-Mutex Lock Mechanisms:

- Mutex locks are employed to protect the modification of the number of chairs.
- This ensures that only one thread can update the count of available chairs at a time, preventing race conditions.

Possible Scenarios:

Scenario One:

There will be zero students coming to visit the TA and the TA will check the hallway outside his office to see if there are any students seated and waiting for him. If there are none, the TA will sleep in his office.

Scenario Two:

When a student arrives at the TA’s office and finds the TA sleeping. Then the student will awaken the TA and ask for help. When the TA assists the student, the student's semaphore changes from 0 to 1 and waits for the TA's semaphore. When the TA finishes helping one student, he will check if there is any other student waiting in the hallway. If yes, he will help the next student and if not, TA goes back to sleeping and TA's semaphore becomes 1 and awaits student's semaphore.

Scenario Three:

When a student arrives while the TA is busy with another student. Then the student who arrived will have to check if the TA is busy. If the TA is busy, the student will have to wait seated outside in the hallway until the TA is done with his session. When the TA completes his session, the student seated outside will be called in by the TA for a review session. Once all students have finished their sessions and left the TA's office, the TA will go back to sleep after making sure no students are waiting.

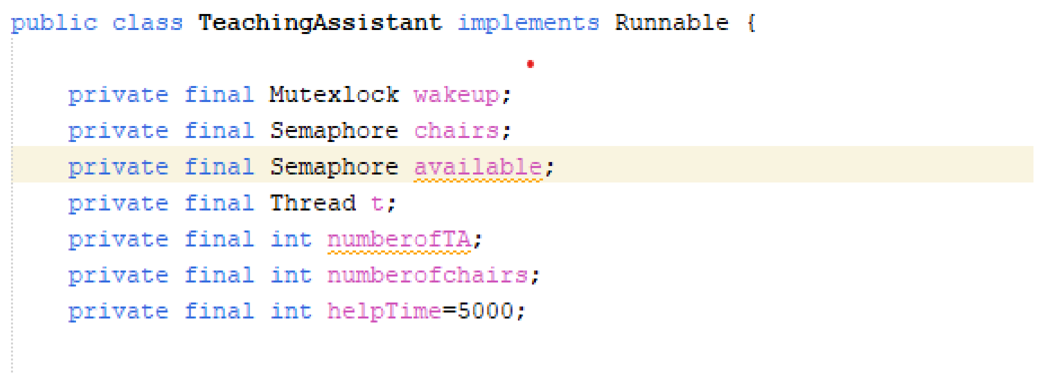
Scenario Four:

When a student arrives while the TA is busy in a review session, and all the seats in the hallway are occupied. Then, student will have to leave the hallway and come back later. When the student comes back, eventually, and there is a seat available, he will take a seat and wait for his turn with the TA.

What we actually did:

We build 4 classes TeachingAssistant, Student, Mutexlock and Work

1- TeachingAssistant Class:



```
public class TeachingAssistant implements Runnable {

    private final Mutexlock wakeup;
    private final Semaphore chairs;
    private final Semaphore available;
    private final Thread t;
    private final int numberofTA;
    private final int numberofchairs;
    private final int helpTime=5000;
```

Instance Variables:

- wakeup: An instance of the Mutexlock class. It seems to be intended for synchronization with students, but its role is not entirely clear.

- chairs: A Semaphore representing the number of available chairs. Students will acquire permits to sit on chairs.

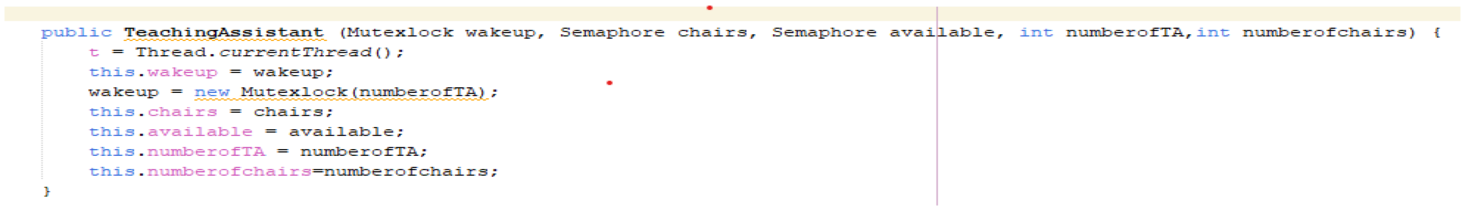
- available: Another Semaphore representing the availability of the TA. It is likely used to control how many students can be assisted simultaneously.

- t: An instance of the Thread class representing the current thread (TA thread).

- numberofTA: An integer representing the total number of Teaching Assistants.

- numberofchairs: An integer representing the total number of chairs in the waiting area.

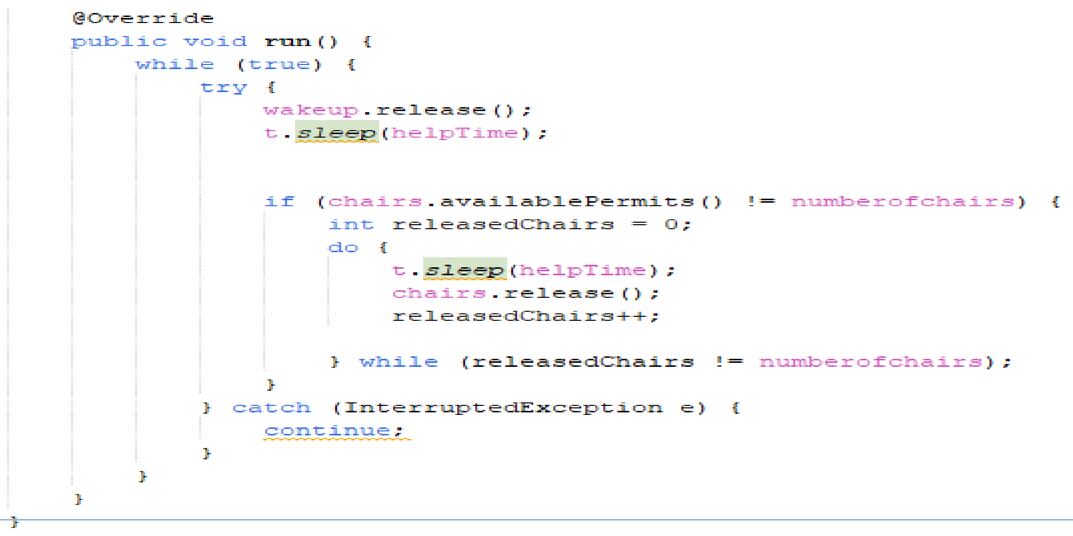
- helpTime: An integer constant representing the time the TA spends helping a student (in milliseconds).



```java
public TeachingAssistant (Mutexlock wakeup, Semaphore chairs, Semaphore available, int numberofTA,int numberofchairs) {
    t = Thread.currentThread();
    this.wakeup = wakeup;
    wakeup = new Mutexlock(numberofTA);
    this.chairs = chairs;
    this.available = available;
    this.numberofTA = numberofTA;
    this.numberofchairs=numberofchairs;
}
```

2- Constructor:

Initializes the instance variables with the values provided in the constructor.

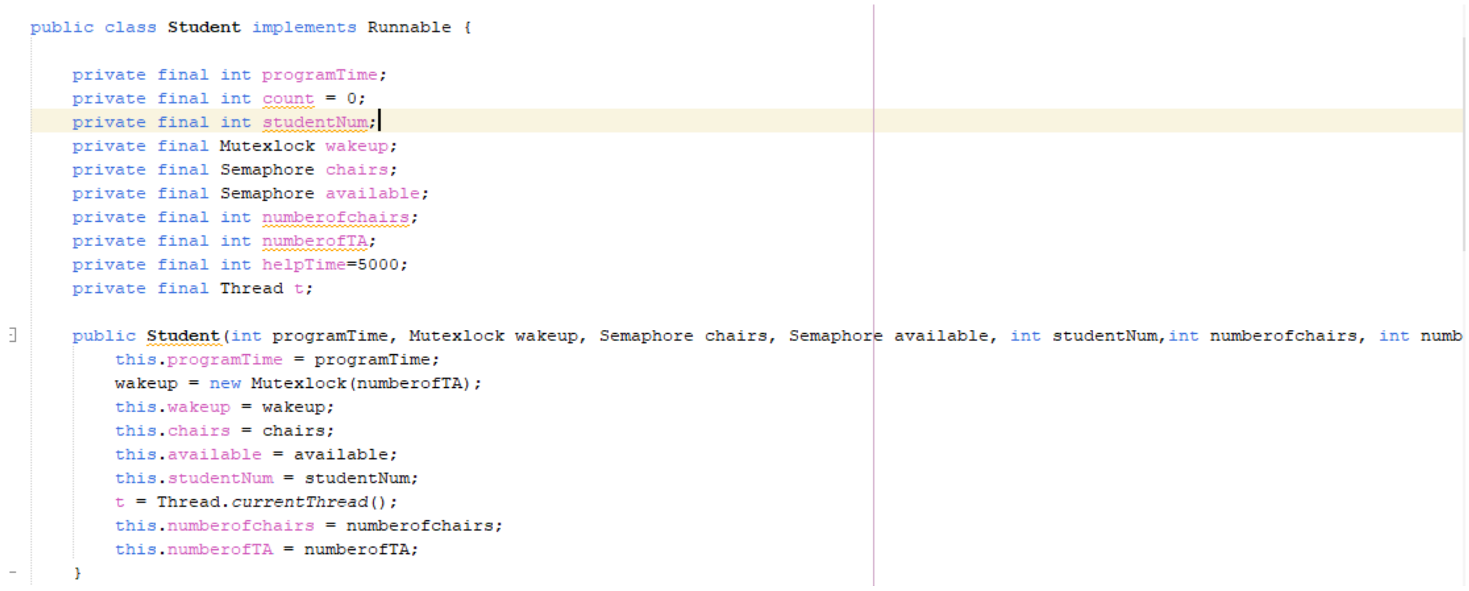


```java
    @Override
    public void run() {
        while (true) {
            try {
                wakeup.release();
                t.sleep(helpTime);


                if (chairs.availablePermits() != numberofchairs) {
                    int releasedChairs = 0;
                    do {
                        t.sleep(helpTime);
                        chairs.release();
                        releasedChairs++;

                    } while (releasedChairs != numberofchairs);
                }
            } catch (InterruptedException e) {
                continue;
            }
        }
    }
}
```

3- run Method:

Infinite loop (while (true)) representing the continuous operation of the TA.
The TA releases the wakeup mutex (this behavior might need clarification, as releasing a mutex without acquiring it first may lead to unexpected behavior).
Sleeps for a specified helpTime.

2-Student Class



```
public class Student implements Runnable {

    private final int programTime;
    private final int count = 0;
    private final int studentNum;
    private final Mutexlock wakeup;
    private final Semaphore chairs;
    private final Semaphore available;
    private final int numberofchairs;
    private final int numberofTA;
    private final int helpTime=5000;
    private final Thread t;

    public Student(int programTime, Mutexlock wakeup, Semaphore chairs, Semaphore available, int studentNum,int numberofchairs, int numb
        this.programTime = programTime;
        wakeup = new Mutexlock(numberofTA);
        this.wakeup = wakeup;
        this.chairs = chairs;
        this.available = available;
        this.studentNum = studentNum;
        t = Thread.currentThread();
        this.numberofchairs = numberofchairs;
        this.numberofTA = numberofTA;
    }
```

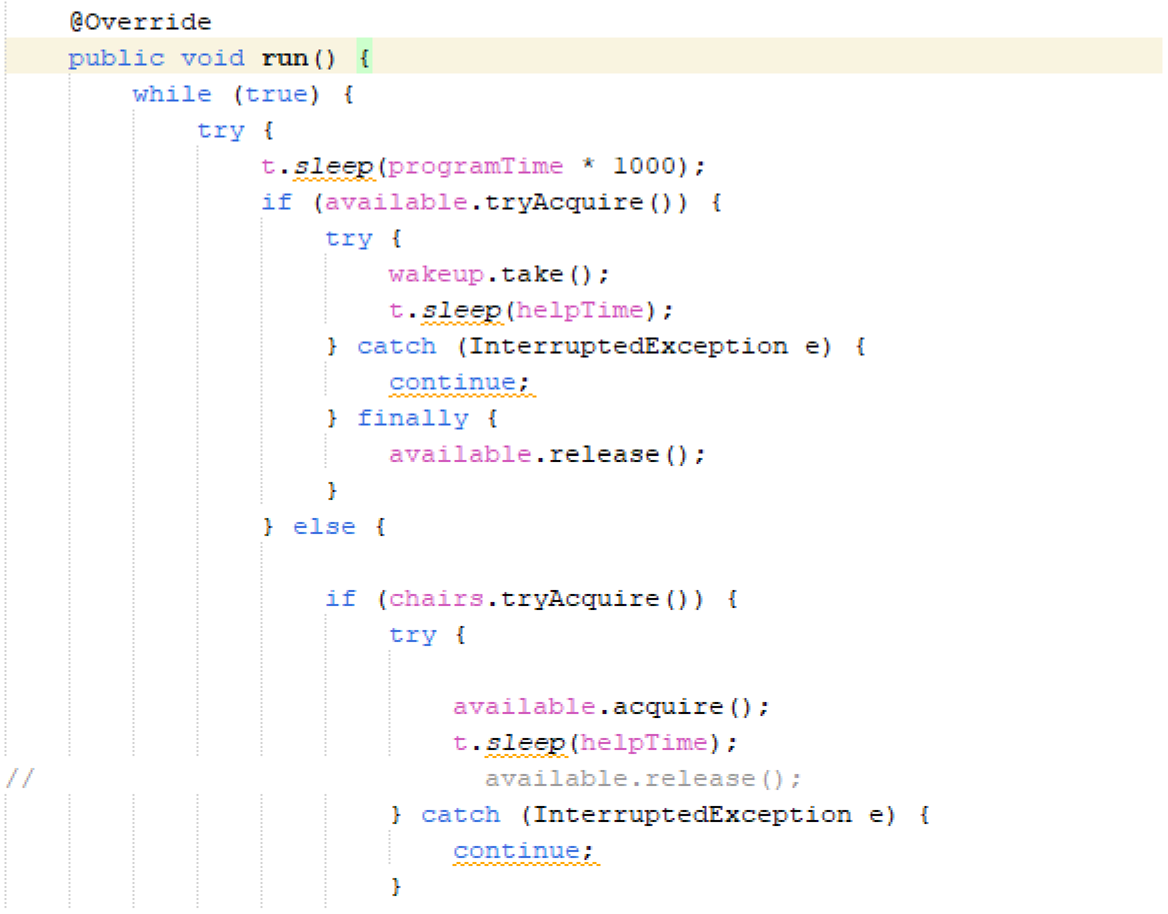
Instance Variables:

- programTime: An integer representing the time interval between student arrivals (in seconds).
- count: An integer variable declared but not used in the code.
- studentNum: An integer representing the unique identifier for each student.
- wakeup: An instance of the Mutexlock class, presumably used for synchronization with the TA.
- chairs: A Semaphore representing the number of available chairs in the hallway.
- available: A Semaphore representing the availability of the TA.
- numberofchairs: An integer representing the total number of chairs in the hallway.

- numberofTA: An integer representing the total number of Teaching Assistants.
- helpTime: An integer constant representing the time the TA spends helping a student (in milliseconds).
- t: An instance of the Thread class representing the current thread (student thread).

Constructor:

Initializes the instance variables with the values provided in the constructor.



```
@Override
public void run() {
    while (true) {
        try {
            t.sleep(programTime * 1000);
            if (available.tryAcquire()) {
                try {
                    wakeup.take();
                    t.sleep(helpTime);
                } catch (InterruptedException e) {
                    continue;
                } finally {
                    available.release();
                }
            } else {

                if (chairs.tryAcquire()) {
                    try {

                        available.acquire();
                        t.sleep(helpTime);
//                          available.release();
                    } catch (InterruptedException e) {
                        continue;
                    }
```

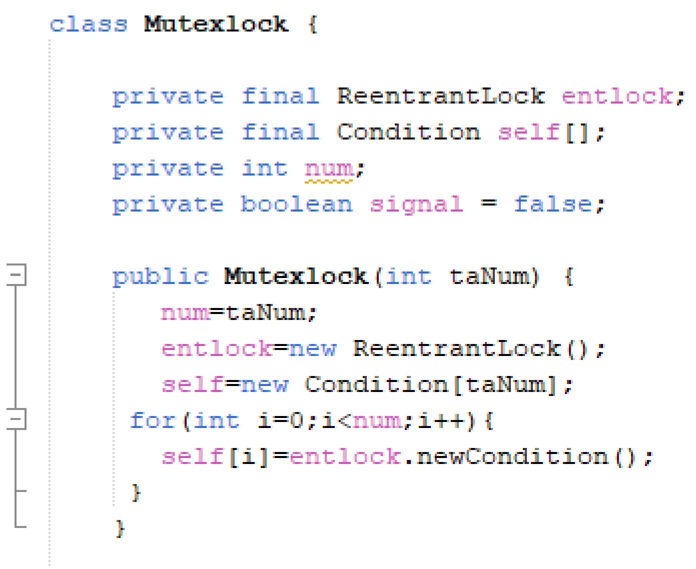
run Method:

Infinite loop (while (true)) representing the continuous operation of the student. Sleeps for a specified programTime before attempting to enter the TA's office.

Two Scenarios:

- If a permit is acquired from the available semaphore (indicating the TA is available), the student wakes up the TA, gets assistance, and releases the permit.
- If a permit is not acquired, the student checks for the availability of chairs.
- If a chair is available, the student acquires a chair, waits for assistance, and releases the chair.
- If no chair is available, the student waits outside the TA's office.
- The loop breaks if an InterruptedException occurs.

3-Mutexlock Class:



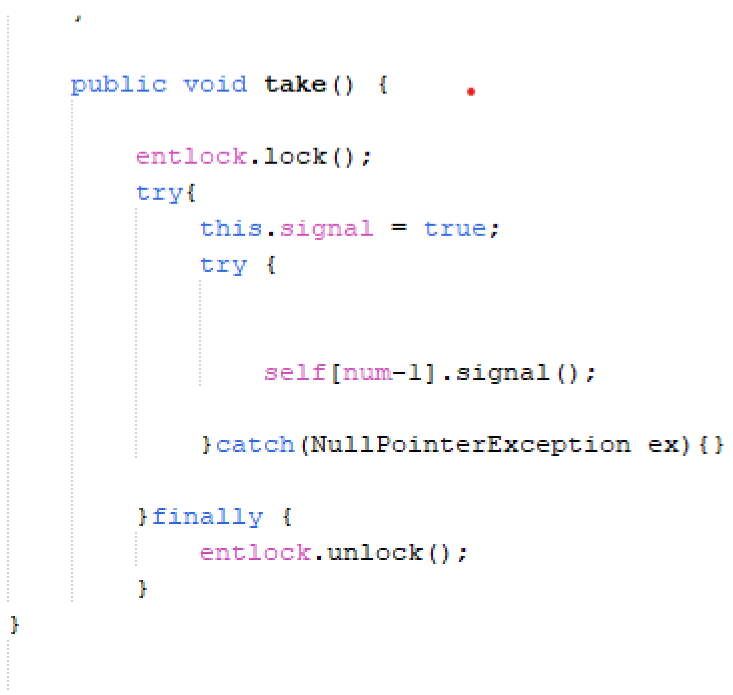
```
class Mutexlock {

    private final ReentrantLock entlock;
    private final Condition self[];
    private int num;
    private boolean signal = false;

    public Mutexlock(int taNum) {
       num=taNum;
       entlock=new ReentrantLock();
       self=new Condition[taNum];
     for(int i=0;i<num;i++){
       self[i]=entlock.newCondition();
     }
    }
```

Instance Variables:

- entlock: An instance of ReentrantLock used as the underlying lock.
- self: An array of Condition variables associated with the lock.
- num: The number of conditions and the size of the array.
- signal: A boolean variable indicating whether the lock has been acquired.

Constructor:

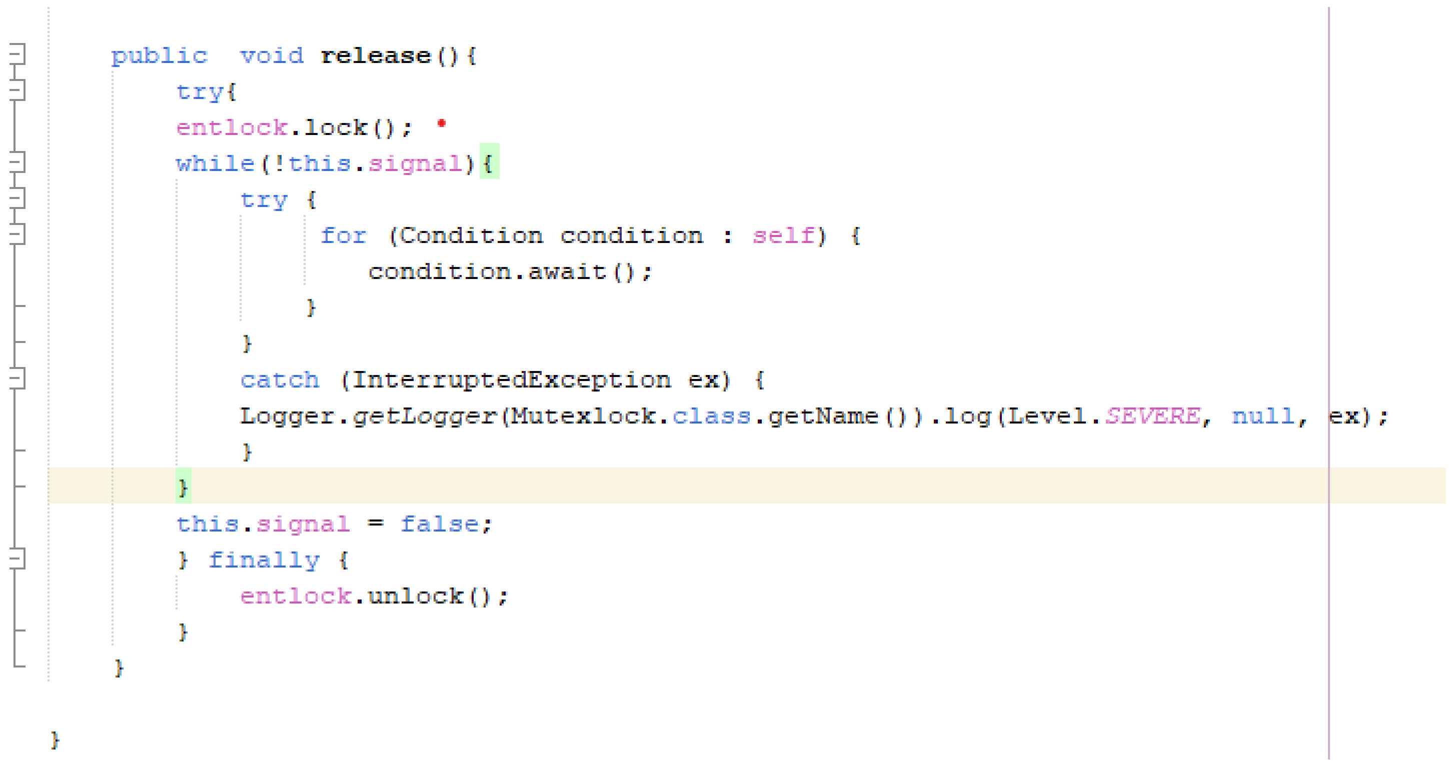
Initializes the entlock, self array, and num based on the provided taNum.



```
    ,

    public void take() {

        entlock.lock();
        try{
            this.signal = true;
            try {


                self[num-1].signal();

            }catch(NullPointerException ex){}

        }finally {
            entlock.unlock();
        }
}
```

take Method:

- Acquires the lock (entlock).
- Sets signal to true, indicating that the lock has been acquired.
- Signals the last condition in the array (self[num - 1]).
- Releases the lock.

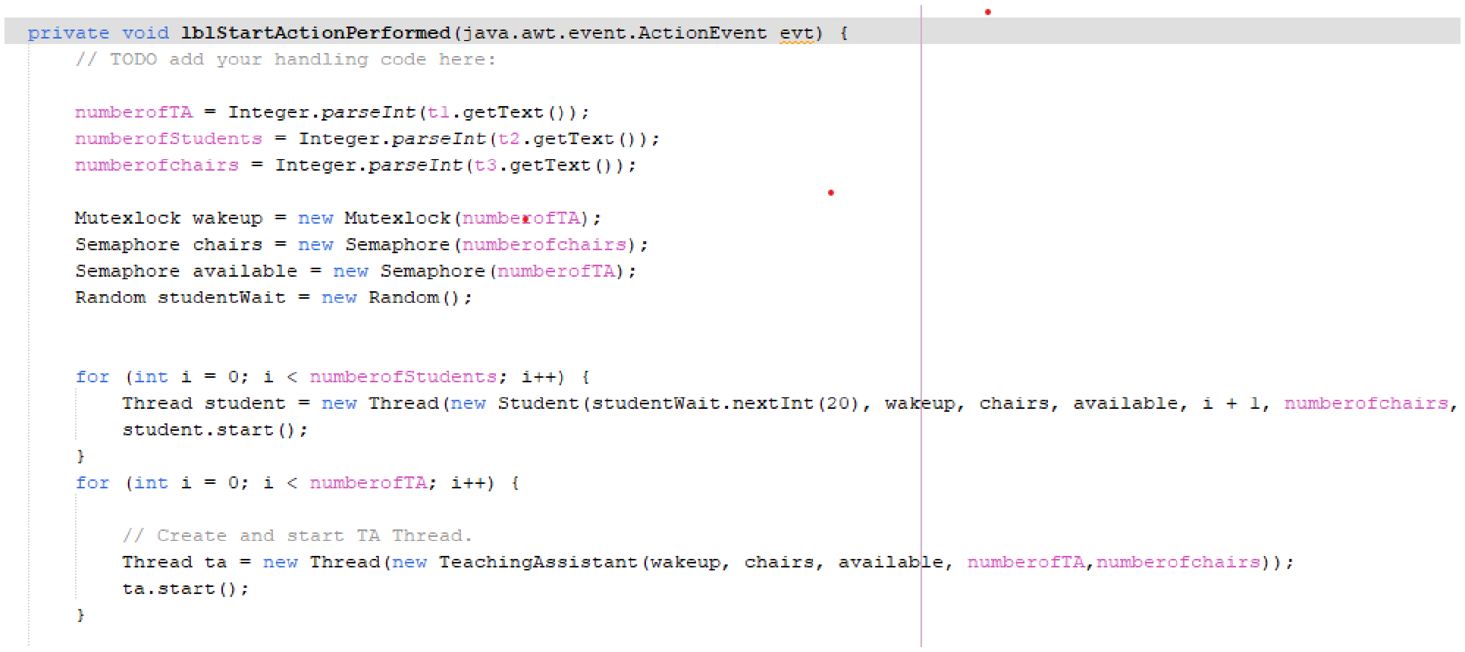


```
    public  void release(){
        try{
        entlock.lock();
        while(!this.signal){
            try {
                for (Condition condition : self) {
                    condition.await();
                }
            }
            catch (InterruptedException ex) {
            Logger.getLogger(Mutexlock.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
        this.signal = false;
        } finally {
            entlock.unlock();
        }
    }

}
```

release Method:

- Acquires the lock (entlock).
- Waits in a loop until signal is true.
- If interrupted, logs the exception.
- Resets signal to false.
- Releases the lock.

4- Work (GUI) Class:



```java
private void lblStartActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:

    numberofTA = Integer.parseInt(t1.getText());
    numberofStudents = Integer.parseInt(t2.getText());
    numberofchairs = Integer.parseInt(t3.getText());

    Mutexlock wakeup = new Mutexlock(numberofTA);
    Semaphore chairs = new Semaphore(numberofchairs);
    Semaphore available = new Semaphore(numberofTA);
    Random studentWait = new Random();


    for (int i = 0; i < numberofStudents; i++) {
        Thread student = new Thread(new Student(studentWait.nextInt(20), wakeup, chairs, available, i + 1, numberofchairs,
        student.start();
    }
    for (int i = 0; i < numberofTA; i++) {

        // Create and start TA Thread.
        Thread ta = new Thread(new TeachingAssistant(wakeup, chairs, available, numberofTA,numberofchairs));
        ta.start();
    }
```

Extract Parameters:

Extracts the values entered in the text fields for the number of TAs (numberofTA), students (numberofStudents), and chairs (numberofchairs).
Semaphore and Mutexlock Initialization:

Creates instances of Mutexlock (wakeup) and two Semaphore instances (chairs and available).

wakeup: Used to manage the wake-up signal for TAs.

chairs: Represents the chairs available in the waiting area.

available: Represents the available TAs.

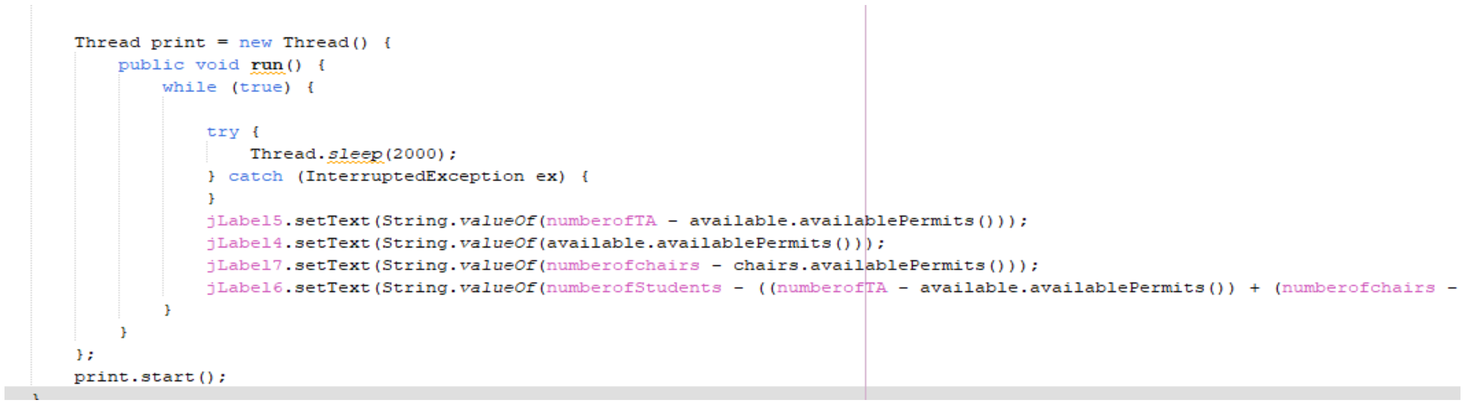
Random Generator:

Initializes a Random instance (studentWait) for generating random program times for students.

Student Threads:

Iterates over the number of students (numberofStudents).
Creates a new Thread for each student, passing random program time, wakeup Mutexlock, chairs Semaphore, available Semaphore, student number, number of chairs, and number of TAs.
Starts each student thread.
TA Threads:

Iterates over the number of TAs (numberofTA).
Creates a new Thread for each TA, passing wakeup Mutexlock, chairs Semaphore, available Semaphore, number of TAs, and number of chairs.
Starts each TA thread.



```
Thread print = new Thread() {
    public void run() {
        while (true) {

            try {
                Thread.sleep(2000);
            } catch (InterruptedException ex) {
            }
            jLabel5.setText(String.valueOf(numberofTA - available.availablePermits()));
            jLabel4.setText(String.valueOf(available.availablePermits()));
            jLabel7.setText(String.valueOf(numberofchairs - chairs.availablePermits()));
            jLabel6.setText(String.valueOf(numberofStudents - ((numberofTA - available.availablePermits()) + (numberofchairs -
        }
    }
};
print.start();
```

Thread Initialization:

Creates a new thread (print) using an anonymous subclass of Thread.
Overrides the run method of the thread to define the code that will be executed when the thread is started.

Infinite Loop:

Inside the run method, there's an infinite loop (while (true)) that repeatedly executes the following steps.

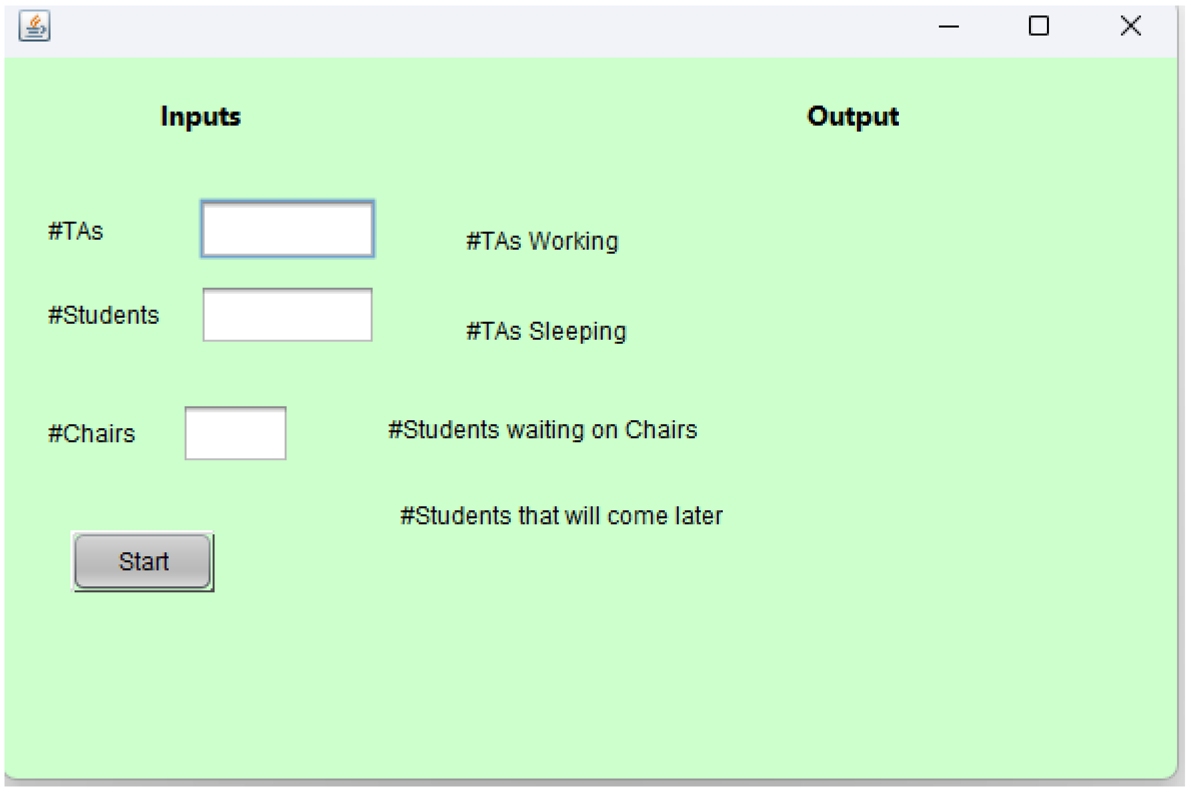
Thread Sleep:

The thread sleeps for 2000 milliseconds (2 seconds) in each iteration using Thread.sleep(2000). This introduces a delay between updates, preventing the GUI from updating too frequently.

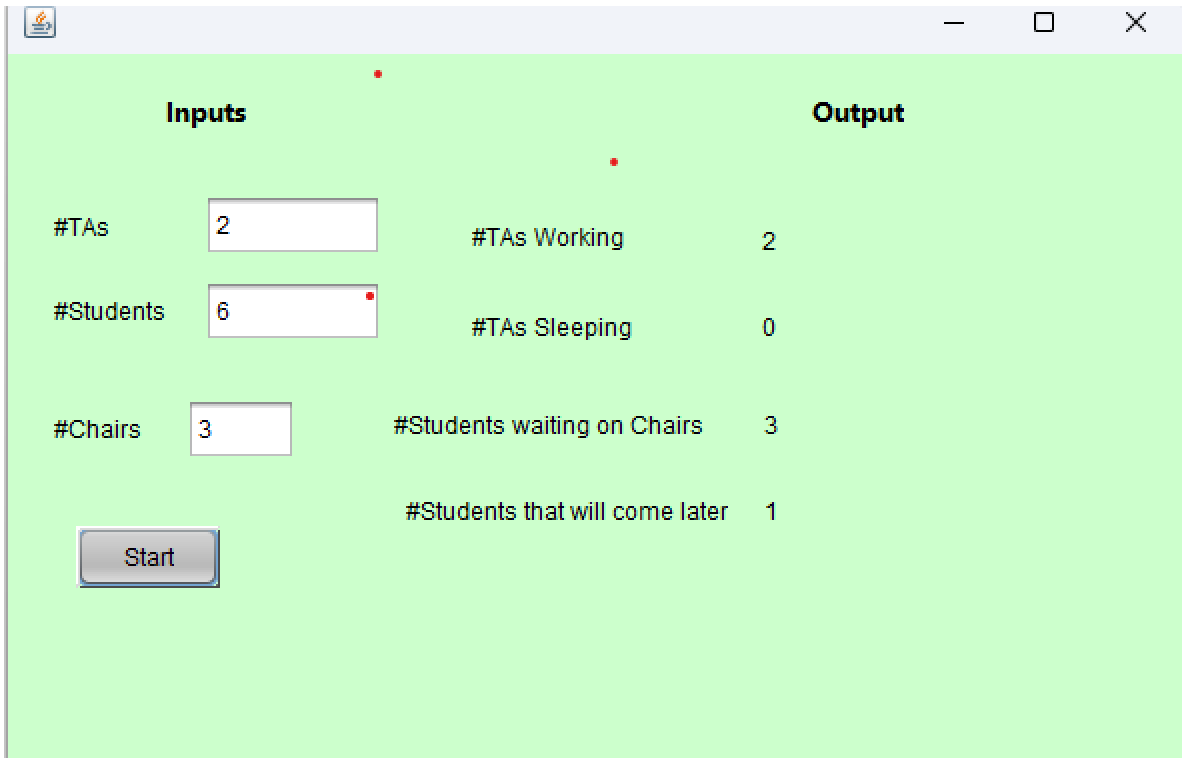
Updating JLabels:

Updates the text of various JLabels (jLabel5, jLabel4, jLabel7, jLabel6) with information about the simulation state.

The GUI before running:



The GUI while running:



#TAs	2
#Students	6
#Chairs	3

#TAs Working	2
#TAs Sleeping	0
#Students waiting on Chairs	3
#Students that will come later	1

Team Members role:

بكر ابوحسيبة زكي -------------------- documentation, TeachingAssistant class

عبدالله خالد عبدالشافي ------------------ Student Class

عمر خالد صلاح الدين عبد الجواد--------------------------- Student Class

محمد فتحي فرج احمد------------------------------------- Mutexlock Class

محمد سعيد إبراهيم عبدالغني------------------------------ Mutexlock Class

محمد ابراهيم عبدالفتاح محمد ----------------------------- Work (GUI) Class

عبدالله ممدوح السيد ابراهيم ----------------------------- Work (GUI) Class