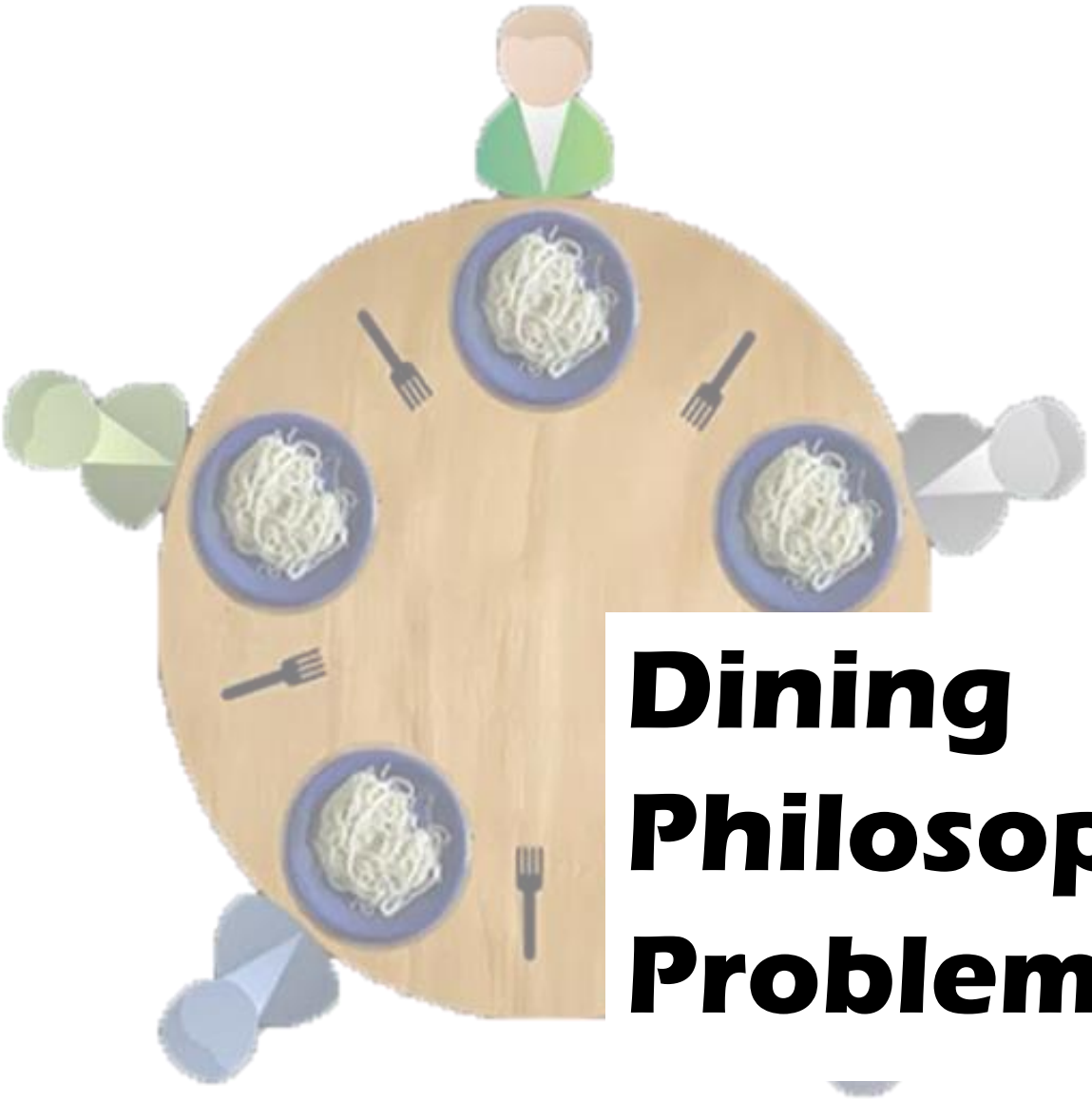

Dining Philosophers Problem

Name	ID	Level	Major	Minor
سارة سامح أبوبكر عبدالباسط	202000369	3	CS	IS
آية هاني قاسم توفيق	202000190	3	CS	IS
روان سعودي صلاح أحمد	202000332	3	CS	IS
هدى حامد مصطفى حامد	202001029	3	CS	IS
تقى أسامة حسن أحمد	202000229	3	CS	IS
آلاء رضا عبدالحميد متولي	202000140	3	CS	IS
أدهم أحمد عيد محمد	202000096	3	CS	IS

Table of contents:

1- Solution pseudocode

2- Examples of deadlock

3- How did we solve deadlock

4- Examples of starvation

5- How did we solve starvation

6- Explanation for real world application and how did we apply the problem

1- Solution pseudo code:

The Dining Philosopher Problem (as general definition) is the classical problem of synchronization which says that (k) philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of rice is placed at the center of the table along with one chopstick between each pair of philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

Solutions to the problem:
One solution is to use **a semaphore** to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The used solution in our problem is **catching NullPointerExceptions** during the phase where the Philosophers pick up their Chopsticks. As if an instance of a chopstick is not initiated right (initiated to NULL) starvation will take place because the philosopher won't know if this chopstick is in use or not so stealing may happen (to use the chopstick while other philosopher is using it), and as our solution depends on dividing our main contributors (Chopstick- as the resource- and the philosopher -as the thread-) into individual classes so we can control every single action done well.

2- Examples of deadlock:

In our problem deadlocks can raise if:

- every philosopher holds a left chopstick and waits perpetually for a right chopstick In this case philosopher P1 waits for the chopstick grabbed by philosopher P2 who is waiting for the chopstick of philosopher P3 and so forth, making a circular chain and Hold and wait conditions which raise deadlock

- If the status of the chopstick is not clear if it's in use or not -maybe not initiated _Null_ - so one philosopher steals from the other the chopstick to use it, while in problem statement philosophers are polite, they don't steal, here we talk about preemption which causes deadlocks.

3- How did we solve deadlock:

1- In (hold and wait) and circular chain problem we solved it by making a function that makes sure if one of the both chopstick is in use so it returns false and thus philosopher wait till this chopstick is available, and if one is available and the other is not so release the picked one and return false here we avoided circular chain problem and hold and wait which are main conditions in making deadlocks, returning false makes philosopher is waiting till both chopsticks are available

here in the opposite figures: pickupChopsticks () function is used to avoid circular chain. So, no philosopher will hold a chopstick blocks other philosophers from using it and no philosopher can eat and stay Hungary.

```
public boolean pickupChopsticks() {
        if (left_chopstick.inUse(chair_number) == false)
                left_chopstick.claim(chair_number);
        else
                return false;

        if (right_chopstick.inUse(chair_number) == false)
                right_chopstick.claim(chair_number);
        else {
                left_chopstick.release();
                return false;
        }

        return true;
}
```

2- We created two classes as mentioned before one is Chopstick and the other is Philosopher to control each object so catching deadlocks gets easier, in Chopstick class chopstick object is initiated to a number which indicates that no one own this chopstick in the current moment, so one philosopher can pick it up for eating here we prevent stealing or more likely (preemption), and in philosopher class we make sure that every chopstick instance is initiated by catching Null pointer in which causes deadlock

```
class Chopstick {
        static final int NO_OWNER = 5;

        private volatile int belongs_to = NO_OWNER;

        private volatile boolean chopstick = false;

        public boolean inUse(int chair_number) {
                if (belongs_to == NO_OWNER)
                        return false;
                else if (chair_number != belongs_to)
                        return true;
                else if (chopstick == true)
                        return true;

                return false;
        }
```

4- Examples of starvation:

- **In our problem starvation can raise if:**
  A philosopher waits forever to eat while other philosopher eats several times

5- How did we solve starvation

At the first place our solution is free from starvation because of being free from deadlocks, in addition to setting eating time and release chopsticks after this time ends so other philosopher can use the shared chopstick without waiting forever, a method in the Philosopher class calls a method of the Chopstick class to free up the Chopsticks for the other Philosophers

```
public boolean pickupChopsticks() {
        if (left_chopstick.inUse(chair_number) == false)
                left_chopstick.claim(chair_number);
        else
                return false;

        if (right_chopstick.inUse(chair_number) == false)
                right_chopstick.claim(chair_number);
        else {
                left_chopstick.release();
                return false;
        }

        return true;
}
```

```
public void eat() {
        changeState(State.EATING);
        try {
                Thread.sleep(eating_time);
        } catch (InterruptedException e) {}

        try {
                left_chopstick.release();
                right_chopstick.release();
        } catch (NullPointerException e) {}

        num_eat++;
}
```

```
public void claim(int chair_number) {
        belongs_to = chair_number;
        chopstick = true;
}

public void release() {
        belongs_to = NO_OWNER;
        chopstick = false;
}
```

```
public int getNumEat() { return num_eat; }

public void setRandomTimes() {
        eating_time = ThreadLocalRandom.current().nextLong(5000);
        thinking_time = ThreadLocalRandom.current().nextLong(eating_time);
}
```

6- Explanation for real world application and how did we apply the problem

In real world application we used Auto Reservation in a hotel management system we assumed the same rules as no guest waits forever to get a room or more than one guest reserve the same room at the same time and both can't occupy it in condition that this hotel has limited numbers of rooms.

We made two classes to define our main contributors in our problem and they were the guest and the room itself we represented a full successful reservation by satisfying two conditions: one is to not exceed your reservation times and the other is to have the room key which means that you reserved and can occupy now.

Same as dinning problem, we change room state if it's owned or not to allow for a guest to occupy it after reservation, and change it another time when the guest leaves the room so no waiting-forever case happens (starvation).

1. Solution pseudocode:

2. Examples of deadlock
3. How did we solve deadlock
4. Examples of starvation
5. How did we solve starvation
6. Explanation for real world application and how did we apply the problem