Club Simulation Report

Enforcing the Rules

The simulation works like a real nightclub. People (patrons) can only go inside if the club isn't full. If the limit is reached, incoming patrons wait outside until someone leaves. As soon as there's space, the next person in line is allowed in.

Once inside, patrons can do different activities like walking around, going to the bar, or dancing.

To make this realistic, the simulation followed these rules:

1. People must use the entrance door to get in and the exit door to leave.
2. Only **one person at a time** can use a door.
3. The number of people inside the club cannot go beyond the set limit.
4. If the door is busy or the club is full, people waiting must be patient.
5. Inside, each person keeps personal space (only one person per spot on the grid).
6. Patrons move one step at a time and in sync with others.
7. The system has to keep running smoothly — no jams or freezes (no deadlocks).

This was all managed using Java's concurrency features like `synchronized`, `wait()`, and `notify()` to make sure the simulation behaved properly.

Challenges and What I Learned

The hardest part was getting synchronization right.

At first, there was a problem where people outside were stuck waiting even though there was still space inside. People already inside also weren't moving together as expected. This happened because the `move()` method wasn't synchronized properly.

The lesson I took away was this: whenever different parts of the program share the same resources (like the doors or movement blocks), rules have to be enforced so only one thread controls that resource at a time. Otherwise, things break down.

How Synchronization Was Done

Different tools were used to keep the program running smoothly:

- **Mutual Exclusion (Mutex):** This means only one person can do an important action at a time, like entering through a door. In Java, this was done using the `synchronized` keyword.
- **Latches:** The simulation didn't start automatically. It waited until I pressed a "start button," which was handled using a latch.
- **AtomicBoolean:** This was used for pausing and resuming the simulation. When "paused," all threads waited. When "resume" was clicked, they all continued normally.

These tools made the system **thread-safe**, so multiple patrons could act at the same time without messing things up.

Keeping Things Moving (Liveness)

“Liveness” just means nobody gets stuck forever. Everyone should eventually get a chance to do something: move, enter, or leave.

- Patrons moved step-by-step inside the club grid, with rules stopping them from entering an already occupied spot.
- At the entrance, patrons waited if the door was in use, then got notified when it became free.

This made sure everyone inside (and waiting outside) eventually got a turn.

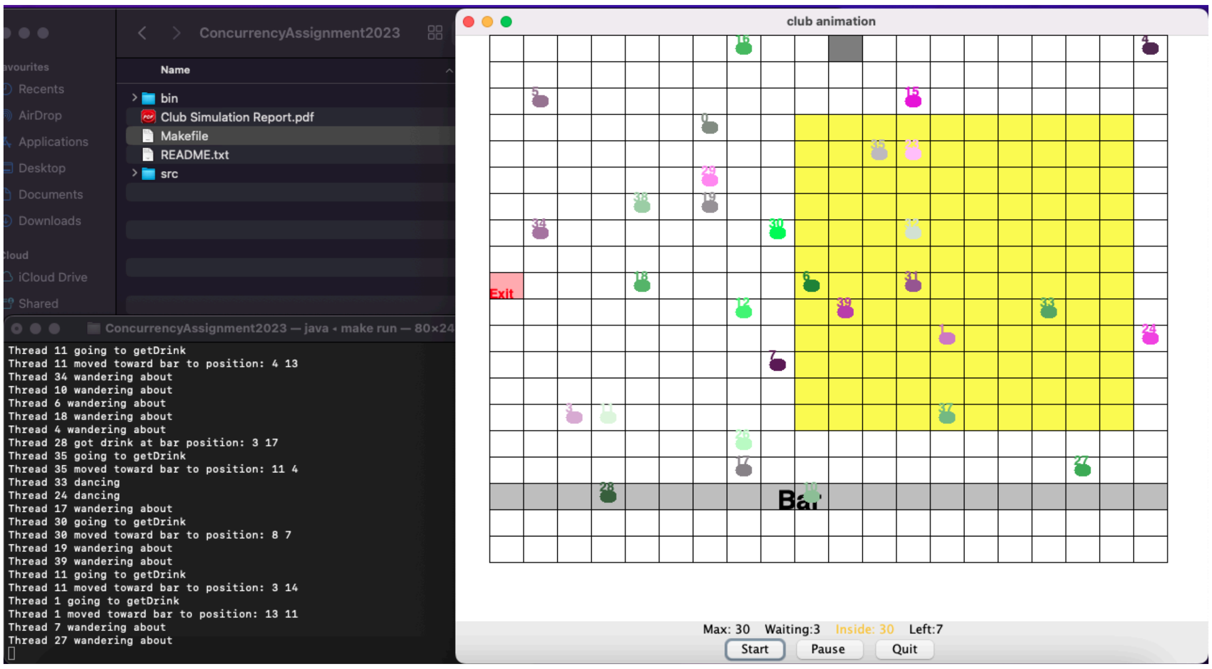
Avoiding Deadlocks

A deadlock is like gridlock traffic — when nobody moves because everyone is waiting on someone else.

In this simulation, a deadlock could happen if two or more people tried to go through the door at the same time. To fix this, both the `enterClub()` and `leaveClub()` methods were synchronized, allowing only one thread at a time. This prevented the system from freezing.

Extra Modern Takeaways

- Instead of just `synchronized`, developers today often use **Locks** (`ReentrantLock`) and **Semaphores** for more control — for example, being able to set timeouts so threads don’t wait forever.
- Newer approaches like **virtual threads** in Java (Project Loom) make handling lots of users easier and more efficient.
- This kind of simulation is similar to real-world challenges in managing websites, ticket systems, or online games where many users interact at once — keeping things orderly and fair is the key.



```
Thread 11 going to getDrink
Thread 11 moved toward bar to position: 4 13
Thread 34 wandering about
Thread 10 wandering about
Thread 6 wandering about
Thread 18 wandering about
Thread 4 wandering about
Thread 28 got drink at bar position: 3 17
Thread 35 going to getDrink
Thread 35 moved toward bar to position: 11 4
Thread 33 dancing
Thread 24 dancing
Thread 17 wandering about
Thread 30 going to getDrink
Thread 30 moved toward bar to position: 8 7
Thread 19 wandering about
Thread 39 wandering about
Thread 11 going to getDrink
Thread 11 moved toward bar to position: 3 14
Thread 1 going to getDrink
Thread 1 moved toward bar to position: 13 11
Thread 7 wandering about
Thread 27 wandering about
```