Explain how you enforced simulation rules:

-The **Start button** initiates the simulation.

-The **Pause button** pauses/resumes the simulation.

To implement the start button to initiate the simulation and the pause button to pause/resume modifications were made in the ClubSimulation and Clubgoer classes.

In the ClubSimulation class modifications were made on the action listener method. This listener responds when the button is pressed. Under this method the '**CountDownLatch'** is initialized and clubgoer threads(patrons) are started. The latch is then released using **latch.countDown()**. All threads must wait until the latch is counted down for the simulation to begin, which is accomplished by the **'CountDownLatch'**.

In the Clubgoer class inside the **startSim()** method modifications were made. This method is uses **latch.await()** responsible for waiting until the latch (initialized in the 'ClubSimulation' class) is counted down before proceeding.

This synchronization helps to achieve a controlled and synchronized start for your simulation, ensuring that all threads begin their work together.

When the 'Pause/Resume' button is clicked, the action listener is triggered. Inside the listener **'pausedSimulation'** is synchronized this shared object is used to control the state of the simulation (paused or resumed) and coordinate between threads.

The synchronization on **'pausedSimulation'** in the ClubSimulation class makes sure that all threads may see changes to its state. Additionally, it avoids race conditions by assuring that only one thread can change the state at once. Race conditions occur when multiple threads attempt to update the state concurrently.

In the Clubgoer class inside the **checkPause()** method modifications were made. It ensures that clubgoer threads are aware of the simulation pause state and act accordingly, waiting when the simulation is paused and resuming when its resumed. This contributes to the coordinated pause and resume behaviour of the simulation.

The **'pausedSimulation'** is also synchronized in the **checkPause()** method. This object is the same shared object used is "Pause/resume" button action listener. This ensures consistent and safe access of the **'pausedSimulation'** state.

-Patrons enter through the entrance door and exit through the exit doors.

-The entrance and exit doors are accessed by **one patron at a time**.

The **get()** method in the GridBlock class is used to determine whether a patron can enter a block (door) or not. If the block is already occupied by another patron, the current patron cannot enter. The **release()** method is used to release a block(door) once a patron exits it, allowing another person to enter. Both these methods use **AtomicInteger isOccupied** to prevent any data races that might occur and are also synchronized to ensure that they are not accessed by multiple threads at once.

In the ClubGrid class the **enterClub()** method the entrance object is synchronized using **synchronized(entrance)**. The method uses a **'while'** loop to check if the entrance is occupied by another patron using the **'entrance.occupied()'** method. The patron waits using **'entrance.wait()'** if the entrance occupied.

The same synchronization method is applied in the **leaveClub()** to notify the threads that the entrance is no longer occupied by using the notifyAll() method.

-The maximum number of patrons inside the club must not exceed a specified limit.
-Patrons must wait if the club limit is reached or the entrance door is occupied.

In the ClubGrid class inside **enterClub()** method the **counter** object is synchronized using **synchronized(counter)**. Within this block there's a **'while'** loop which checks whether the club is over capacity, if it is then it makes threads outside the club wait. Synchronization is used to ensure that only one thread at a time can enter the club when the limit is not exceeded.

-Inside the club, patrons maintain a realistic distance from each other (one per grid block).

When a patron uses the **move()** method to move around the bar, they first check to see if the target GridBlock is occupied. If it's occupied, they stay in their current block; if not, they release it and use synchronized operations to occupy the target block.

__Challenges faced:__

I had problems with implementing the pause button. When I clicked the pause button to pause the patrons, they paused but when I decided to resume the simulation the patrons that were already inside the club froze. The patrons that were waiting to get inside the club continued to get into the club. This happened because I didn't put all the implementation of changing the pausedSimulation state inside the synchronization block.

I also struggled to ensure that when the club is at capacity or maximum the patrons should outside the club. When I was implementing this, I had more issues; when the club is at capacity the patrons would outside but after a while, they would violate this rule and exceed the capacity or when enough patrons have left the club and the club is no longer at capacity the patrons outside would wait forever.

__Lessons learned:__

I found that synchronizing the method and having a synchronized block in the method is a complete overkill. When I tried to run the simulation, it completely froze. Lesson learned from this is that rather than synchronizing the whole method having a synchronized block is much more beneficial.

I also noticed that it is important to use the **notifyAll()** only after the state(flag) has changed because some threads misbehave, or they miss the signal. Therefore, it is important to have a conditional statement before using the **notifyAll()** method. Furthermore, you need to be clear which object you are notifying to ensure consistency.

Synchronizing many objects or method calls causes contention issues, there have been instances were threads take a long time to head to exit and exit the club.

Explain how you ensured liveness and prevented deadlock:

To ensure liveness as mentioned above that contention is caused by synchronizing a method and within that same method having a synchronized block. Therefore, to reduce contention and ensure liveness when I have a synchronized block I unsynchronized that method or when I synchronize the method, I avoided using a synchronized block.