Introduction:

This report will describe the implementation and design of a multi-threaded club simulation in java using concurrent programming. This simulation involves the use of multiple patrons inside the club, utilizing synchronization mechanisms allowed safe interactions between the threads to ensure that the simulation rules are enforced. The main issued that was faced in this simulation was ensuring that patrons actions where coordinated by ensuring no liveness failures, preventing potential deadlocks and and safe access to shared resources.

Enforcing Simulation Rules:

In order to enforce the simulation rules, synchronization was utilized using the synchronized() keyword, this was to control access to shared resources among threads this includes access to club blocks, entrance and exit.

Simulation rules were also enforced by ensuring that the number of patrons in the club did not exceed the capacity that was allowed in the club, synchronization of the enterclub() was applied to control the capacity of the club.
The waitToEnter () method also checked for the capacity and is used for patrons to wait until there is space available before allowing threads to enter.

The simulation rules included and implantation of start and pause mechanisms.The checkPause()and the startSim() method needed synchronization to signal to threads when to start and when to stop. AtomicBoolean was utilized to manage the pause state as the AtomicBoolean flag was used to indicate the simulation is paused or not.

Challenges Faced:
One of the was thread safe interaction between patrons, coordinating them in such a way to prevent deadlocks and race conditions. A race condition is an error where the critical section of a program where the shared memory is accessed is concurrently being used by multiple threads and the dependence of the programs performance is on the order execution of the threads. Race condition occur from lack of synchronization. There are two types: data races, where different threads try to access the same memory location and bad interleavings this exposes a bad intermediate state.

The synchronization issue faced was ensuring that the Clubgoer class's startSim() method and ClubSimulation classes "Start" buttons ActionListener used the same startLock object for synchronization.  issue with ensuring that the threads have a chance to enter the wait() state before the "Start" button is pressed.

Another issue faced was a race condition issue was when the capacity rule needed implemenrtation, threads missed the signal of entering the club or when threads where signalled to enter the club there was a race between the threads to enter the club when the capacity was not at a maximum. This had to be fixed using entrance.notifyAll().
It was important to note that synchronized methods or blocks might cause threads to block and reduce concurrency.

Synchronization Mechanisms and Appropriateness:

Throughout the implementation, we used synchronized blocks and methods to ensure that critical sections of code were accessed by only one thread at a time. For instance, in the ClubGrid class, synchronized blocks were used when entering or leaving the club, ensuring that occupancy levels were properly maintained. Synchronization has made updates in methods such as move(), enterClub(), and leaveClub() in the ClubGrid executed correctly. Synchronization was needed as multiple threads can update the location of a patron simultaneously which could lead to a race condition. This is because multiple threads will try to enter or leave the club. The counter() will also be updated incorrectly causing multiple threads to leave the club at the same tims.

The ClubGrid is crucial coordinating the access to the club blocks and managing the occupancy levels of the club, it manages the actions of patrons entering the club, heading to the bar, and leaving. In the clubgoer class has its own lock (`startLock`) used to start the threads as they wait for the signal.
A shared `pauseFlag` is used to pause and resume the simulation. Threads use `wait()` and `notifyAll()` to pause and resume their actions.The synchronization mechanisms used are appropriately since they ensure that each patron starts at the same time and can be paused and resumed collectively. The mechanisms also allow for safe and coordinated access to shared resources like the club grid.

In the Clubgrid class it enforces rules related to moving, checking for collisions, and occupancy limits. Synchronization is used when the patorns are moving to prevent multiple patrons from moving to the same location simultaneously. The occupancy count and individual grid blocks are shared and accessed using locks. The synchronization mechanisms make sure that only one patron can move to a location at a time.

In the ClubSimulation class the start() and pause() action listener shared the resources from the Clubgoer class to execute and pause the program. This has been done efficiently through synchronization.

Ensuring Liveness and Preventing Deadlock:

Liveness and Deadlock in Multithreaded Programming Liveness refers to the fact that a thread can make progress while deadlock refers to a situation where a thread is stuck in a deadlock state. Deadlock occurs when a thread is waiting for another thread to release resources. When this happens, the entire system becomes unresponsive. To avoid deadlock, locks are implemented in a consistent manner across multiple classes. Wait() and notify() methods are used in your code to coordinate behavior among threads. When a thread needs to wait for a certain condition to be met, the wait() method releases the lock and allows other threads to execute. The notify() method is used to notify all waiting threads when a condition is changed. This way, the thread can wake up and see if the condition it was waiting for is satisfied.

Using AtomicBoolean for Pausing, this is to control whether threads should pause or continue executing. By using an AtomicBoolean, multiple threads can safely access and modify the flag without running into issues like data inconsistency or unexpected behaviour. This helps prevent deadlocks and ensures that threads can respond to the pause command promptly.
Synchronized Blocks: synchronized blocks to protect critical sections where multiple threads might access shared resources concurrently. For example, in the checkPause() method, usex a synchronized block to check the value of pauseFlag. This ensures that only one thread at a time is checking or modifying the flag, preventing potential race conditions and deadlocks.

Conclusion and lesson learned:

In conclusion, what has been deduced is that proper synchronization mechanisms are essential to manage shared resources and prevent race conditions. It is also crucial to consider exactly where to implement synchronization points and where to use wait and notify mechanisms is vital to ensure correct and efficient behaviour between threads allowing controlled coordination.