Concurrency Report

**Student Number:** *P14184295*

**Student Name:** *Sukhwinder Singh*

**Student Course:** *Computer Science*

Choice of language

During the initial phase of deciding how the assignment should be approached, there were many languages to choose from and some were learnt last year such as C and Haskell but they were still not very well understood and the most convincing one and the one that most appealed was Java.

Although Java is being taught in the current year of Computer Science and languages like C help to understand the lower level of computation much better, it was a difficult choice at the beginning on which language should be chosen out of the two. After learning that Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries, it was clear that Java was the ideal language that will be used for this assignment.

Another reason for choosing Java over C was very clear. C is a low-level language, it provides no abstraction compared to Java. Everything in C has to be built by hand with the help of pthreads library but in Java it was a clear and concise approach such as using Java keywords like *synchronized* that helped with synchronous communication and has the same principles as traditional locks but at a very high abstraction level, providing mechanisms like wait and notify and allowing the program to monitor the state of the threads.

Design and Implementation

During the design and implementation phase, a well-drawn out plan was created. Illustrating how the referee will communicate with the player threads whilst maintaining record of the player threads internal states and making sure they are updated once two players have played.

After going over the design many times and looking through the Java API a few times, the referee and player classes were created and they both implemented the *Runnable* Interface that required adding the *run()* method. The *run()* method was used for implementing the behaviour for the threads.

(Oracle, Runnable (Java Platform SE 8), n.d.)

**Referee Thread**

For the referee thread, a *BlockingQueue<E>* interface was thought to be the best approach as the name suggests, for adding players to the queue. The *BlockingQueue<E>* will add players to the queue and support operations that wait for the queue to become empty when storing players. Although *BlockingQueue<E>* is just an interface, the class that implements the *BlockingQueue<E>* interface was a stack based structure called *ArrayBlockingQueue<>* that uses the algorithm First In First Out (FIFO) for storing players in order.

Once that was decided, it was clear how the referee and players will communicate with one another. By using *put()* and *take()* methods that allow asynchronous communication for the producer and consumer to wait and notify. The *BlockingQueue<E>* was added as a global variable allowing both the referee and player constructors to hold a value of the same queue during the running of the main program, allowing them to communicate with one another.

The referee thread had also added a global variable of a custom class called *Calculate* that requires the input of two player threads and calculates their results by getting their shapes and returning an integer value that represents a win, loss or a draw. Another global value was an integer variable that holds the result returned by the *Calculate* class and was used in the if/else statements to check what value was returned by the *Calculate* class and the internal states are then updated accordingly by the referee.

When the referee thread runs, it creates an array of Players of size two. It runs through a loop that iterates twice to add (*take()*) a player from the queue at each index. After retrieving the players, an if/else statement checks if the length of the array is two and if *true* the results are calculated through the use of the *Calculate* class and are then passed to each player correspondingly, updating their internal states. The referee also displays the output of players winning or losing.

The idea behind the array was thought to be a simple and efficient approach due to the nature of knowing that only two players are allowed to play at the same time. The referee then just takes the two players from the queue and allows them to play. With the help of the *Calculate* class, only two players have to be passed as arguments and with the use of an array of size two, it makes it more flexible and after having the results returned, the updating of player’s internal states is made much easier as it requires retrieving players through their indexes which correctly identifies each player and allowing the retrieval of their *getResult()* function which then retrieves *putMVar()* of type Integer function to update the players states.

(Oracle, ArrayBlockingQueue (Java Platform SE 8), n.d.) (Oracle, BlockingQueue (Java Platform SE 8 ), n.d.)

**MVar<E> Class**

The *MVar<E>* class is a custom class of its own and is implemented within the player class for asynchronous communication, just as if it were using the *BlockingQueue<E>* between the referee and the player for asynchronous communication. The same applies for *MVar<E>* but it is used between players. This class like the *BlockingQueue<E>,* provides methods for putting and taking elements of type E. In the Java program, the *MVar<E>* is used for updating the player threads results. The function *putMVar<E>* as previously mentioned and the function *takeMVar<E>* are both methods used with the keyword *synchronized,* allowing monitoring and locking of states when one player is being updated, making the other players wait until notified.

(Oracle, BlockingQueue (Java Platform SE 8 ), n.d.)

**Player Thread**

For the player thread on the other hand, it was a simpler implementation. It required several global variables such as the integer id variable to keep track of each player threads to uniquely identify them. *MVar<E>* was used for retrieving the result from the referee and the players would then update their results. *BlockingQueue<E>* of type Player shared both by the referee and player as previously mentioned to keep track of players being added and removed from the queue. There were also two Shapes variables, for retrieving and holding the Shapes value. Shapes is a custom class made for retrieving a random shape for the user to be used during the simulation of the game. Other variables were all of type integer such as win, lose and draw for keep score of the tally. The variables *turns* and *NTURNS* were used for keeping track of turns and also used for debugging purposes. They helped check if each player has played their turns. Lastly, the *tempResult* variable is used for keeping track of the result and updating tally.

The player threads default constructor requires an id, number of turns and the queue that will be shared by both the referee and the player. The player first checks whether the turns passed through the constructor are greater than 0, if so then the player runs through each iteration of the turns and incrementing their turn count as well as retrieving a random shape through a function call from the Shapes class, that passes the value of the random shape for the player to store. The player then puts (*put())* in the queue shared by the referee and player. The queue by default has built in functionality that allows the players to notify the referee and the player waits and vice versa. Once the referee retrieves the two players and calculates and returns the score back to the players, the players will stop waiting and the results passed to the player will cause the other players to wait as the *MVar<E>* is shared by all players. This mechanism like the queue allows the players to wait between each other for the retrieval of the result, updating one player at a time.

(Oracle, BlockingQueue (Java Platform SE 8 ), n.d.)

**Graceful halting of the Referee Thread**

After testing the player and referee threads, the only thing left to do was to gracefully halt the referee thread once the players had stopped playing. The players would complete their turns and shutdown after the loop within the threads closed down. However the referee still waited for players to be added to the queue when there were no players left to play.

Initially the players and referees started off by running through the *Thread* class. The referee was allocated space on the memory by starting a single thread and the players were allocated space on memory through iterating through a for-loop that started each player thread. Although that approach worked, it was later realised that it was not the most efficient of ways. After looking through the Java API, *ExecutorService’s* were later discovered, that allowed a pool of threads to run that proved to be more efficient.

The use of *ExecutorService* also allowed the use of the *Future<?>* interface that would represent a result of an asynchronous computation, in this case the referee thread. Methods were provided with the *ExecutorService* to check the computation of player threads, to see if they were complete such as *shutdown()* otherwise methods like *awaitTermination()* would wait for the completion of the threads. The result of the computation was then used by the *Future<?>* to terminate the referee thread once all player threads have shutdown.

A simple interruption caused by the *Thread* class method would cause the internal state of the referee to check with the interruption within the referee thread and throw an exception which would cause the graceful shutdown but now through the use of *Future<?>* it was much simpler and safer, as the referee thread will only shutdown after all players have shutdown.

Throughout the process of trying to halt the referee, no problems were encountered besides trying to figure out which method would invoke the interruption and allowing the referee to check for all player threads and whether or not they are alive before gracefully halting the referee thread.

(Oracle, ExecutorService (Java Platform SE 8), n.d.) (Oracle, Thread (Java Platform SE 8), n.d.)

Experience Gained

Overall the assignment presented a lot of challenges and a lot of design considerations were taken before tackling the entire assignment. It helped with the understanding of how a real-life project might take place and how one might approach it. Especially not having learnt concurrency before, it was a fun project and at the same time a project that involved a lot of thinking.

The best aspects of the project was the idea of the referee communicating with the player and vice versa. Being able to come up with an approach for the two to communicate seemed challenging at first but with the help of the Java API it proved much simpler, which was another skill gained from the project, being able to use code provided by third parties. It was interesting because it was not like any other programs done before that provided sequential running through the code but rather two programs or classes talking to each other whilst running simultaneously, just like a client/server.

*ExecutorService* and *BlockingQueue<E>* were the two new discoveries made whilst implementing the program through the help of the Java API. They were never used before and they proved to be very helpful otherwise in the beginning of the implementation, custom classes were made to handle the communication problem that did not help as it gave deadlocks and starvation problems and the help of *Threads* class was used to start and stop the threads that too did not prove efficient as memory was being wasted. Through the use of *ExecutorService* and *BlockingQueue<E>,* a pool of threads were created and a queue that provided built in asynchronous communication to aid the communication between the referee and player threads. They provided with more storage space and a clean connection between the referee and player threads without any problems and worked as intended.

If the program was to be re-written again, it would be written in Java most likely due to the vast amount of Java API’s out there for concurrency support. Some of the custom classes would also be kept such as the *Calculate* class for easy retrieval of the player score and the Shapes class for random retrieval of the shapes but the overall design will contain overridden functionality of interfaces used within the current program such as overriding the *Future<?>* to provide a custom halting mechanism. The classes can also be reduced down to one class instead of having a separate *Player* and *Referee* classes. They could be implemented in one main class as inner classes with the main method for starting the threads. This way it will reduce the size of the program as well as providing majority of the functionality in one file. The queue can also be initiated once instead of having two global declarations within each file. Although it proves efficient in terms of size, there can be a problem in terms of clustering of code all in one file but overall it is just a simulation and it would work.

(Oracle, BlockingQueue (Java Platform SE 8 ), n.d.) (Oracle, ExecutorService (Java Platform SE 8), n.d.)

# References

Oracle. (n.d.). *ArrayBlockingQueue (Java Platform SE 8)*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html [Accessed: 31/03/2016]

Oracle. (n.d.). *BlockingQueue (Java Platform SE 8 )*. Retrieved from Java Platform Standard Ed. 8: view-source:https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html [Accessed: 31/03/2016]

Oracle. (n.d.). *Executors (Java Platform SE 8)*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html [Accessed: 31/03/2016]

Oracle. (n.d.). *ExecutorService (Java Platform SE 8)*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html [Accessed: 31/03/2016]

Oracle. (n.d.). *Runnable (Java Platform SE 8)*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html [Accessed: 31/03/2016]

Oracle. (n.d.). *ScheduledExecutorService (Java Platform SE 8)*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html [Accessed: 31/03/2016]

Oracle. (n.d.). *Thread (Java Platform SE 8)*. Retrieved from Java Platform Standard Ed. 8: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html [Accessed: 31/03/2016]