ASSIGNMENT 2 – CONCURRENCY

INTRODUCTION

The problem presented for this assignment was to finish an incomplete golfing game which utilized multiple threads and shared memory. The aim of this was to enforce concurrency guidelines in the game so as to create a thread safe game that still utilized the full potential of parallelism. A number of classes were given that were missing vital synchronizers and shared memory constraints which would have caused threads to engage in data races and bad interleaving's. Each class was modified to enforce concurrency guidelines by locking classes appropriately and creating constraints on shared memory. All classes will be listed below along with descriptions of the classes.

OVERVIEW OF CLASSES

No classes were added to the golf game as all classes given were adequate to make the fully functional golf game with all requirements including extra credit fulfilled therefore all current classes used will be described below and the modifications will then be broken down below that

Class	*Description*
golfBall	Object class to act as a golf ball with a unique ID for each golf ball made. By making this an object we are allowed to pass the golf ball objects between classes more easily. Since if primitive variables were passed, a local copy would be kept in each thread which makes communication between threads (with golf balls) harder.
DrivingRangeApp	Class to manage the driving range i.e Golfer's entering, Creating Bollie, Start and End times, Creating stash and maintaining Range The DrivingRangeApp will take in all appropriate all command line parameters, then create the appropriate of golfer threads, Create ball stash with assigned size, Create the Range Object and finally create Bollie. The class will then sleep for a random time between 5 and 25 seconds then close the driving range (game over).
Golfer	Thread class to simulate the actions of a golfer in the driving range. Thread acts separately from all other threads thus needs to fulfill concurrency requirements while running. The thread will attempt to fill its bucket by accessing the BallStash object to retrieve a number of golf ball objects. This method is synchronized to prevent data races when different golfer threads are trying to get golf ball objects from stash at the same time. The golfer thread will then (if successfully filled bucket) attempt to swing the balls attained. The golfer must try gain access to a tee (golfing room) of which there are limited amount (extra credit). To allow for limiting use on the rooms, a semaphore

	was used to manage rooms given. Golfers needed to acquire permits to access a room to start swinging. Golfer would then go through each ball in its bucket and “hit” each ball onto the field. Therefore the golfer would pass each ball to the range class to store. This required synchronization as the adding of a golf ball to a list, which is what is done in the range object, is a compound action and thus can have bad intermediate states. Golfer threads will keep checking if Bollie is on the field and if Bollie is found to be on the field then the golfers will be made to wait by the use of a semaphore once again. Golfer thread will golf until he runs out of buckets or if the driving range closes
Bollie	Thread class to simulate the actions a golf ball collector (“ball boy”) Thread will randomly attempt to enter the field to collect balls. This is done through the use of random number generators. When Bollie enters the field, he starts collecting balls. To prevent golfer threads from swinging while Bollie is on the field, a semaphore is controlled to force golfer threads to wait for acquisition and to only be able to carry on when Bollie releases permits on semaphore. Bollie also sets an indicator so that other threads will be able to recognize that he is on the field. After collecting all golf balls from the range object, Bollie will access the BallStash and add all golf balls collected into the stash. The thread then loops continuously entering the field at random times
Range	Class that will act as the field in the driving range. Stores all golf ball objects that are “hit” at the field by golfer threads. This is synchronized to prevent bad interleaving’s amongst threads. Allows access for Bollie to collect all golf ball objects currently stored inside Object. This is also synchronized to prevent golfers from interfering with the collecting from golf balls objects storage. Methods: Synchronized hitBallOntoField(golfBall ball) : Allow balls to hit on field Synchronized collectAllBallsFromField(): Allow Bollie to collect Balls from the field setCartToOff(): change’s Bollie on field flag
BallStash	Class to store and keep track of all golf ball objects in storage at Driving range Allows access to golfer threads to be able to fill their buckets by taking golf ball objects from storage. Synchronized to prevent golfer threads from clashing Allows access to Bollie to refill the storage of golf ball objects from the collection of golf ball objects on the field. Synchronized to prevent data races between golfer threads and Bollie. Will also make golfer threads wait if the BallStash is empty and the golfer threads need golf ball objects. Golfer threads are only “woken up” when Bollie adds to the stash. Methods: Synchronized getBucketBalls(int golferID) : Allow golfers to "take" balls in storage Synchronized addBallsToStash : allow Bollie to add more golf balls to stash/storage Synchronized notifyTheWaitingGolfers() :Wake up golfer threads

MODIFICATIONS TO GIVEN CLASSES

golfBall.java

- No real modification added in the golfBall class as the golfBall class just assigned IDs to newly created objects. No Concurrency requirement as golfBall Objects were created sequentially and not in a multi-threaded environment hence there was no reason to synchronize any methods.
- A toString() method was created to display the unique ID of the object that was being used and passed around by golfers. Used specifically when the BallStash class prints out the remaining balls left in its stash and when the golfer specifies which ball is being "hit"

Range.java

- Change the storage method of golf ball objects on field from golfBall[] to a Linked List. This was done so as to be able to push any golf ball object (no matter ID) to top of the list and be able to pop off golf ball objects. This simulates the randomness when golfers retrieve whatever golf balls were last put in stash as opposed to an array where the lowest golf ball ID will be retrieved.
- An AtomicInteger is used to store the number of balls that are currently on the field to efficiently keep a counter of golf balls
- A hitBallOntoField(golfball ball) method was created to take in a golf ball object and store that object on the field. This thus pushes the ball into the Linked list but this part is synchronized as it is a compound action. It is a compound action as it reads in the array, adds to it, then writes which could cause a bad intermediate state to be exposed to other threads hence the pushing of the ball into the linked list is synchronized. The Atomic Integer does not need to be synchronized as it is already an atomic object which makes sure only one thread increments at a time.

```
synchronized (this){
    ballsOnFieldList.push(ball);
}
//This is an atomic variable so doe
numBallsOnField.getAndIncrement();
```

- A collectAllBallFromField() method was also created which creates a new golfBall[] array and takes out all golf balls from the Range linked list and pushes them into the new golfBall[] array. The new array is then returned. This method is synchronized to prevent golfer threads from changing the golf linked list whilst it is being emptied i.e during the compound action.
- Created setters for the Bollie on field flag to be able to change the cart flag easily

BallStash.java

- Again, changed golfBall[] to a linked list of golfBall objects for similar reasons to Range object storage.
- New golfBall objects are created when constructor is called, these golfBall objects are then pushed to the linked list storage.
- Created getBucketBalls() method which allows golfer threads to retrieve golf ball objects from the stash storage into their local bucket storage. First thing the method does is check if the golf ball stash is empty, if it is then the golfer threads are made to wait by calling this.wait(). This makes any thread that reaches that point to pause its running. The thread will only "wake up" when

```
if (golfBallsList.isEmpty()){
    System.out.println("Golfer
    //Make all threads that re
    this.wait();
```

the Bollie adds more golf balls to the stash and all thread are then notified with this.notifyAll(). If stash is not empty then golfer thread can retrieve as many golf ball objects as his/her bucket permits. This new array of the golfer's retrieved bucket of ball is returned.

- Also added a notifyTheWaitingGolfers() method which is used when the driving range closes. When the driving range closes, all golfers who are still waiting for the stash be filled must leave (according to requirements of Assignment) hence at end of driving range this method is called to force waiting golfer threads to "wake up" and leave the range.

DrivingRangeApp.java

- Added functionality to take in command line parameters to set number of golfers, stash size , balls per bucket, number of tees, number of buckets
- Added random generators to be able to simulate a random opening and closing time of range i.e the driving range class will sleep for a random amount of time between 5 and 25 seconds.
- Randomly generate random times for golfer threads to be created (golfers randomly entering range)
- At the end of class will notify all waiting golfer threads to leave the driving range if they are still waiting for ball stash to be filled.
- Create Range, BallStash then loop through number of golfers and create and start each golfer thread, then create the Bollie thread.

Bollie.java

- Extended Randomly generated range for Bollie randomly entering field from 1 second to 5 second
- Added semaphore to make golfer threads wait until done collecting balls to carry on swinging. Will be explained in concurrency features below
- Set indicator flag to true when start collecting balls
- If no balls could be collected from field then just wait an arbitrary time and try again
- Temporarily stores balls collected from the Range object then passes those balls to the stash to store.

Golfer.java

- Stores all necessary variables to be able to run, i.e semaphores, random generator, cart flag etc
- Will check if golfer thread has used up all of its buckets constantly and if he/she has then they leave the range
- Gets an array of golf ball objects from the ball stash which is now the golf balls in bucket
- Loops through the bucket of golf balls and "hits" each ball onto field, will check if driving range closed while getting golf balls then golfer leaves range
- Stores number of balls in bucket as it can be any number between 0 and max amount of bucket
- Golfer tries to get a tee whenever it wants to swing. Must acquire permit from semaphore and if none are available then it waits

- Checks if Bollie is on field after each swing and if he is then try to acquire a semaphore which is unattainable while Bollie has all permits so golfer waits until Bollie releases all permits to progress
- At end of loop, golfer thread leaves tee
- At the end of run, golfer thread leaves range

CONCURRENCY REQUIREMENTS AND FEATURES

The requirements set out by the task is broken down into:

- Deadlock prevention
- Mutual exclusion
- Livelock prevention
- Bad interleaving prevention
- Stale data prevention
- Random generator generating the same values

Each requirement will be set out with the description of the problem in task and the mitigation strategy used to prevent it.

Requirement	Description	Prevention
Deadlock Prevention	When a Collection of threads are all blocked, wait on one thread to do something that will never happen. Could occur in this game if many locks are acquired when entering an object eg. If Bollie locked both range and stash at the same time and then ran into an exception then golfer threads would not be able to access stash so as to progress	Only ever acquire one lock in a class
Mutual Exclusion	Making sure only one thread can access something at a time. This could cause issues as if we didn't have mutual exclusion then we have the possibility of a read occurring while a write occurs giving the wrong data.	With the use of synchronizers and semaphores, it was enforced that only one thread could access important information at a time. Atomic variables were also used for this issue
Livelock	When threads process something in response to another thread's actions. In this task, we could have an empty while loop running to make golfers wait in response to the Bollie thread indicating that it is on the field. The issue with this is that the golfer threads are left processing (spinning) while they aren't really doing anything, which is wasting computational power	A semaphore was used to prevent this as the Bollie thread would hold on to all permits while on field and force golfers to wait for permit while he was on the field
Bad interleaving prevention	When a thread exposes a bad intermediate state that can be potentially accessed by another thread thus giving out stale information. This issue is raised in this task as there are many compound actions that need to be protected eg adding golf balls to field list	Synchronizers are used again to protect methods that utilize compound actions

Stale data prevention	This is the issue regarding the fact that threads store values in their cache which could potentially out of date. Eg storing an old number of balls on field	This is prevented with synchronizers as they force threads to refresh their cache
Random generator generating the same values	This is the issue that if we create multiple random number generators, some of the instances could spawn the same seed thus create a sequence of the same numbers	Prevented by passing the same number generator to all golfer threads

Validation

The code was run multiple times and analyzed thoroughly to check for data races and bad interleaving's. Also the parameters of the program where changed to check if program could work under any circumstance. This also created more combinations of interactions with threads.

Due to the non-deterministic nature of parallelism with threads, it is difficult to debug through program and be fully sure that the program works correctly. This is due to the fact that there is millions of combinations of thread interaction and the possibility that one of these interactions is a problematic one is challenging. Thus the best way to approach concurrency is by thoroughly analyzing and considering the worst combinations of thread interaction which is what is believed to be done in this task.

Also, a large amount of methods are protected with synchronization which is believed to be an efficient way of thread safety with the repercussion of less parallelism existing between threads. All methods were made to be synchronized then analyzed to check if synchronization was necessary then removed if not.

This is the best validation methods that could be used in a problem with many possible outcomes.

EXTRA CREDIT

The following extension to the assignment was completed

1.4. Extensions

If you finish your assignment with time to spare, you can attempt one of the following for extra credit:

- Allow golfers to arrive at random times, put a limit on the number of buckets per golfer and the number of tees (= number of golfers allowed to practice at a time) and make golfers wait for a tee if none are free.

```
for (int i = 0; i < noGolfers; i++) {
    //Creating Golfer thread with all necessary para
    Golfer newGolfer = new Golfer(stash,sizeBucket,r

    //Will sleep for a random amount of time so golf
    Thread.sleep(golferEntranceTime.nextInt(3000));

    newGolfer.start();  //Runs thread
}
```

Golfers enter the driving range at random times by using a random number generator in the driving range. The driving range is set to sleep for a random amount of time between creating the golfer threads which simulates golfers entering at random times.

The limit on the number of buckets per golfer is enforced by passing the golfer the bucket limit and after every bucket used by the golfer, the golfer would increment a bucket counter. When the bucket counter is larger than or equal to the bucket limit then the golfer leaves the range.

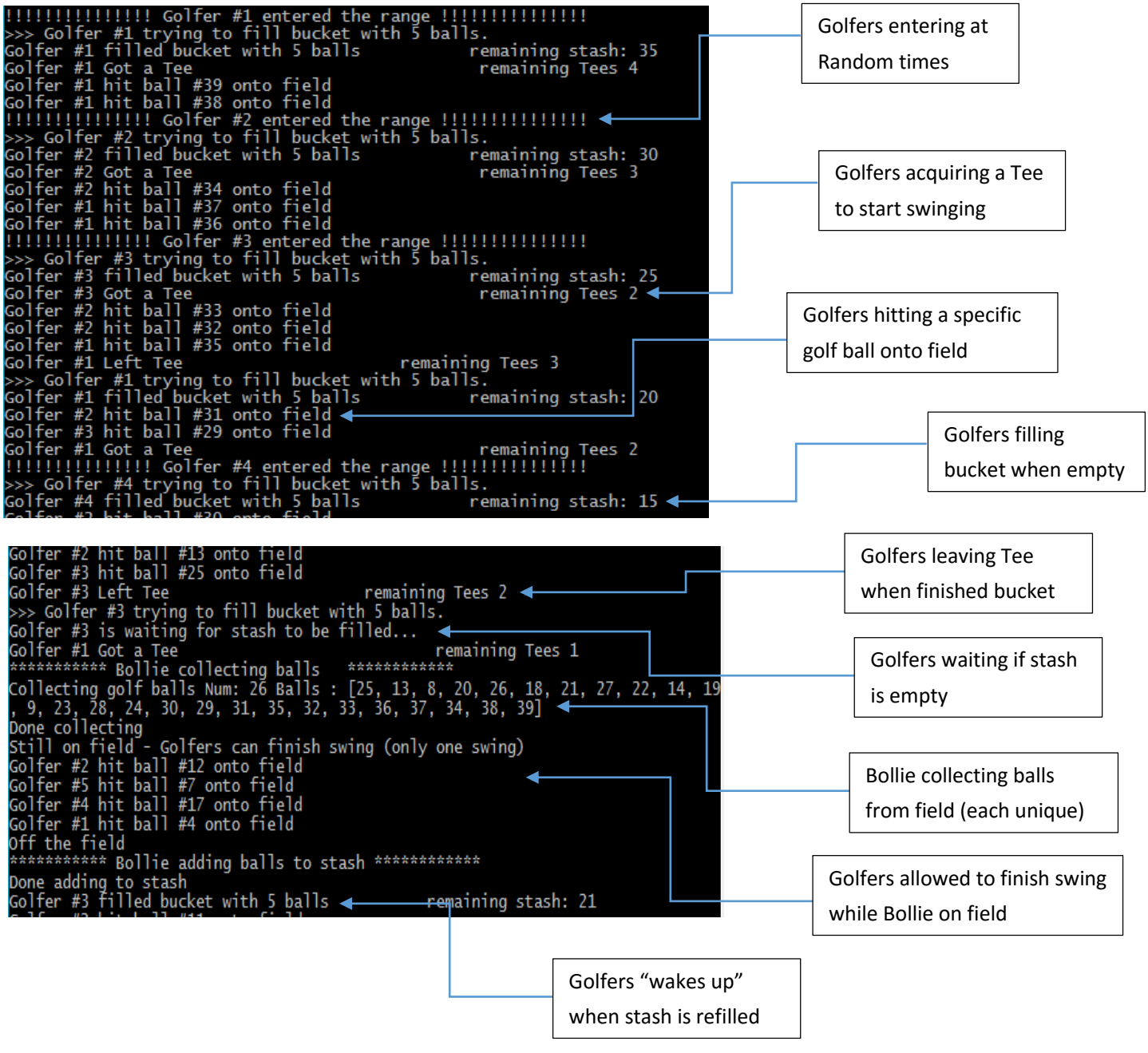
```
//Golfer used all of his/her buckets -
if (bucketsUsed > numBucketsPerGolfer){
    System.out.println("########### Gol
    break;
```

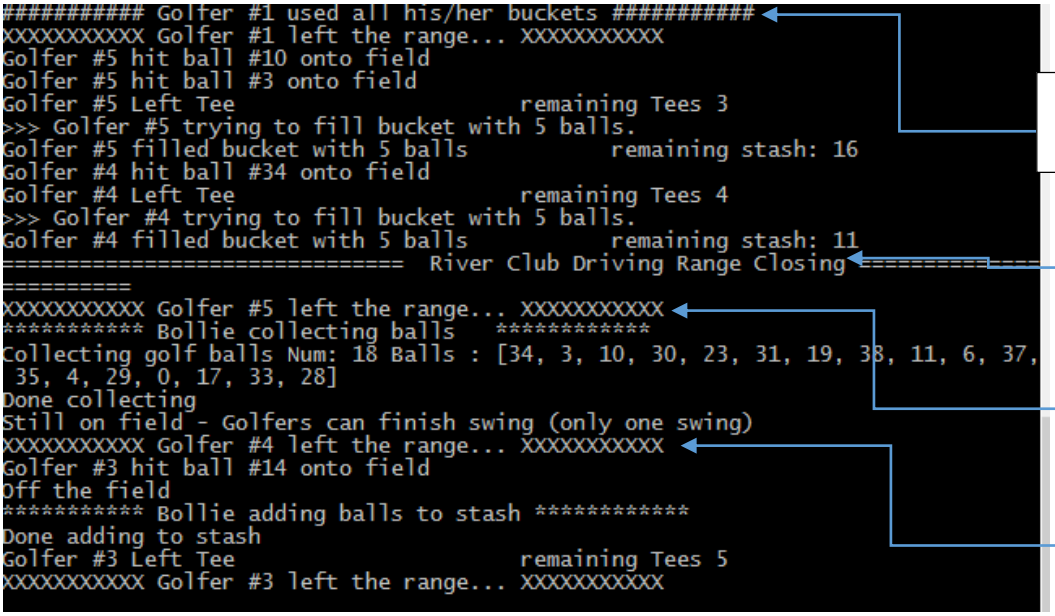
The limited number of tees were enforce with the use of semaphores. A semaphore has an amount of permits that you initially set and any thread can the acquire a permit but once there are none availlable, then the thread must wait until one does become available. Hence the semaphore is used to permit the use of a tee. Each time a golfer thread attempts to start swinging, they must acquire a tee (permit) and when they are done with the bucket they are also done with the tee, hence they leave the tee (release the permit). If there are no tees available then a golfer must wait for a tee to become available (wating for acquire of permit to go through which only happens when a permit becomes available).

```
if (b == 0){
    sharedTees.acquire(); //Gets tee
    System.out.println("Golfer #" + m
}
```

```
if (b == numBallsInBucket -1){
    sharedTees.release(); //Make tee available
    System.out.println("Golfer #" + myID + " L
    bucketsUsed ++; //Used a bucket
```

EXAMPLE OUTPUT





Golfers can use up limited number of buckets

Driving range closing

Golfers leave range when still filling bucket

Golfers leave range when finish with current bucket