Concurrency - CSC2002S - assignment 2

Jason Smythe - SMYJAS002

Introduction

The code of course explains its own (self contained) functionality better than long write-ups or paragraphs can. However, code alone cannot easily explain what alternatives were forgone while coding it, and what preplanning and ideology went into it. That is what I hope to focus on in this write up. I do not want to spend much time spelling existing code out in words.

[[Asside about my code commenting - I did not simply comment what blocks of code do, but also where relevant a short *why*. This is simply to further emphasise and aid the outline of my methodology as described in this paper.]]

To run my code please run the two bash scripts that I have provided, for interactive execution (one for the *vanilla*(no extra features) code, the other for the extra feature code).
Run "./compile-run_Normal" and "./compile-run_EXTRA" in terminal respectively.

Overview of the code and my modifications

golfBall

I made the myId' variable final as there is no need for this to ever change. Enforces safety and peace of mind when using this object in a concurrent program.

ballStash

I set 'sizeStash' and 'sizeBucket' to final and removed any static references to them (I will have to change this for the extension of course where the stash size is no longer constant). This makes sure that these variables are thread safe. Additionally I removed all static methods from this class as I am creating a shared object that can be used instead, so there is no need to have some methods static and others not. [asside: it would be interesting to investigate the effects of making all the methods in this class static, and *sharing* the class instead of the object]

Golfer

I used a Queue to store the golf balls instead of an array. This allowed me to simulate the situation more accurately as the ball could easily be removed from the golfer's bucket and refilled without having to instantiate new arrays each time.

GolferController

I added this class to control the creation and starting of all the Golf threads. This class plays no tangible role in the simulation. I added it simply to make my code more streamlined, object oriented and understandable. This class did, however, become more important in the *extra credit* task.

KeyboardListener

This class is the core of my extension. It controls the GUI and listens for keyboard inputs that change the parameters of the simulation, in a thread safe manner of course.

Extra Credit

I did not make any structural or conceptual changes to my code for the extra credit section. The extra credit section allowed me to test my code in a way that it hadn't been tested before. I did not come across any issues and this makes me confident that it is completely thread safe.

Requirements In Code (conceptualising)

I would like to outline here all the parts of the code that need to be locked, synchronized and atomic. This will give you the idea behind my code and justify my actions.

General Concepts

Closing time

Firstly this task is made somewhat easier due to the fact that the GolfRang only closes once. Now any conditions that depend on the Golf Range being open will be able to check if closed and not just not perform the given action, but also no preceding actions. Thus a return statement can be used to break out of the method with a if(done.get()) return.

Bollie collecting

While Bollie is collecting balls nothing else new can happen (nothing can start), until the Bollie has finished collecting and depositing the golf balls. The assignment spec says: 'The Bollie thread has priority over golfers'. However, here the meaning of the word priority doesn't necessarily have the meaning that if multiple golfers and a Bollie are waiting for a single resource that the Bollie will always get it first. It merely means that if Bollie is currently collecting and/or depositing balls the Golfers will be blocked from swinging (adding to the balls Range) until it is finished.

The approach I went with was to simply make sure both the Golfers swinging and the collecting&depositing of balls by Bollie use the exact same lock. This way the entire time (including the sleep of 1000 ms) that Bollie is performing his task, no Golfers can swing as they cannot access that same lock. I used the Range object as a common lock, as it is the object that is acted upon by both actions, and also easily accessible to both.

Methods of notifying threads of Bollie that require using a shared variable (such as an atomicBoolean, as is done with Closing Time, quickly becomes messy. This is because then you depend on while loops that test whether to wait, and you have to effectively prevent read-act compound actions (interleaving occurring between reading the value of that shared variable and acting).

Printing

Another question one must ask oneself is how closely do you want the printing out of messages that display precisely what the program is doing at that given time.

There are times when it is absolutely Essential that no other thread ‘slips’ in between the printing of the process and the actual process. One such time is at closing time (or after) no golfer can acquire a new bucket, or for that matter *try* and fetch a bucket.

The other time is when it is not essential, is when the print happens before the action is actually carried out, and the action cannot be stopped midway anyway. A key example of this is the golfer's swing, it must only print when the process of starting a swing has started, and not necessarily only when the swing has fully completed. Why I bring this up, is that if any other thread interleaves between the printing and the completion (although completely thread safe), may cause the order in which tasks are **completed** to differ slightly with when they are **started** (and thus the order of the printing). One solution to avoid confusion is to simply is to treat these cases as in the previous paragraph, and only print anything on completion.

What I have used was a tight coupling of action and printing to console. This was achieved by including many of the print statements inside the locks and synchronization of the calling functions. I know that this traded of concurrency, but having print statements that print inaccurately would deem the results useless as you would have no way of telling their validity.

Locks

I attempted to keep it so that at no stage in my program is more than one lock acquired by a single thread. This is to prevent any deadlock possibilities completely. The only exception to this rule is when I set Bollie to keep its lock on the Range while he deposits the balls into the BallStash, acquiring ball stashes lock too (see section *Bollie collecting* above). This however is not an issue as this only ever happens with the Bollie thread, every other thread only ever acquires one lock.

More specifically

Methods and Actions (crucial ones)

The basic actions/interactions that take place (after initialisation of course):

1. Golfer gets his bucket with golf balls
    a. Locking and basic thread safety.
    b. Ensuring stash has enough balls to give to the Golfer.
    c. Must not give out any buckets to waiting Golfers after closing time.
    d. Possible improvements (Theorised)
2. Golfer hits a single golf ball onto the field
    a. Make sure not to swing while Bollie is collecting&depositing.
3. i) Bollie goes onto the field and collects balls
   ii)Bollie deposits his balls into the stash for the players
    a. Both done under the same lock of Range.
    b. Notify all Golfers that may be waiting for buckets of balls.
4. The driving range closes
    a. Prevent ‘accidental’ out of order printing.

NOTE: I have not included actions such as “ball ‘lands’ on the field” because although they can be thought of actions they are integrally connected with another action (in this case #2). So the whole compound action needs to be thread safe.

In Depth Analysis (of the above mentioned Methods and Actions**)**

1. Here the bucket is filled with balls that are in the stash until the bucket is full and ‘given’ to the Golfer. The work is done by the getBucketBalls() method in the BallStash class. The golfer merely receives the result of this method on the common stash.

    a. The first thing that needed to be dealt with is that the the locking. Here various interleaving issues are the things to watch out for. For example, two different golfers both trying to acquire new buckets at the same time. The following ordering could occur:

        [BallStash only has enough balls for one bucket]
        Golfer1 checks Stash has enough balls for his bucket
        Golfer2 checks Stash has enough balls for his bucket
        Golfer2 fills bucket
        Golfer1 attempts to fill bucket, but cannot.:

    This is an example of a read-modify compound action. This needs to be avoided! One way to do this would be to use synchronise on the BallStash object. This is a good solution as it is safe. However, it is unnecessarily to lock on the whole BallStash object as this object's state is only dependent on the stash variable (the other variables are not related to the object's state and are either shared or constant). So I can use a lock on the stash directly using ‘synchronize(stash)’.

This as useful too as in Java's implementation of BlockingQueue 'all queuing methods achieve their effects atomically'.[1]

b. The Stash needs to have enough balls to be filled before attempting to fill a bucket for a Golfer. If it doesn't have enough balls it must not attempt to fill the bucket at all. This is achieved using a stash.wait() contained in a while loop whose condition is to check that the stash has insufficient balls. If there are sufficient balls the bucket is simply filled with the correct amount without worry, as there is no chance of the stash being modified by any other threads due to the lock.

   All the Golfers that were made to wait for the stash are 'woken up' by astash.notifyAll() command that is called as the Bollie finishes depositing balls into this stash.

   I used the notifyAll() method rather than just a notify here for two reasons. Firstly I want to make completely sure there are never any *missed signals* (lets say I only notified 3 Golfers with bucket size 5 when 15 balls were added even though 10 Golfers were waiting). Secondly, because Bollie deposits an undetermined number of balls into the stash, the mechanism to test exactly how many threads to notify would likely be more cumbersome than simply waking them all up.

c. Must not give out any buckets to waiting Golfers after closing time. [[see more on closing time under the *closing time* section]]. This an achieved via locking blocks of relevant code with the 'done' object. eg:

```
synchronized(done) {
        if(done.get()) {
                System.out.println("<<< Golfer #"+ myID + " filled…..");
                return false;
        }
}
```

2. Golfer hits a single golf ball onto the field, this is not an inherently difficult thing to synchronize but there are a few things one needs to look out for. One thing, however that we are spared from looking out for is the closing time condition.

   The basic mechanism for this a *pop* from the bucket queue to the range queue, while synchronized. Here, unlike in number 1 above I am locking on the Range object itself and not the stash, this is because of Bollie, as explained above.

[1] "BlockingQueue (Java Platform SE 7) - Oracle Documentation." 2011. 26 Aug. 2015 <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

   a. See more about how Bollie prevents Golfers from swinging in the section *Bollie collecting* above.

3. i) Bollie goes onto the field and collects balls
   ii)Bollie deposits his balls into the stash for the players
   This seemingly compound action is put into a large locked section including the sleep() that is synchronized on the Range object.

   a. Refer to the section *Bollie collecting* above, for complete explanation of my locking choice.

   b. I notify all the Golfers that are waiting for enough balls to fill a bucket. Details explained in 1D above.

4. The driving range closes. Here I need to make sure threads that are not meant to run don't run. I scattered my code with conditional returns that completely prevent the thread from executing further by leaving the run() function completely. See example from Bollie below:

```
synchronized(done) {
        if (done.get()) return;
        //prevent interleaving on the print. Value of done cannot change inbetween.
        System.out.println("*********** Bollie collecting balls   ************");
}
```

   a. See section *printing* above for more details.

How I validated my system and checked for errors

The most basic way in which I validated my system was by examining the output. When I saw that no statements were misprinted or in *illegal* order I was confident that the placing/structure of my code was in place.[I also was very strict about the printing procedure making sure that the printing reflected exactly what was happening internally, see the *printing* section] I required further examination of the code itself to convince myself of its thread safety.

I knew that my code was deadlock free because I made it an impossibility for multiple threads to hold two locks simultaneously. I also made sure that no intermediate state was ever exposed to any of my threads. I made use of *final* variables where possible, and made use of Atomic variables where necessary.

I tested my simulation with many different values. Here are some of the example values that I used as arguments. [NumberGolfers, NumberGolfBalls, BucketSize]:[30, 10, 1];[1,10,10];[30, 100, 1]; [5,20,5]; [2, 50, 25]

The extra credit part of my assignment was really helpful in verifying code. I was changing parameters that maybe shouldn't have been changed and my code withstood.

Sample output:

```
=======   River Club Driving Range Open  ========
======= Golfers:5 balls: 20 bucketSize:5  ======
>>> Golfer #1 trying to fill bucket with 5 balls.
>>> Golfer #2 trying to fill bucket with 5 balls.
>>> Golfer #3 trying to fill bucket with 5 balls.
>>> Golfer #4 trying to fill bucket with 5 balls.
>>> Golfer #5 trying to fill bucket with 5 balls.
<<< Golfer #1 filled bucket with          5 balls (remaining stash=15)
<<< Golfer #5 filled bucket with          5 balls (remaining stash=10)
<<< Golfer #4 filled bucket with          5 balls (remaining stash=5)
<<< Golfer #3 filled bucket with          5 balls (remaining stash=0)
*********** Bollie collecting balls   ************
*********** Bollie adding balls to stash ************
Golfer #4 hit ball #10 onto field
Golfer #3 hit ball #15 onto field
Golfer #1 hit ball #0 onto field
*********** Bollie collecting balls   ************
*********** Bollie adding balls to stash ************
Golfer #1 hit ball #1 onto field
Golfer #4 hit ball #11 onto field
Golfer #5 hit ball #5 onto field
Golfer #3 hit ball #16 onto field
Golfer #4 hit ball #12 onto field
Golfer #1 hit ball #2 onto field
*********** Bollie collecting balls   ************
*********** Bollie adding balls to stash ************
Golfer #5 hit ball #6 onto field
<<< Golfer #2 filled bucket with          5 balls (remaining stash=4)
Golfer #3 hit ball #17 onto field
Golfer #4 hit ball #13 onto field
*********** Bollie collecting balls   ************
*********** Bollie adding balls to stash ************
Golfer #2 hit ball #10 onto field
Golfer #1 hit ball #3 onto field
Golfer #4 hit ball #14 onto field
>>> Golfer #4 trying to fill bucket with 5 balls.
Golfer #5 hit ball #7 onto field
<<< Golfer #4 filled bucket with          5 balls (remaining stash=2)
Golfer #4 hit ball #5 onto field
Golfer #3 hit ball #18 onto field
*********** Bollie collecting balls   ************
*********** Bollie adding balls to stash ************
Golfer #3 hit ball #19 onto field
>>> Golfer #3 trying to fill bucket with 5 balls.
Golfer #1 hit ball #4 onto field
<<< Golfer #3 filled bucket with          5 balls (remaining stash=3)
>>> Golfer #1 trying to fill bucket with 5 balls.
Golfer #5 hit ball #8 onto field
Golfer #2 hit ball #15 onto field
*********** Bollie collecting balls   ************
*********** Bollie adding balls to stash ************
Golfer #2 hit ball #0 onto field
<<< Golfer #1 filled bucket with          5 balls (remaining stash=2)
Golfer #4 hit ball #16 onto field
Golfer #5 hit ball #9 onto field
>>> Golfer #5 trying to fill bucket with 5 balls.
Golfer #3 hit ball #17 onto field
Golfer #2 hit ball #1 onto field
=======  River Club Driving Range Closing ========
Golfer #2 hit ball #11 onto field
Golfer #1 hit ball #7 onto field
Golfer #4 hit ball #12 onto field
Golfer #3 hit ball #13 onto field
Golfer #1 hit ball #5 onto field
Golfer #4 hit ball #2 onto field
Golfer #3 hit ball #10 onto field
Golfer #4 hit ball #6 onto field
Golfer #1 hit ball #18 onto field
Golfer #3 hit ball #3 onto field
Golfer #3 hit ball #14 onto field
Golfer #1 hit ball #19 onto field
Golfer #1 hit ball #4 onto field
```

Extension

I am not going to talk much about my extension, because all of the concurrent parts are left completely unchanged.

I created a graphical display, created with Swing, that displays the state of the DrivingRange. You can change the state by pressing keys on your keyboard. This prints out a quirky (of my invention) comment in the console and you can immediately see how it affects the rest of the output on the console. My extension allows you to test your code for a much larger variety of values then you would otherwise, all at the touch of a key.

Here are the instructions:
a - adds a golfer;
r - removes a golfer;
i - increase bucket size;
d - decrease bucket size;
t - toggle Bollie's auto collect functionality;
c - make Bollie collect balls;
s - close the range (but be carefull! Can only be pressed once)

So don't just read this, try it, read my quirky comments, be amazed!

(run "./compile-run_EXTRA" in terminal)