CSC2002S - Assignment 2 2015

Anna Borysova
BRYANN004

Introduction

A simulation is a simplified imitation of a real world process over time.[1] It is useful in determining the behaviour of a system without the associated risks, hence it is invaluable in scientific and engineering fields for flight safety testing, disaster preparedness etc. In this project we will be simulating a golf course.

Simulation Rules

- Golfers are allowed to swing at the same time.
- Golfers may not swing whilst Bollie is out on the field.
- Golf balls are never lost (or gained)! They are either in the stash, on the field, or with a golfer.
- The Bollie thread has priority over golfers. i.e., golfers that are currently practicing will be interrupted after their current swing when Bollie heads out

Other Assumptions

- If the field is empty, the Bollie waits until the stash is empty, then carries on heading out at random times
- Golfers may finish their current bucket after the range closes
- Golfers waiting for a bucket when the range closes are denied a bucket

Modifications and Thread Safety Mechanisms

No extra classes were added, only the existing classes were modified to implement some of the functionality and add mechanisms for thread safety.

GolfBall

The `golfBall` class was renamed to `GolfBall` to conform to Java naming conventions[2] but was otherwise left unchanged as all of the mutable functionality of `GolfBall` is used in one thread only. They are all constructed in the main thread, and only accessed in the other threads.

Range

The `doneFlag` was added to the `Range` class, and both `AtomicBoolean` flags were made volatile, in order to ensure that when `doneFlag` gets set to true, the variable gets updated immediately in all the threads. An `ArrayList` was used to store the balls on the field rather than an `ArrayBlockingQueue` in order to handle the waiting and notifications manually. The constructor accepts the `cartFlag` and `doneFlag` parameters.

[1] J. Banks, J. Carson, B. Nelson, D. Nicol (2001). Discrete-Event System Simulation. Prentice Hall. p. 3.
[2] http://www.oracle.com/technetwork/java/codeconventions-135099.html

Four methods were added, `collectAllBallsFromField`, `hitBallOntoField`, `getNumBalls`, and `wake`. Each method used synchronisation over the entire method, as they are all small and it is essential in every one to have the entire method synchronised.

The method `collectAllBallsFromField` is a mutator method, and thus possibly victim to concurrency errors such as data races or race conditions from bad interleaving. The `synchronize` keyword ensures that nothing else can be done on the field while the `Bollie` is collecting. A `while` loop checks if the field has no balls to collect, and makes the `Bollie` thread wait if there are not. However, in the same while loop, the `doneFlag` is checked again, and if it is true the method returns. This functionality is used in the driver class, where the driver class calls the `wake` method of the `Range` object after setting the `doneFlag` to `true`. The `wake` method is a synchronized method which calls `notify` to wake the potentially waiting `Bollie` thread. This makes the `Bollie` check the `doneFlag` and return, because it is true. After the while loop ends, the `cartFlag` is set to true. This prevents the `Golfer` threads from hitting the balls onto the field. It is only set here so as not to prevent the `Golfer` threads from running when there are not even balls on the field. Every ball in the `ballsOnField ArrayList` is added to the (empty) input `ArrayList`, the `ballsOnField ArrayList` is cleared, and the `cartFlag` is set to false.

The `hitBallOntoField(GolfBall ball)` method simply adds the given ball to `ballsOnField` and is synchronised, even though two golfers hitting two balls onto the field at once does not seem problematic, to prevent two `Golfer` threads attempting to add to the same location. It is also necessary for atomicity, because `ArrayList` does not ensure thread safety or atomicity for its methods. The motivation behind using `synchronized` for `getNumBalls` is to prevent stale data and potential bad interleaving due to `size()` not being an atomic action.

BallStash

The `BallStash` class operates similarly to the `Range` class. All the getter and setter methods were left as is for the same reason as in the `GolfBall` class - they are thread safe because they are never used by the multiple threads. The setters are used before any threading begins, and the getter are safe to use as is because the values they are getting are immutable from the point of view of the threads. The `volatile AtomicBoolean, doneFlag` was added and the `ArrayList ballsInStash` which *spoiler alert* stores the `GolfBall` objects which are in the stash.

The constructor sets the `doneFlag` and fills the stash with the required number of `GolfBall` objects. Note here, again, that this is only called once, in the main thread, before any of the `Golfer` or `Bollie` threads are created, and is thus thread safe.

The synchronised method `getBucketBalls` fills the input array of the required bucket size with balls and returns the number of balls remaining in the stash. The exact same while loop trick is used here as with the `Range` class, except the condition is `while(getBallsInStash()<sizeBucket)` and to exit the method it returns `-1`. After the bucket is filled with the `GolfBall` objects and the same `GolfBall` objects are removed from the `ballsInStash ArrayList`, `notifyAll()` is called if there are fewer than a bucket's worth of balls in the stash.

The `getBallsInStash` method is the only getter which is synchronised as it is used within threads. It needs to be atomic, as the number of balls in the stash is variable, and no bad interleavings should be able to occur.

Golfer

The two `AtomicBoolean` flags were made `volatile` for reasons explained earlier. Unlike with the `noBalls` variable in the `GolfBall` class, the `noGolfers` variable needs to have additional thread safety mechanisms. For this reason it was made a `volatile AtomicInteger` initialsed at `1`. The `newGolfID` method is not synchronised because it uses the atomic `getAndIncrement()`. None of the other instance variables need further protection as they are either local to each thread (`myID`) or they are protected through the use of `synchronize` later on. The constructor sustained only minor naming alterations, and the `setBallsPerBucket` and `getBallsPerBucket` are left as is for the same reason as the getters and setters were left as is in the `BallStash` class.

The `run()` method generally keeps to the structure suggested by using the functions suggested in the comments as they were implemented in the other classes with some additions. After the `getBucketBalls` method is called, there is a check for the value of the `doneFlag` (and breaks out of the while loop if it is true) as it might have changed in the time it took to return from the `getBucketBalls` method. For every `GolfBall` in the array, `hitBallOntoField` is called, and at the end of the attempted swing there is a while loop which forces the `Golfer` thread to sleep until `cartFlag` is false.

Bollie

The `Bollie` only has the `doneFlag`, treated as it is in the rest of the classes. The other instance variables and the constructor were not changed. Again, there is another check of the `doneFlag` for the same reasons as before. The rest of the class follows the comment guidelines and adds nothing more.

DrivingRangeApp

The only non-trivial addition to this class is the lines `stash.wake();` and `field.wake();`. These lines ensure that no threads are left waiting indefinitely, and that the `DrivingRangeApp` does terminate. The mechanisms of these methods have been explained earlier.

Other Concurrency Requirements

The concurrency requirements of this simulation are fairly basic, and can all be taken care of using synchronised methods. The mutual exclusion requirement of reading and writing of variables is taken care of by synchronising on the objects in which the variables that need to be protected are stored. Examples of such variables in the code are the `ballsOnField ArrayList` and the `ballsInStash ArrayList`. An explicit requirement of the simulation was that no golfer can hit balls while the bollie is on the field. Instead of using `synchronize` or locks, there is an `AtomicBoolean` flag which is, safely and correctly, shared between threads, and is modified manually. In a couple of places, volatile variables, `AtomicInteger` and `AtomicBoolean` object types were used to make the code a bit lighter. However, speed is not a concern in this simulation, as in comparison to the time it takes for the golfers to hit the ball, and all the other timed delays, the small improvement in speed would be irrelevant. It is much more important to ensure thread safety.

Testing

The simulation was run several times for each of the following input arrays: 3 20 5; 5 50: 10; 20 450 5. In each case, the output was carefully examined for any errors. Particular attention was paid to the IDs of the golfers and the balls: looking for repeated ball ids and disappearing golfers (which would indicate that they are never being notified of the stash being filled). The number of balls that were hit onto the field was compared against the number the bollie collected; them being equal indicated a correctness in the use of the `GolfBall` class.

Extensions

Golfers now arrive at random times, and have a limit on the number of buckets they can use, and get removed if they ask for another bucket. The total number of golfers is still fixed so as to adhere to the initial specifications. Adding this extra functionality required no additional safety measures, however the extension was still tested using the methods described for the classical simulation.