CSYE7215: Homework 2:

Due date:
Pseudo-code and Questions: September, 2018, 05:59 pm
Completed project: September 27, 2018; 05:59 pm

Goal: In this project you will simulate an auction service in which sellers can offer items and bidders can bid on them. In particular, you will implement a class, AuctionServer, in Java, that provides methods supporting all aspects of an auction. Because many sellers and bidders will be interacting at the same time, the program should be thread-safe.

Additionally, the requirement for this assignment is to inject unfairness (or bias) in the behavior of AuctionServer. This means that AuctionServer would give some preferential treatment to some of the buyers (not sellers). You will have to come up with a strategy which you will need to document in your submissions.

Restrictions. The focus of this project is to learn how to write thread-safe code. For this reason, you are not allowed to use any *concurrent collections* in the Java libraries. Specifically, do not use anything contained in java.util.concurrent or the synchronized collections in the Collections class in java.lang.Object. You may use the regular (i.e. non-synchronized) version of collections if you wish, however.

Model: The auction service follows a basic client/server model. There are two types of clients:

- Sellers:
  - Can submit new items to the server
- Bidders:
  - Can request a listing of current items
  - Can check the price of an item
  - Can place a bid on an item
  - Can check the outcome of a bid

These actions will be implemented as methods in the AuctionServer.

Rules: There are a number of restrictions that must be enforced with regard to listing and bidding:

- Listing:
  - Sellers will be limited to having maxSellerItems different active items at any given time
  - The AuctionServer will have a limit, serverCapacity, for the number of total items offered from all Sellers

- Bidding:
  - New bids must *at least match* the opening bid if no one else has bid yet, OR *exceed* the current highest bid if other bids have already been placed on the item
  - Bidders will be limited to having maxBidCount active bids on different current items
  - Once a Bidder holds the current highest bid for an item they will only be allowed to successfully place another bid if another Bidder overtakes them for the current highest bid.

Assumptions: The following assumptions must hold about the AuctionServer:

- **Listing:**
  - All prices will be listed in whole dollars only
  - All items will open with non-negative opening price not exceeding $100
  - If an auction expires with no bids placed, the Seller will not re-list the item and the server receives no profit from it
- **Bidding:**
  - Items can receive any number of bids as long as the auction has not expired
  - Once a bid has been placed it cannot be retracted
  - A single Bidder cannot place more than one bid at a single moment

Bidder/Seller behavior: The Bidder and Seller client classes have been implemented for you. You'll note that they both implement the Client interface that is also provided. Also an Item class is provided. Many clients will run in different threads, and will all access the singleton instance AuctionServer which you will be implementing.

The lifecycle of the test program follows the following three stages (see the UML diagram):

1. Create several clients and execute them on multiple threads
2. Wait for all of the clients to finish
3. Verify the correctness of the system state based on information given below

`Seller` clients will behave in the following way:

- They will hold a list of items to sell (ours generates a large number of items at random to try to sell).
- They are initialized with things such as a unique name and how many cycles it should execute and the longest it should sleep between attempts to list an item.
- When the thread is run it enters a loop and iterates the specified number of cycles, where in each cycle it will
  - randomly pick an item and try to submit it to the server, then
  - if the submission is accepted, that item is then removed from its own list of things it wants to sell, and then
  - after each submission attempt to the server, the Seller sleeps for some random amount of time

`Bidder` clients will behave in the following way:

- They are initialized with things such as a unique name and how much cash it has available to spend, how many cycles it should TRY to execute (if it's out of cash, why bother continuing) and the longest it should sleep between attempts to buy and check items.
- When the thread is run it enters a loop and iterates the specified number of cycles (though there is an escape clause for if it runs out of cash) where in each cycle it will
  - try to buy things and
  - check up on all of its active bids
- To try to buy things within each cycle, it will
  - retrieve a list of items available for sale, then
  - randomly picks an item that it can afford going on the worst---case assumption that it wins all outstanding bids, and
  - adds $1 to the highest bidding price and makes the bid
- To check up on all of its active bids within each cycle, it will
  - check the status of the bid and

- if the bid was successful (i.e., it won the auction) it will deduct the price of that item from its cash reserve.

Note that these classes do not enforce the restrictions listed earlier in the "**Rules**" section. This is your job to implement in the AuctionServer.

Code to implement: For this project you are only required to modify one file, AuctionServer.java. The methods to implement will start with the comment // TODO: IMPLEMENT CODE HERE. A description of each of the methods follows below. Refer to the comments for each method for further notes on what these methods should do.

- submitItem(...)
  A Seller calls this method to submit an item to be listed by the AuctionServer. A Seller uses sellerName and itemName to identify itself and the Item that is submitted. The unit for the bidding duration is in milliseconds. If the Item can be successfully placed, this method returns a unique positive listing ID generated by the AuctionServer. If the Item cannot be placed, for instance, the Seller has already used up its quota or the server has reached maxSellerItems items listed, this method returns -1.
- getItems()
  A Bidder calls this method to retrieve a copy of the list of Items currently listed as active. Each Item object in the list provides access to its name and its initial minimum bidding price. (It is important to remember the current bid price of the item may have changed from its initial value and the actual bid price can be retrieved by calling the method itemPrice(), see below.)
- itemPrice(...) A Bidder checks the current bid/opening price for an Item by supplying the unique listing ID of that Item. The value returned by this method is the highest bid made so far, or the minimum bid value supplied by the seller if nobody made a bid on the item. If there is no Item with the supplied listing ID the method indicates an error by returning a value of -1.
- itemUnbid(...) A Bidder checks whether an Item has not yet been bid upon by supplying the unique listing ID of that Item. This method returns true if no bid has been placed and false otherwise. If there is no Item with the supplied listing ID the method returns a value of true since it is true that the non-existing Item has not yet been successfully bid upon.
- submitBid(...) A Bidder calls this method to submit a bid for a listed Item. This method returns true if the bid is successfully submitted and false if the submission request is rejected. There are several situations when a bid submission request can be rejected. If a Bidder already has bid on too many items, the Bidder is not allowed to place bids on new items. If a Bidder already has a bid on an item, the Bidder is not allowed to place a new bid on the same item until another Bidder has placed a higher bid. The bid can also be rejected if the item is no longer for sale, or if the Bidder has been added to the blacklist due to violation of a Rule, or if the listing ID corresponds to none of the items submitted by the sellers.
- checkBidStatus(...) A Bidder calls this method to poll the AuctionServer to check the status of a bid the Bidder may have on an Item. There are three possible status results:
  1. **SUCCESS** (return value 1): If this item's bidding duration has passed and the Bidder has the highest bid.
  2. **OPEN** (return value 2): If this item is still receiving bids.
  3. **FAILED** (return value 3): If bidding is over and this Bidder did not win, or if the listing ID doesn't correspond to any Item submitted by the sellers. As part of its job, if the item being checked is no longer open and still appears on the list of items currently listed as active, this method will remove it from that list and update the appropriate locations to reflect that it is no longer being bid upon and also update the appropriate fields if it was successfully sold to anyone.

Fields: The limits mentioned earlier in the description are all stored as public constant integers. Please note that we can change these values as part of our grading but we will not change the names of the constants. They will be:

- an integer constant called maxBidCount
- an integer constant called maxSellerItems
- an integer constant called serverCapacity

You will also be required to keep track of two statistics for the AuctionServer's sales:

- a mutable integer called soldItemsCount (the total number of items per 100 milliseconds that have been sold so far; this is like a selling rate)
- a mutable integer called revenue (the revenue generated so far)

These values are required to stay up-to-date and also be **thread-safe**.

Testing: It is important to check your program's performance thoroughly. The class Simulation.java has been provided for you to perform just the basic testing. By default, the main method simply creates Sellers and Bidders and runs them on the AuctionServer. You may modify the Simulation.java class as you see fit. Sharing of tests for this project is **encouraged**. After all, even though your program may pass your tests, it may fail on someone else's test.

Since this assignment requires that your ActionServer is biased, you need to provide some code in the Simulation class that will analyze this aspect, i.e., it will show that AuctionServer behaves according to the bias policy that you declare and implement. In other words, your Simulation should expose the advantage received by the buyer you chose to give preferential treatment to. For this purpose, you need to propose and implement the statistic that will show how much that seller benefited due to the server's bias as opposed to other sellers. Note that the bias doed not need to be caught in any single run, but it should come out in a number of runs.

Submission: Use your Last Name (jut one word; you can simplify your last name if you prefer) for the package name. Submit a .zip file containing your project files to Assignments in Blackboard.

1st Deliverable: Pseudo-code, invariants, pre- and post-conditions for the AuctionServer and Simulation classes.
2nd Deliverable (Final): Code for AuctionServer and Simulation.