Project 1 – Auction House

Due: 09/11/2019 11:59:59 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 must be thread-safe.

Getting started. The project skeleton can be downloaded as a zip file from the course website. The directions given in Project 0 may be used to unpack the zip file to create a project named "p1" in Eclipse (similar approaches apply for other IDEs as well).

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.

Specification in Detail. The auction service follows a basic client/server model. There are two types of clients: Sellers and Bidder. A Seller will never be a Bidder, and vice-versa.

1. Sellers
    a. can submit new `Items` to the server

2. Bidders
    a. Can request a listing of current `Items`
    b. Can check the price of an `Item`
    c. Can place a bid on an `Item`
    d. Can check the outcome of a bid
    e. Can pay for `Items` they have won

These actions will be implemented as methods in the `AuctionServer`.

There are a number of restrictions which must be enforced with regards to listing and bidding.

1. Listing
    a. Sellers will be limited to having `maxSellerItems` different active `Items` at any given time.
    b. The `AuctionServer` will have a limit, `serverCapacity`, for the number of total active `Items` offered from all Sellers.

2. Bidding

a. New bids must at least match (greater than or equal to, >=) the opening bid if no one else has bid yet, or **exceed** (strictly greater than, >) the current highest bid if other bids have already been placed on the item.
b. Bidders will be limited to having `maxBidCount` active bids on different current items.
c. Once a Bidder holds the current highest bid for an item they will only be allowed to successfully place another bid for that item if another Bidder overtakes them for the current highest bid.
d. Once a Bidder has won an auction, they must pay at least the amount of their bid in order to receive the item.

One situation that can occur during an auction is that a Bidder could place several bids which sum to an amount that is higher than the Bidder can afford, thinking that they are bound to not win all of the items which they have bid upon. If the Bidder is in fact the high bidder for all these items, then they would not be able to pay for all these items, thereby shortchanging the auction and other Bidders who might have won items otherwise. To discourage this behavior, if a Bidder is unable to pay the `AuctionServer` at least the amount of their bid for an item which they have won, that Bidder will have all their active bids[1] cancelled and will be permanently banned from placing bids on the `AuctionServer` in the future. When a bid is cancelled in this manner, it is as if the item was never bid upon (i.e. someone may restart bidding on the item at the opening price, provided the bidding is still open, although the remaining duration for the item is not reset).

You may make the following assumptions when implementing the `AuctionServer`:

1. Listing
    a. All prices will be listed in whole dollars only.
    b. All items will open with prices greater-than or equal-to zero.
    c. All items will open with bidding durations greater-than zero.
    d. If an auction expires with no bids placed, or if the winning bidder is unable to pay for the item, the Seller will not re-list the item, and the server receives no profit from it.

2. Bidding
    a. `Items` can receive any number of bids as long as the auction has not expired.
    b. Once a bid has been placed it cannot be retracted by the Bidder.
    c. A single Bidder cannot place more than one bid at a single moment.

The `Seller` class and two different `Bidder` client classes have been implemented for you. You will note that they all implement the `Client` interface that is provided. We have also given an `Item` class. Many clients will run in different threads, and will all access the singleton instance `AuctionServer` that you will be implementing.

The lifecycle of the test program follows these three stages.

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 behave in the following way:

1. They will hold a list of items to sell (ours generates a large number of items at random to try to sell).
2. 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.
3. When the thread is run it enters a loop and iterates the specified number of cycles, where in each cycle it will:
    a. randomly pick an item and try to submit it to the server, then
    b. if the submission is accepted, that item is then removed from its own list of things it wants to sell, and then
    c. after each submission attempt to the server, the Seller sleeps for some random amount of time.

Bidder clients behave in the following way:

1. They are initialized with things such as a unique name and how much cash they have available to spend, how many cycles they should try to execute, and the longest they should sleep between attempts to buy and check items.
2. When the thread is run it enters a loop and iterates the specified number of cycles (though there is an escape clause in case it runs out of cash), where in each cycle it will:
    a. try to buy as many things as it can, and
    b. check on all of its active bids.
3. To try to buy things within each cycle, it will:
    a. retrieve a list of items available for sale, then
    b. randomly pick an item that it can afford (in the case of the `ConservativeBidder`, going on the worst-case assumption that it wins all outstanding bids), and
    c. add $1 to the highest bidding price and makes the bid.
4. To check up on all of its active bids within each cycle, it will
    a. check the status of the bid, and
    b. if the bid was successful (i.e.: it won the auction) it will deduct the price of that item from its cash reserve and pay for it using the `payForItem` method. Note that the provided classes do not enforce the restrictions listed above. 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, this method returns -1.

Notes:

1. You do not need to check if the `Item` has the same name as another `Item`
2. You ***do*** need to make sure that no two items have the same listingID
3. This method should return -1 if:
    a. the Seller has already used up its quota; or,
    b. the server has reached `serverCapacity` number of `Items` listed
4. This method should return some unique number >= 0 otherwise

~ ~

`getItems()`
A Bidder calls this method to retrieve a **copy** of a list of the `Items` that are currently available for bidding.

Note: make sure this returns a copy of the list, not the list itself!

~ ~

`itemPrice()`
A Bidder checks the current bid/opening price for an `Item` by supplying the unique listingID 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 the item does not have a valid bid on it.

Notes:

1. Once an item has been successfully paid for, this method should continue to return the highest bid, even if the buyer paid more than that amount for the item, and even if the buyer is subsequently blacklisted.
2. If this item is still up for bidding, but the current highest bidder has been blacklisted after failing to pay for another item, this method should return the original listing price.
3. If there is no `Item` with the supplied listingID, then the method indicates an error by returning a value of -1.

~ ~

`itemUnbid()`
A Bidder checks whether or not an `Item` currently has a bid on it by supplying the unique listingID of that Item. This method returns **true** if there is no bid 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.

A bid can be rejected for the following reasons:

1. The Bidder has active bids on too many `Items`
2. The Bidder already has the highest bid on this `Item`
3. The bid provided is less than or equal to the current highest bid. **Exception: if the `Item` has not yet been bid upon, a Bidder can successfully bid >= the minimum amount provided by the Seller.**
4. The `Item` is no longer for sale
5. the listingID corresponds to none of the `Items` submitted by the sellers
6. the Bidder has been added to the blacklist

~ ~

`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: This item's bidding duration has passed and the Bidder has the highest bid.
2. OPEN: This item is still receiving bids.
3. FAILED: The bidding is over and this Bidder did not win, or the listingID doesn't correspond to any `Item` submitted by the Sellers.

In addition to returning the status of a bid, this method should perform the following cleanup actions in the case where bidding on the item being checked is no longer open:

1. Remove the item from the list of items up for bidding
2. Decrement the number of items being sold by that item's Seller
3. If the item has a highest bidder, increment the uncollectedRevenue field by the highest bid amount. **It is important to only do this once per item – multiple Bidders may call this method!**

~ ~

`payForItem()`

A Bidder calls this method to pay the `AuctionServer` for an `Item` which they have won. If the Bidder is the winner of the `Item`, and the `amount` paid is greater than or equal to the price of the `Item`, this method should return the name of the `Item` and update the appropriate fields based on the `amount` paid to indicate the `Item` was sold (note that this includes decrementing the `uncollectedRevenue` field by the price of the `Item`, since the revenue has now been collected). If the Bidder won the item, but the `amount` paid is insufficient, the server does not receive any profit and this method should cancel all the Bidder's active bids, add them to the blacklist, and throw an `InsufficientFundsException`. (Note that the revenue for this item remains uncollected forever in this case.) If the `listingID` does not correspond to any of the `Items` submitted by the Sellers, the bidding on the `Item` is not yet closed, the `Item` has already been paid for, or if the Bidder was not the winner of the `Item`, this method should return **null** and the money being paid by the bidder does not change hands.

Notes:

1. To successfully pay, the following conditions must be met:
    a. Bidding on the `Item` must be closed
    b. The `Item` must have been bid upon
    c. The Bidder must have the same name as the current highest bidder for the `Item`
    d. The amount paid must be greater than or equal to the current highest bid

2. After payment, you must correctly update the following:
    a. Decrement uncollectedRevenue by the current highest bid
    b. Increment the number of items sold by the server
    c. Increment the server's revenue by the amount provided
    d. Return the name of the `Item`

3. If the following conditions hold, you must instead blacklist the Bidder:
    a. An `Item's` bidding duration has expired
    b. The `Item` has a highest bid
    c. The winning Bidder attempted to pay for the `Item` but did not have enough money

4. When blacklisting a Bidder, you must do the following:
    a. Add the bidder to the blacklist
    b. For any `Item` still OPEN, on which the Bidder is the current highest bidder:
        i. Reset the highest bid to the minimum bid supplied by the Seller
        ii. Remove the current highest bidder
    c. Do not update revenue, uncollectedRevenue, or itemsSold
    d. Return **null**