# Project 1 – Auction House

## Due: 09/19/2018 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
  - 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
  - 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.

- 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 active 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 for that item if another Bidder overtakes them for the current highest bid.
  - 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 s/he is bound to not win all of the items which s/he has bid upon. If the bidder is in fact the high bidder for all these items, then s/he 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`.

- Listing
  - All prices will be listed in whole dollars only.
  - All items will open with non-negative opening prices.
  - Sellers will not list items with opening prices exceeding $100.
  - 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.
- 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 by the bidder.
  - A single Bidder cannot place more than one bid at a single moment.

The `Seller`, 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.

[1] For the purposes of this project, a bid is *active* if the item on which the bid has been placed would be included in the list returned by the `getItems()` method described later.

Seller clients 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 behave in the following way.

- 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.
- 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
    - try to buy as many things as it can, and
    - check 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 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
    - add $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 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 (for instance, if the Seller has already used up its quota or the server has reached `serverCapacity` items listed), this method returns -1.

- `getItems()`
  A Bidder calls this method to retrieve a copy of the items that are currently available for bidding. 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 the item does not have a valid bid on it. (Note that 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.) If there is no `Item` with the supplied listing ID, 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 listing ID 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. 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, if the listing ID corresponds to none of the items submitted by the sellers, or if the Bidder has been added to the blacklist due to failing to pay for an item they had previously won.
- `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): This item's bidding duration has passed and the Bidder has the highest bid.
  2. OPEN (return value 2): This item is still receiving bids.
  3. FAILED (return value 3): The bidding is over and this Bidder did not win, or the listing ID doesn't correspond to any `Item` submitted by the sellers.

  In addition to returning the status of a bid, this method should update the list of active items in the case where the item being checked is no longer open. In this situation the method should remove the item from that list and update the appropriate locations to

reflect that it is no longer being bid upon. If the item was sold to someone, this method should also update the `uncollectedRevenue` field appropriately, to indicate that the server is due to receive payment for the item in the future.

- `payForItem()`
  A Bidder calls this method to pay the `AuctionServe`r for an item which he or she has 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.

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, which are as follows.

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

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

- a mutable integer called `soldItemsCount` (the total number of items that have been sold and paid for so far)
- a mutable integer called `revenue` (the sum of the amount of money that buyers have paid the server for items)
- a mutable integer called `uncollectedRevenue` (the sum of the highest valid bids on items which have been sold to a bidder, but have not yet been purchased successfully)

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

**Testing.** We will provide only basic sanity tests for this project, so it is important to check your program's performance thoroughly. The secret tests we will use for grading will examine both single-threaded and multi-threaded cases for correctness, and may use clients other than those provided. The class `Simulation.java` has been provided for you to perform testing. By default, the main method simply creates sellers and bidders and runs them on the `AuctionServer`. You may modify this class as you see fit (it will not be used for grading). Sharing of tests for this project is encouraged.

**Submission.** Submit a .zip file containing your project files to the CS submit (submit.cs.umd.edu). You may use the same approach outlined in Project 0 to create this from inside Eclipse.