CMSC 433 Project 1: Auction House

Due February 27th (Friday), 11:59:59pm

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**. You may find it useful to explore atomicity and concurrent access to collections.

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 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 if another Bidder overtakes them for the current highest bid

Givens: 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, 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. We have also provided an `Item` class. 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:

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 what we know it should be.

`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
    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 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
    a) try to buy as many things as it can and
    b) check up on all of its active bids
- To try to buy things within each cycle, it will
    a) retrieve a list of items available for sale, then
    b) randomly picks an item that it can afford going on the worst-case assumption that it wins all outstanding bids, and
    c) 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
    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

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(...)`
  Note: A `Seller` may call 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()`
  Note: A `Bidder` may call this method to retrieve a copy of the listed `Items`. 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(...)`
  Note: A `Bidder` can check 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 (it never existed) the method indicates an error by returning a value of -1.
- `itemUnbid(...)`
  Note: A `Bidder` can check 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(...)`
  Note: 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 listing ID corresponds to none of the items submitted by the sellers.
- `checkBidStatus(...)`
- **Note:** 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. This should occur on the first call of this function regardless of the `Bidder` checking.

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 that have been sold so far)
- a mutable integer called `revenue` (the total sum of the highest valid bids on the items sold so far)

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. 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 Submit Server.