package p1;

/\*\*

\* @author Vishakha Sawant

\*/

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import p1.Item;

public class AuctionServer

{

/\*\*

\* Singleton: the following code makes the server a Singleton. You should

\* not edit the code in the following noted section.

\*

\* For test purposes, we made the constructor protected.

\*/

/\* Singleton: Begin code that you SHOULD NOT CHANGE! \*/

protected AuctionServer()

{

}

private static AuctionServer instance = new AuctionServer();

public static AuctionServer getInstance()

{

return instance;

}

/\* Singleton: End code that you SHOULD NOT CHANGE! \*/

/\* Statistic variables and server constants: Begin code you should likely leave alone. \*/

/\*\*

\* Server statistic variables and access methods:

\*/

private int soldItemsCount = 0;

private int revenue = 0;

public int soldItemsCount()

{

return this.soldItemsCount;

}

public int revenue()

{

return this.revenue;

}

/\*\*

\* Server restriction constants:

\*/

public static final int maxBidCount = 10; // The maximum number of bids at any given time for a buyer.

public static final int maxSellerItems = 20; // The maximum number of items that a seller can submit at any given time.

public static final int serverCapacity = 80; // The maximum number of active items at a given time.

private final Object readlock = new Object();

private final Object writelock = new Object();

/\* Statistic variables and server constants: End code you should likely leave alone. \*/

/\*\*

\* Some variables we think will be of potential use as you implement the server...

\*/

// List of items currently up for bidding (will eventually remove things that have expired).

private List<Item> itemsUpForBidding = new ArrayList<Item>();

// The last value used as a listing ID. We'll assume the first thing added gets a listing ID of 0.

private int lastListingID = -1;

// List of item IDs and actual items. This is a running list with everything ever added to the auction.

private HashMap<Integer, Item> itemsAndIDs = new HashMap<Integer, Item>();

// List of itemIDs and the highest bid for each item. This is a running list with everything ever added to the auction.

private HashMap<Integer, Integer> highestBids = new HashMap<Integer, Integer>();

// List of itemIDs and the person who made the highest bid for each item. This is a running list with everything ever bid upon.

private HashMap<Integer, String> highestBidders = new HashMap<Integer, String>();

// List of sellers and how many items they have currently up for bidding.

private HashMap<String, Integer> itemsPerSeller = new HashMap<String, Integer>();

// List of buyers and how many items on which they are currently bidding.

private HashMap<String, Integer> itemsPerBuyer = new HashMap<String, Integer>();

//List of users to be disqualified

private HashMap<String, Integer> userQualification=new HashMap<String, Integer>();

// Object used for instance synchronization if you need to do it at some point

// since as a good practice we don't use synchronized (this) if we are doing internal

// synchronization.

//

// private Object instanceLock = new Object();

/\*

\* The code from this point forward can and should be changed to correctly and safely

\* implement the methods as needed to create a working multi-threaded server for the

\* system. If you need to add Object instances here to use for locking, place a comment

\* with them saying what they represent. Note that if they just represent one structure

\* then you should probably be using that structure's intrinsic lock.

\*/

/\*\*

\* Attempt to submit an <code>Item</code> to the auction

\* @param sellerName Name of the <code>Seller</code>

\* @param itemName Name of the <code>Item</code>

\* @param lowestBiddingPrice Opening price

\* @param biddingDurationMs Bidding duration in milliseconds

\* @return A positive, unique listing ID if the <code>Item</code> listed successfully, otherwise -1

\*/

synchronized public int submitItem(String sellerName, String itemName, int lowestBiddingPrice, int biddingDurationMs) throws ArithmeticException

{

//System.out.println(lowestBiddingPrice);

// TODO: IMPLEMENT CODE HERE

// Some reminders:

// Make sure there's room in the auction site.

Item item;

if(lowestBiddingPrice<0)

return -1;

if(lowestBiddingPrice>99)

{

//disqualify seller if he tried to put itemprice greater than 99 three times or more than three times

if(this.userQualification.get(sellerName)>=3){

return -1;

}

else

{

if(this.userQualification.get(sellerName)==0)

{

this.userQualification.put(sellerName,1);

}

else

{

int count=this.userQualification.get(sellerName);

this.userQualification.put(sellerName,count++);

}

return -1;

}

}

if(this.itemsUpForBidding.size()==serverCapacity || this.itemsAndIDs.size() == serverCapacity)

return -1;

// If the seller is a new one, add them to the list of sellers.

Integer count= this.itemsPerSeller.get(sellerName);

synchronized(writelock)

{

if(count==null)

{

count=0;

this.itemsPerSeller.put(sellerName,count);

}

// If the seller has too many items up for bidding, don't let them add this one.

if(count>=maxSellerItems)

return -1;

// Don't forget to increment the number of things the seller has currently listed.

//Acquire lock on lastListingId

item=new Item(sellerName, itemName,++this.lastListingID, lowestBiddingPrice, biddingDurationMs);

if(!this.itemsAndIDs.containsKey(item.listingID()))

{

this.itemsAndIDs.put(this.lastListingID,item);

this.itemsPerSeller.put(sellerName,itemsPerSeller.get(sellerName)+1);

this.itemsUpForBidding.add(item);

//this.highestBids.put(this.lastListingID, lowestBiddingPrice);

//System.out.println("Item submitted:"+ item.lowestBiddingPrice());

}

}

//Release Lock

return item.listingID();

}

/\*\*

\* Get all <code>Items</code> active in the auction

\* @return A copy of the <code>List</code> of <code>Items</code>

\*/

public synchronized List<Item> getItems()

{

// TODO: IMPLEMENT CODE HERE

// Some reminders:

// Don't forget that whatever you return is now outside of your control.

List<Item> newCopy=new ArrayList<Item>();

for(Item i:this.itemsUpForBidding)

{

if(i!=null)

newCopy.add(i);

}

return newCopy;

}

/\*\*

\* Attempt to submit a bid for an <code>Item</code>

\* @param bidderName Name of the <code>Bidder</code>

\* @param listingID Unique ID of the <code>Item</code>

\* @param biddingAmount Total amount to bid

\* @return True if successfully bid, false otherwise

\*/

synchronized public boolean submitBid(String bidderName, int listingID, int biddingAmount) throws ArithmeticException, IllegalArgumentException

{

// TODO: IMPLEMENT CODE HERE

// Some reminders:

// See if the item exists.

// See if it can be bid upon.

Item item = this.itemsAndIDs.get(listingID);

synchronized(writelock){

// if(!this.itemsUpForBidding.contains(item))

// this.itemsUpForBidding.add(item);

}

if(item==null || !item.biddingOpen()){

return false;

}

// See if this bidder has too many items in their bidding list.

Integer bidCount = this.itemsPerBuyer.get(bidderName);

if(bidCount!=null && bidCount >= maxBidCount){

return false;

}

if(bidCount==null)

bidCount=0;

// Get current bidding info.

// See if they already hold the highest bid.

// See if the new bid isn't better than the existing/opening bid floor.

Integer currentBid = this.highestBids.get(item.listingID());

if(currentBid!=null && currentBid> biddingAmount){

return false;

}

//Acquire lock

else{

synchronized(writelock)

{

//Decrement the former winning bidder's count

if(this.highestBidders.get(item.listingID())!=null){

String formerBidder= this.highestBidders.get(item.listingID());

Integer formerbiddercount=this.itemsPerBuyer.get(formerBidder);

if(formerBidder.equals(bidderName)){

return false;

}

this.itemsPerBuyer.put(formerBidder, formerbiddercount-1);}

//PUT bidderName AND biddingAmount+1 IN A LIST OF ACTIVE BIDS FOR THE BIDDER;

//this.itemsPerSeller.put(bidderName, biddingAmount+1);

this.itemsPerBuyer.put(bidderName, bidCount+1);

//Release lock

// Put your bid in place

//Acquire lock on highestBidders

//PUT bidderName AND listingID IN HIGHEST BIDDERS LIST;

//Release lock

//Acquire lock on highestBids

//PUT listingID AND biddingAmount IN highestBids;

this.highestBids.put(item.listingID(), biddingAmount);

//Release lock;

//System.out.println("bid placed");

}

this.highestBidders.put(listingID ,bidderName);

return true;

}

}

/\*\*

\* Check the status of a <code>Bidder</code>'s bid on an <code>Item</code>

\* @param bidderName Name of <code>Bidder</code>

\* @param listingID Unique ID of the <code>Item</code>

\* @return 1 (success) if bid is over and this <code>Bidder</code> has won<br>

\* 2 (open) if this <code>Item</code> is still up for auction<br>

\* 3 (failed) If this <code>Bidder</code> did not win or the <code>Item</code> does not exist

\*/

synchronized public int checkBidStatus(String bidderName, int listingID)throws IllegalArgumentException

{

// TODO: IMPLEMENT CODE HERE

// Some reminders:

String HighestBidderName=null;

int price=0;

Item item= this.itemsAndIDs.get(listingID);

//Boolean itemexist=this.itemsUpForBidding.contains(item);

synchronized(writelock)

{

if(!item.biddingOpen()){

if(this.highestBids.get(item.listingID())!=null)

{

HighestBidderName=this.highestBidders.get(item.listingID());

price=this.highestBids.get(item.listingID());

}

// Update the number of open bids for this seller

Integer sellercount=this.itemsPerSeller.get(item.seller());

// Decrease the count of items being bid on by the winning bidder if there was any...

Integer bidcount= this.itemsPerBuyer.get(bidderName);

// If the bidding is closed, clean up for that item.

if(HighestBidderName !=null && HighestBidderName.equals(bidderName))

{

// Remove item from the list of things up for bidding.

//if(bidcount!=null)

this.itemsPerBuyer.put(bidderName,bidcount-1);

this.itemsUpForBidding.remove(item);

// this.highestBids.remove(item.listingID());

// this.highestBidders.remove(bidderName);

this.itemsPerSeller.put(item.seller(),sellercount-1);

this.revenue+=price;

this.soldItemsCount++;

return 1; //success

}

else

return 3; //failed

}

else

return 2 ; //open

}

}

/\*\*

\* Check the current bid for an <code>Item</code>

\* @param listingID Unique ID of the <code>Item</code>

\* @return The highest bid so far or the opening price if no bid has been made,

\* -1 if no <code>Item</code> exists

\*/

synchronized public int itemPrice(int listingID) throws IllegalArgumentException

{

// TODO: IMPLEMENT CODE HERE

int itemprice=0;

Item item = this.itemsAndIDs.get(listingID);

if(item!=null){

//Acquire lock for reading

synchronized(readlock)

{

if(this.highestBids.get(item.listingID())!=null)

itemprice= this.highestBids.get(listingID);

else

itemprice= item.lowestBiddingPrice();

}

//Release lock

/\*if(itemprice==null)

itemprice= item.lowestBiddingPrice();\*/

}

else

itemprice=-1;

return itemprice;

}

/\*\*

\* Check whether an <code>Item</code> has been bid upon yet

\* @param listingID Unique ID of the <code>Item</code>

\* @return True if there is no bid or the <code>Item</code> does not exist, false otherwise

\*/

synchronized public Boolean itemUnbid(int listingID) throws IllegalArgumentException

{

// TODO: IMPLEMENT CODE HERE

Item item= this.itemsUpForBidding.get(listingID);

//IF item is found in highestBids.get(item.listingID) THEN

Integer iteminbids=this.highestBids.get(item.listingID());

if(iteminbids>0)

return false;

else

return true;

}

synchronized public int highestbidssum()

{

int sum=0;

for(int i:this.highestBids.keySet())

{

sum+=this.highestBids.get(i);

}

return sum;

}

synchronized public void maxselleritemcount()

{

for(String seller:itemsPerSeller.keySet())

{

System.out.println(seller+':'+itemsPerSeller.get(seller));

}

}

synchronized public void servermaxbidcount(){

System.out.println("itemsup for bidding:"+this.itemsUpForBidding.size());

for(String buyer:this.itemsPerBuyer.keySet())

{

System.out.println(buyer+':'+this.itemsPerBuyer.get(buyer));

}

}

}