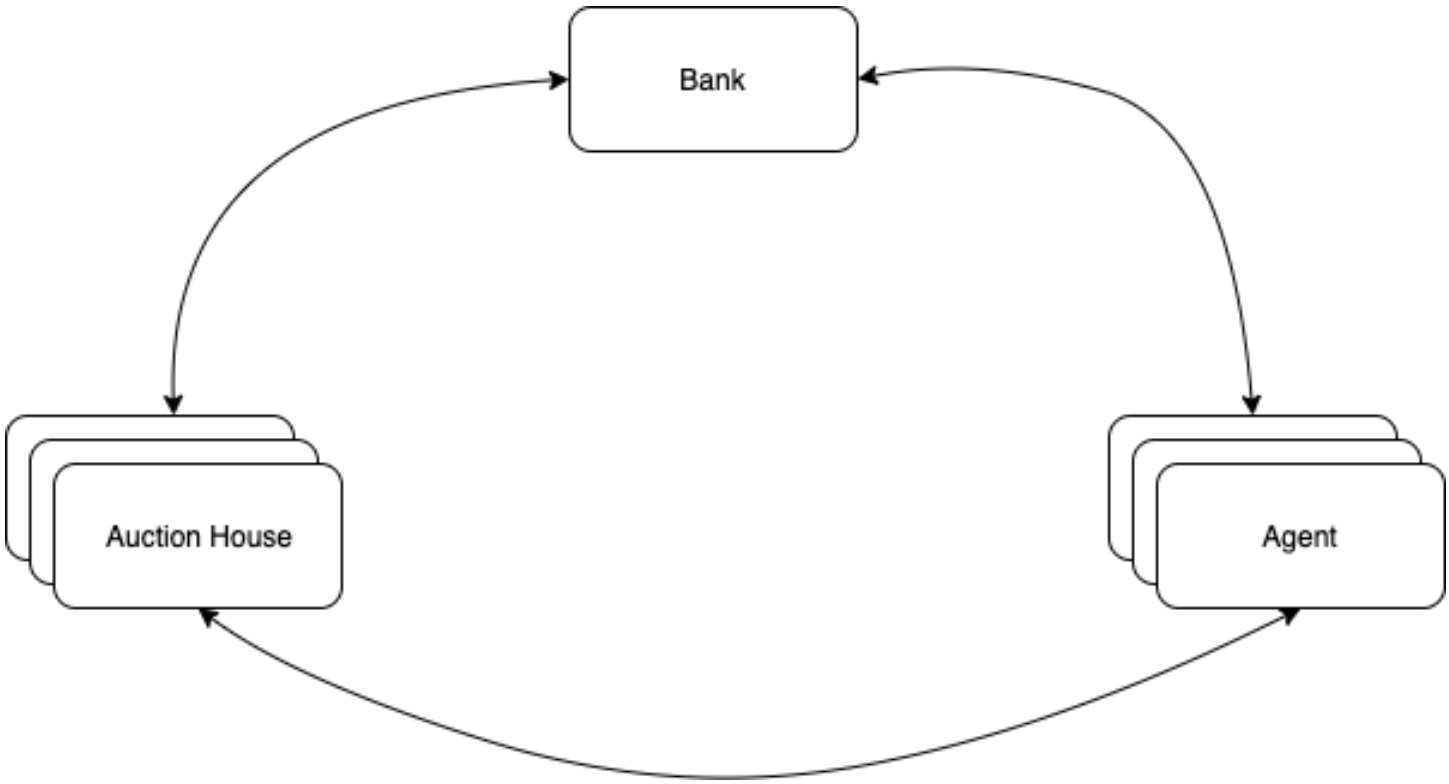
Distributed Auction Design

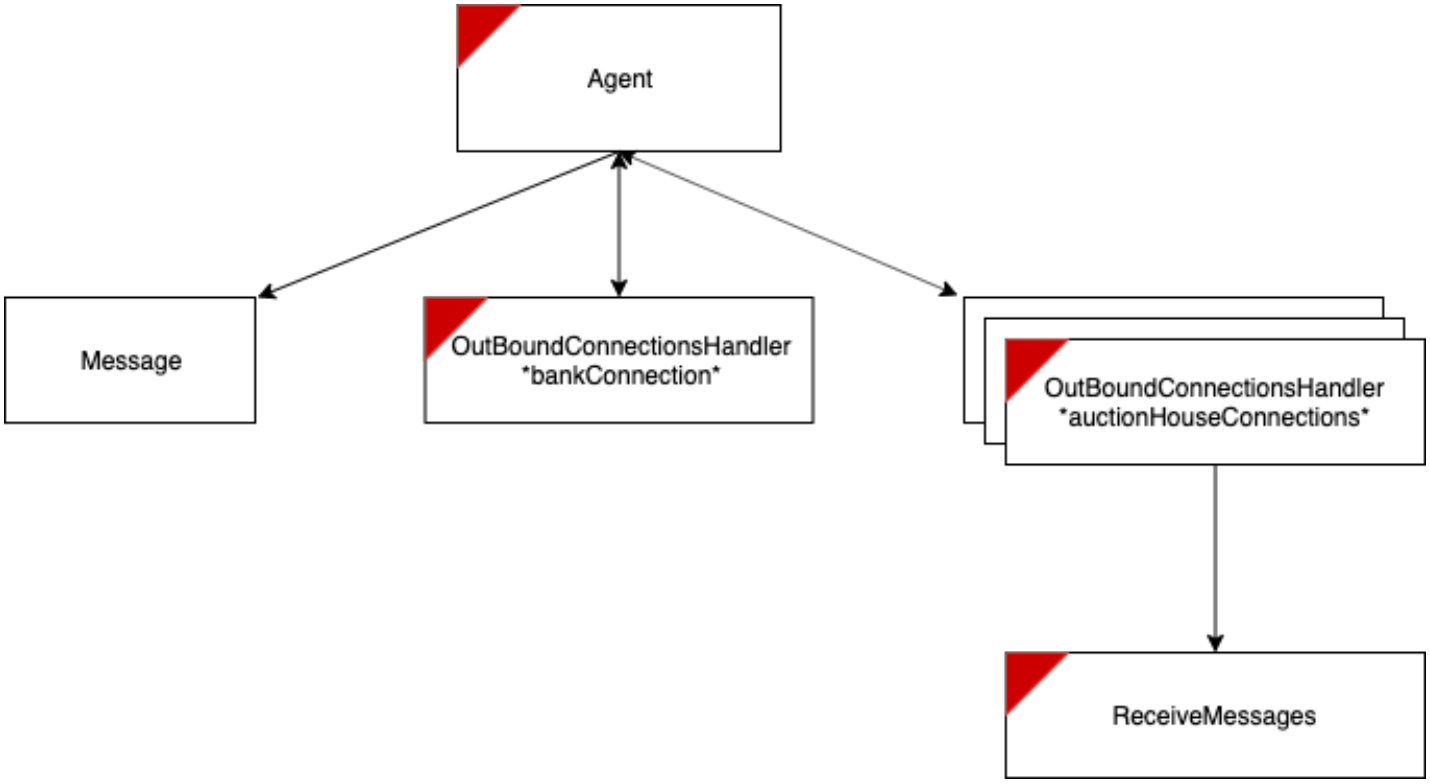
General Architecture

Note: The diagram below should not be considered as an object design diagram. It is simply here to help the reader and grader understand the structure of our Distributed Auction House Design and the communication between each part

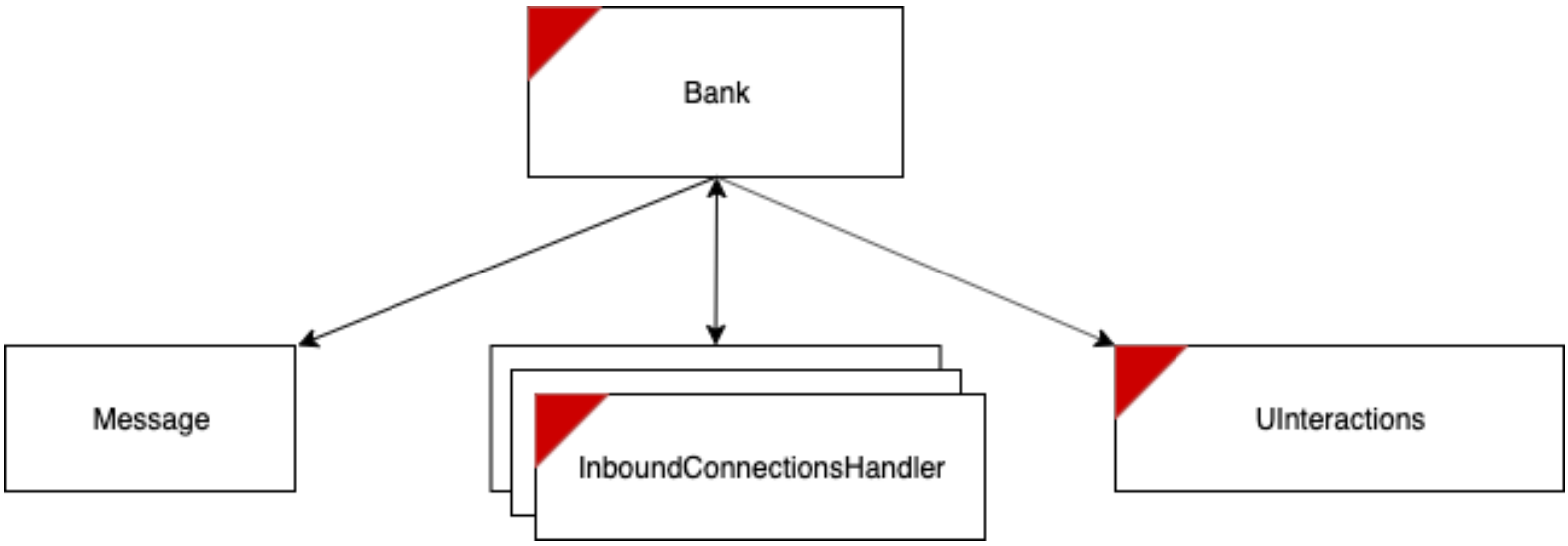


Object Oriented Diagrams

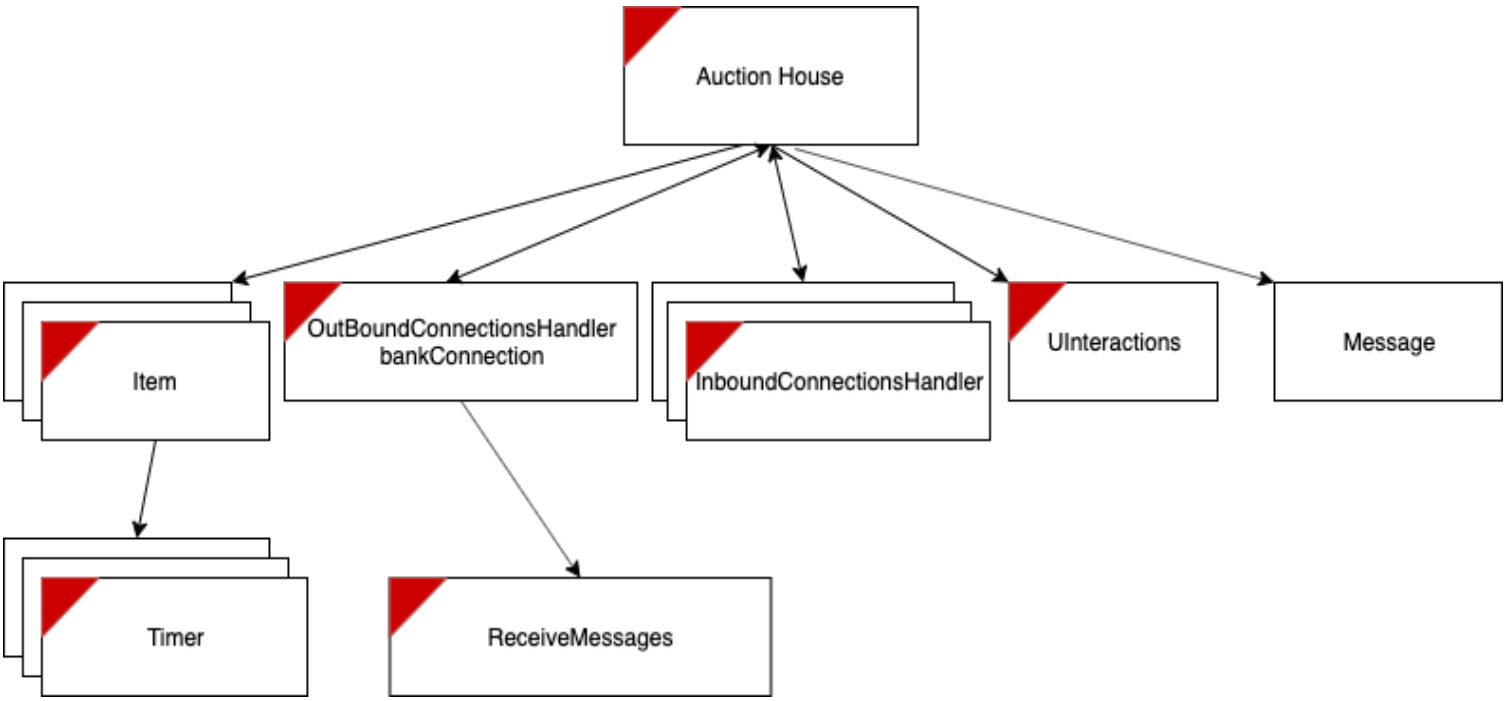
Agent



Bank



Auction House



Communications

All the communications implementations of the distributed auction program are conducted via sockets. Using sockets seemed intimidating at first, but after basing our code and implementations on the KnockKnock Client-Server example given by Oracle, setting up the communications in our design made much more sense.

We have divided our communications in two major elements: InBoundConnections (Server-side) and OutBoundConnections (Client-side). Both these elements are two fully separate classes implemented in our code. These two connections, however, work hand in hand. Indeed, an outBoundConnectionsHandler will request a new connection to the InboundConnectionsHandler. When this request is accepted, a communication channel will be created between both entities.

InBoundConnectionsHandler - Server

Before we get into depth on the implementation of our InBoundConnectionsHandler, it is important to note that two parts of our project will act as Servers: the Bank and the Auction Houses. The Bank will act as a Server for all of its communications. The Auction Houses will only act as Servers only for its communications with Agents. The inbound communications for the Bank and Auction Houses is quite straightforward and very much resembles the KnockKnockServer example given by Oracle: Listen for new connections, when a new connection arrives, start a new thread to handle that connection. Consequently, every single connection from Bank to Auction House or Agent and from Auction House to Agent, will be launched on a separate independent thread.

The InboundConnectionsHandler class can be reduced to a single purpose: infinitely wait for incoming message from the client. When a message is read, process that message by sending it back to Bank or Auction House (depending on the current connection), and send a response back on the received message to the client. In a few lines, we have defined the purpose of the InBoundConnectionsHandler. It is, indeed, a rather straightforward implementation.

The methods in InBoundConnectionsHandler are the following:

- InBoundConnectionsHandler
    - variableSetter
    - sendMessage
        - run

Each method is carefully commented and described as Javadocs in the code. For more information on the methods and variables used, please refer to the code.

OutBoundConnectionsHandler - Client

Here again, before we get into depth on the implementation of our OutBoundConnectionsHandler, it is important to note that two parts of our project will act as Clients: the Agents and the Auction Houses. The Agents will act as a client for all of its communication while the Auction Houses will act as client only for their communications with the Bank. For the Agent, a new OutBoundConnectionsHandler in the Agent is started every

time it connects to a new Auction House. Consequently, an Agent will have many different instances of OutBoundConnnectionsHandler, each running on an independent, separate thread. It will have one handler for the Bank, and several for the Auction Houses. As mentioned below, we have decided to automatically connect an Agent to an Auction House, whenever this Auction House appears. Thus, each time an Auction House is created and connects to the Bank, the Agent will instantiate a new instance of OutBoundConnectionsHandler and request a connection to the Auction House. For connections to the Bank, both the Agent and the Auction House will have to manually enter the Bank's IP address and port number in order to request a new connection between the Auction House/Agent OutBoundConnectionsHandler and the Bank's listening InBoundConnectionsHandler.

The user will note that we have defined a nested class in OutBoundConnectionsHandler called ReceiveMessages. We have quickly noted that our implementation of clients was not set up to receive messages without making a prior request. But what if the Bank or Auction House server sides decided to send a message to one of their clients after the occurence of a certain event? This is why we have implemented a small nested class called ReceiveMessages inside the client side OutBoundConnections. This class will basically continuously wait for incoming messages and send them back to the current client.

The methods in OutBoundConnectionsHandler are the following:

- OutBoundConnectionsHandler
    - setHostAndPort
    - setAuctionHouse
    - setAgent
        - run
- ReceiveMessages
    - connectNewAuctionHouse
    - parseReceivedMessage
        - run

Each method is carefully commented and described as Javadocs in the code. For more information on the methods and variables used, please refer to the code.

Agent

As mentioned in the README, the Agent class is basically the client part of the auction house program. It creates out-bound connections with the bank and the auction house and uses their services. In short the Agent class is the UI for this project. This class is dynamic as it creates a new agent every time we run it. As the agent.java file is executed The main method starts by asking to connect to the bank. We can connect to the bank using the IP address and the port number on which the bank waits for connections. As soon as the agent is connected to the bank, an outbound communication thread with the bank is created. After establishing the communication with the bank a while loop is started which waits for user inputs until the program is exited.

Each instruction is pushed as arguments into the method called **processMessagesToSend** which takes a destination for the message and the message (request) itself. All the requests by the agents are then processed in the **processMessagesToSend.** To get all the services provided by the bank the agent should have a bank account in the bank which is automatically created

as soon as the user connects to the bank. When the connection with the bank is set, the bank provides the agent with an account and a unique agentID. We have chosen the initial amount of each Agent's bank account to be 100. This can be easily modified in the code. At this point, we have also decided not to let the user deposit money in an Agent account. Consequently, once an Agent reaches an account balance of 0, it will not be able to conduct any bids anymore.

The agent can also connect to auction houses. We have decided that connections to auction houses will be automatic. That is, as soon as an auction house is created after the agent is running, the bank will send a message to the Agent and the connection Agent-AuctionHouse will be automatically made. A similar process is implemented for Auction Houses that have been created before the user/agent connects to the bank. However, we have also decided to implement a way for the user to manually connect to an Auction House in case errors or unexpected behavior might occur.

Finally, one of the most important aspects of the Agent is the bidding system, explained in more detail below. The agent will be able to bid on several items as proposed by the current auction houses. When an agent makes a bid, it will have to check with the bank if it has sufficient funds in its bank account. Whenever the bid is placed and won, the agent will have 30 seconds to transfer the money from the bank to the auction house's account. To do so, the user will have to ask the bank to transfer money.

We realized that using the agent for the first time might not necessarily be a straightforward task. Consequently, we have incorporated in our agent implementation a handy "help" menu that displays all the possible commands and their related description. At any time during the simulation, the user can simply type the command "help" to figure out which command to use and what steps to take in order to communicate with the bank or with an auction house.

It is also interesting to note that the Agent will be the only element of the distributed auction project that is strictly a client of a server-client relationship.

The methods in Agent are the following:

- Agent
  - setBidding
  - terminateAgent
  - setAgentID
  - removeAuctionHouse
  - connectToAuctionHouse
  - startBankCommunicationsThread
  - printHelpMenu
  - processMessagesToSend

Each method is carefully commented and described as Javadocs in the code. For more information on the methods and variables used, please refer to the code.

Auction House

The Auction House is meant to store Item objects that it can then sell to Agents when an agent bids on an item. Each Item object is supposed to represent a famous painting. An Auction House will always sell exactly three items at a time, unless they run out of items to sell. At this point, an Auction House would be able to sell less than three items. The Auction House when first run starts threads in order to handle InBoundConnections with Agents, the OutBoundConnection with the Bank, the UInteractions, and each of the Items that the Auction House is currently selling. Communication with the Auction House is implemented by these connection handlers and allows for the Auction House to process incoming messages and return messages to the sender for each message that was processed. The two methods **processIncomingMessage** and **processBankInformation** are where the Auction House is able to respond to the communications between Agents and the Bank. In addition, the Auction House is able to send messages through the **processMessageToSend** method which puts a Message into the BlockingQueue of a Bank or the outputToClient of the Agent within the InboundsConnectionHandler. When an Agent first sends a message to the Auction House, it is then able to request that the bank tell if the Agent ID it is receiving is currently connected to the Bank. If so, it adds this ID to a HashMap in order to reference this to send messages to that specific agent based on an ID. The ID takes form "A-#" where # is an integer.

Bidding System

Initially, the Auction House sells three items with random minimum selling values between 5 and 15. If the Auction House receives a request to bid on an item, several checks are made in order to make sure that they input is robust. These include does the item they are bidding on exist, is that item still in the bidding process, did they send enough money to reach the minimum bid, etc. If the Auction House decides that it is a legal bid, a message is then sent to the Bank in order to request that the Bank block their funds for the amount that they are bidding on. The Bank then replies to the Auction House with either a "blocked" or "bid was invalid" message. If the bid was invalid, then a message is sent to the agent telling them that they were not able to bid on the item. If their funds were blocked, then the item is then sent a message to either start a timer or to interrupt an ongoing timer. This timer lasts for 30 seconds and waits for bidding to occur. In addition to updating the Item's Timer it also updates the Item's bidding status (current highest bid in dollars), agentID for the highest bidding agent, the minimum bid, and send a message to the Agent that bid on the item saying that their bid was successful.

Finally, if the timer was interrupted instead of started then a message is sent to the current highest bidder that they were outbid on that item. If there are no bids within 30 seconds, then a message is sent to the Agent with the current highest bid stating that it is time to pay for the item. It is then their responsibility to transfer the funds to the Auction House that is selling them the item through the bank. If they send enough money to buy more than one item that they are eligible to buy, then they will receive both Items. If they send more money than needed, then they are refunded that amount and notified that they have been refunded for a specific amount of money. Finally, if they do not transfer the money within the given time (30 seconds) then the item is resold with a new minimum value between 5-15. If an Item is sold, then a new item will be listed immediately afterward for a random minimum values between 5 and 15.

Subclasses for Auction House

The only subclass for the AuctionHouse is the UInteractions which is solely used for testing purposes and the exit command. The AuctionHouse will be terminated if and only if

there is no bidding that is currently in progress for any of the items that the AuctionHouse is currently selling.

Subclasses for the Item

The only subclass for an Item is the Timer. This timer is used for waiting 30 seconds for bids and for waiting on money to be transferred to the Auction House. If this timer is interrupted, it exits with nothing done to the Item, and the Item then restarts the Timer. However, if the timer finishes then it will update the Statuses of the Item that it is "attached to".

The methods in AuctionHouse are the following:

- AuctionHouse
  - AuctionHouse
  - start
  - main
  - startBankCommunicationsThread
  - startUInteractions
  - startItemThread
  - pickItemsToSell
  - processMessageToSend
  - processIncomingMessage
  - processBankInformation
  - sortByLowestCost
  - findItem
  - sellItem
  - resellItem
  - sendMessageToAgent
  - sendMessageToBank
  - parseLine
  - isInteger
  - terminateAuctionHouse
  - UInteractions
    - run
- Item
  - Item
  - run
  - setBiddingStatus
  - setAgentIDWithHighestBid
  - setMinBid
  - getAgentIDWithHighestBid
  - getId
  - getBiddingStatus
  - getMinBid
  - getName
  - getInput
  - isAbleToShutDown
  - isAbleToInterrupt
  - isBidding
  - isSold

- Timer
  - run

Each method is carefully commented and described as Javadocs in the code. For more information on the methods and variables used, please refer to the code.

Bank

The bank acts as the main server for agents and auction houses. Inside its main method, a new server socket is started and then the bank waits for clients to connect via the inbounds connections handler class. Any new connection is handled on its own thread. The main information stored in the bank consists of maps that map agent ID's and auction house ID's to their balances as well as their sockets. The bank uses one subclass, that being the UInteractions class. When a new object of this class is created, and is ran on its own separate thread, a scanner is created that waits for user commands. Once the bank is running, simply typing in "bank info" (without the quotes, of course) will display information related to all currently connected auction houses and agents. Here, the balances of every client will be displayed, as well as a separate line dealing with an agent's blocked funds. How were the ID's of each client chosen? Well, the format is as follows: agents will have ID's that look like "A-#" and auction houses will have "AH-#." The numbers are determined by how many agents or auction houses have already connected. So, for example, if the bank starts and the first agent connects, its ID will be "A-1." Also, whenever a new connection or disconnection occurs, the bank will print messages acknowledging these.

Now is the time to discuss the blocked funds of agent. Whenever an agents bids on something, money is removed from its balance and placed into initially empty map called "agentToBlockedFunds". This map is used to make the transferring and unblocking of funds easier. This way, when transferring funds, only money needs to be removed from an agent's blocked funds, and not its balance. For instance, say an agent has 100 dollars and bids 10. It will then have 90 in its balance and 10 in its blocked funds. If a bid is won, that 10 dollars will be the money that is directly transferred. If an agent loses a bid, that 10 will be added back to its balance. Because clients will all be attempting to modify the same map, the bank's processIncomingMessage method is synchronized. The operations done by bank are not particularly time consuming, so only letting one client thread request the bank to do things at a time is perfectly fine. The main algorithm behind the bank's processIncomingMessage method is as follows:

1. Determine the source (client) of the incoming message
2. Use the contents of the message then to determine what actions to take
3. The method also takes a socket and InboundsConnectionsHandler; it is used mainly in relation to setting up of accounts.

If the bank needs to send a message to a client directly, it may do so by first identifying it through its socket. This will get the client's inboundConnectionsHandler which comes with its own sendMessage method. Otherwise, the bank's processIncomingMessage will return a string back to the client that first sent a message. Methods that deal with blocking or transferring (or vice-versa) have some error checking in them. Bids will be rejected if an agent doesn't have enough money, for example. Say the agent decides to transfer too much money, this will be seen by the bank and it'll print a message. However, the more detailed error message will be printed by the agent interface.

As one final note, the initial balances for each client are determined by constants, 100 dollars for the agents and 0 for the auction houses.

The methods in Bank are the following:

- Bank
    - main
    - blockFunds
    - unblockFunds
    - startUInteractions
    - transferFunds
    - untransferFunds
    - processIncomingMessage
    - sendMessageToAgents
    - sendMessageToAuctionHouse
    - parseString
    - UInteractions
        - run

Each method is carefully commented and described as Javadocs in the code. For more information on the methods and variables used, please refer to the code.