INFO1113 Assignment 1

Due: 24th September 2021, 23:59

This assignment is worth 14% of your final assessment

Task Description

In this assignment you will develop a Stock Exchange application in the Java programming language. Use the examples below and the Javadoc found under resources on EdStem to write the program to specification.

You are encouraged to ask questions on Ed using the assignments category. As with any assignment, make sure that your work is your own, and you do not share your code or solutions with other students.

Working on your assignment

You can work on this assignment on your own computer or the lab machines. It is important that you continually back up your assignment files onto your own machine, external drives, and in the cloud.

You are encouraged to submit your assignment on Ed while you are in the process of completing it. By submitting you will obtain some feedback of your progress on the sample test cases provided.

Implementation details

Write a program in Java that implements the Stack Exchange application as documented in the Javadoc on EdStem. You can assume that our test cases will contain only valid input commands and not cause any integer overflows. Commands are case insensitive.

The Exchange stores details on all the registered traders, as well as a reference to the matching engine (market) used to process orders and trades. Traders are identified by their ID, and store both their balance (in dollars) as well as their inventory, which tracks the products and quantities that the trader possesses.

The Market stores collections of orders (within sell and buy books, for sell and buy orders respectively) and trades. An order can be made by a trader to buy or sell an amount of a product for a given price per unit. Each order has a unique ID hexadecimal String of 4 characters. Order IDs begin at 0000 and increment up (such that the second order will have ID 0001). Note that as the IDs use hexadecimal values, after 0009 is 000A, and after 000F is 0010.

A trade is for a single product for a set amount at a given price per unit and occurs between a sell order and a buy order. Each trade represents a completed transaction in which funds and product have been transferred between traders.

The Market utilises the Price-Time Priority Algorithm (explained below) when a new order is made to the engine. When an order is created, it is initially open. The Price-Time Priority Algorithm then determines the best order from the corresponding book (sell book for a new buy order, buy book for a new sell order), if any, with which a trade can be made. If a trade can be made, the price from the existing order is used, and the minimum amount of the two orders is transferred. Note that as the price is per unit, the money transferred is equal to $price * amount$. After the money and product is transferred, if one or both of the orders no longer require any more amount, the order(s) is closed and is removed from the order book. If an order cannot be matched, or remains open after going through one or more trades, it is added to the market's order book. Examples of this process are provided below.

Your program should only be comprised of Exchange.java, Market.java, Trader.java, Trade.java and Order.java. Do not modify any of the existing method signatures found in the documentation. You are encouraged to write your own additional methods to improve style and reduce repetitive code. Your program must produce no errors when built and run. Your program will read from standard input and write to standard output.

Your program output must match the exact output format shown in the examples below. You are encouraged to submit your assignment while you are working on it, so you can obtain some feedback.

In order to obtain full marks, your program will be checked against automatic test cases, manually inspected by your tutors and you must submit a set of test cases that ensure you have implemented functionality correctly.

Price-Time Priority Algorithm

The Price-Time Priority Algorithm first prioritises by price, and then uses time to differentiate between orders at the same price. If a sell order is made, the algorithm will find a corresponding buy order for the same product with the highest price greater or equal to the price specified by the sell order. If there are multiple such buy orders, the earliest one is used.

If a buy order is made, the algorithm will find a corresponding sell order for the same price with the lowest price less than or equal to the price specified by the buy order. If there are multiple such sell orders, the earliest one is used.

Algorithm Example

Initially both the buy book and the sell book are empty. Let there be 5 traders in the market, with ID's trader1, trader2, trader3, trader4 and trader5, each with $100 dollars balance, and assume that trader1 has imported 100 units of product ABC.

trader2 wants to buy ABC and decides to make a buy order for it. They want to purchase 10 units, and the most they are willing to pay per unit is $7.50. They hence make a buy order for 10 units of ABC at $7.50 per unit. As the sell book is currently empty, no trades can be made, so the order is added unchanged to the buy book.

trader3 also wishes to buy some ABC and hence also makes a buy order for it. They want to purchase 5 units, and the most they are willing to pay per unit is $10.00. They hence make a buy order for 5 units of ABC at $10.00 per unit. As the sell book is still empty, no trades can be made, so the order is added unchanged to the buy book, which now has trader2's order (with ID 0000) and trader3's order (with ID 0001).

trader4 realises that ABC must be a popular product and will probably rise in value soon, so hence decides to make a buy order for it. They want to purchase 15 units, but for a maximum price per unit of $5.00. They hence make a buy order for 15 units of ABC at $5.00 per unit.

trader5 comes to the same conclusion as trader4 and makes a buy order for 20 units of ABC with a maximum price per unit of $7.50.

trader1 notices the interest in the market for ABC and decides to sell all of their stock. They decide to make a sell order for 100 units of ABC, with the lowest price they will accept be $6.50. The buy book is currently not empty, so the Price-Time Priority Algorithm gets to work to determine whether any trades can be made.

Currently the buy book is as follows:

Order ID	Product	Trader	Amount	Price per unit $
0000	ABC	trader2	10	7.50
0001	ABC	trader3	5	10.00
0002	ABC	trader4	15	5.00
0003	ABC	trader5	20	7.50

As the algorithm is attempting to match with a buy order, it will first find the order(s) with the highest price per unit (note that if it were trying to match with a sell order, it would find the orders with the lowest price). Order 0001 has the highest price per unit at $10.00. As this price is above the minimum price specified by the sell order, a trade is possible. 0001 only wants to purchase 5 units, hence trader1 transfers 5 units of ABC to trader3, and trader3 transfers 5x$10 back to trader1, completing the trade. trader1 now has a balance of $150, and trader3 a balance of $50.

As there are still 95 units of ABC yet to be sold, the sell order is not closed. Instead, the algorithm attempts to make another trade. The buy book is now as follows:

Order ID	Product	Trader	Amount	Price per unit $
0000	ABC	trader2	10	7.50
0002	ABC	trader4	15	5.00
0003	ABC	trader5	20	7.50

There are now two orders with the highest price, 0000 and 0003 with a price per unit of $7.50. To differentiate between the two, the algorithm prioritises by time and chooses the earliest made order, which is 0000 from trader2. As the price is above the minimum price specified by the sell order, a trade can be made. trader1 transfers 10 units of ABC to trader2, and trader2 transfers 10x$7.50 back to trader1.

There are still 85 units of ABC to be sold, so the algorithm continues to try making more trades. The buy book is now as follows:

Order ID	Product	Trader	Amount	Price per unit $
0002	ABC	trader4	15	5.00
0003	ABC	trader5	20	7.50

The highest priced order is 0003 at $7.50 per unit, which is above the minimum price for the sell. A trade is hence made for 20 units of ABC at $7.50 per unit, with 20 units transferred to trader5 and 20x$7.50 transferred to trader1.

The sell order remains open as there are still 65 units to be sold. The algorithm continues, this time with the buy book as follows:

Order ID	Product	Trader	Amount	Price per unit $
0002	ABC	trader4	15	5.00

The only order in the book has a price per unit of $5.00, which is below the minimum price specified by the sell order. A trade hence cannot be made.

As the algorithm is unable to identify an order to make a trade with, trader1's sell order cannot be closed at the current stage and instead is added to the sell book. The remaining 65 units of ABC in the order are transferred to the market. Note that this is not the same for when a buy order goes unmatched (such as with trader2's buy order from earlier), in which case the money is transferred at the time of sale.

After this process the buy book is as follows:

Order ID	Product	Trader	Amount	Price per unit $
0002	ABC	trader4	15	5.00

And the sell book is as follows

Order ID	Product	Trader	Amount	Price per unit $
0004	ABC	trader1	65	6.50

Additionally, the balances and inventories of the traders are as follows:

Trader ID	Balance $	Amount of ABC
trader1	375.00	0
trader2	25.00	10
trader3	50.00	5
trader4	100.00	0
trader5	-50.00	20

Commands

Your program should implement the following commands:

EXIT
Ends the program execution and outputs "Have a nice day."

ADD [id] [balance]
Adds a new trader to the exchange with the given ID and initial balance.

If a trader with the given ID already exists, output "Trader with given ID already exists."

If the initial balance is negative, output "Initial balance cannot be negative."

If the trader is successfully added, output "Success."

BALANCE [id]
Outputs the current balance of the given trader, preceded by a "$" symbol, with the balance printed with 2 decimal places.

If no traders have the given ID, output "No such trader in the market."

INVENTORY [id]
Outputs the product strings for all products in the trader's inventory, in alphabetical order, each on a new line.

If the trader has an empty inventory, output "Trader has an empty inventory."

If no traders have the given ID, output "No such trader in the market."

AMOUNT [id] [product]
Outputs the amount of product the trader has in their inventory, to 2 decimal places.

If the product is not in the trader's inventory, output "Product not in inventory."

If no traders have the given ID, output "No such trader in the market."

SELL [id] [product] [amount] [price]
Makes a sell order for the given trader for the given amount of the product with a set minimum price.

If the order is closed after the matching engine processes it, output "Product sold in entirety, trades as follows:", followed by the list of trades done in order processed, with a trade per line.

If the order is not closed after the matching engine processes it, but one or more trades were completed, output "Product sold in part, trades as follows:", followed by the list of trades done in order processed, with a trade per line.

If the order is not closed after the matching engine processes it and no trades were completed, output "No trades could be made, order added to sell book."

If the trader does not exist, output "No such trader in the market."

If the order could not be made for any other reason, output "Order could not be placed onto the market."

BUY [id] [product] [amount] [price]
Makes a buy order for the given trader for the given amount of the product with a set maximum price.

If the order is closed after the matching engine processes it, output "Product bought in entirety, trades as follows:", followed by the list of trades done in order processed, with a trade per line.

If the order is not closed after the matching engine processes it, but one or more trades were completed, output "Product bought in part, trades as follows:", followed by the list of trades done in order processed, with a trade per line.

If the order is not closed after the matching engine processes it and no trades were completed, output "No trades could be made, order added to buy book."

If the trader does not exist, output "No such trader in the market."

If the order could not be made for any other reason, output "Order could not be placed onto the market."

IMPORT [id] [product] [amount]

Imports the given amount of the product to the trader's inventory.

On success, output "Trader now has <total> units of <product>.", replacing <total> with the total amount of the product they have (2 decimal places), and <product> with the product code.

If the amount is negative or zero, output "Could not import product into market."

If the trader does not exist, output "No such trader in the market."

EXPORT [id] [product] [amount]

Exports the given amount of the product out of the trader's inventory.

On success, output "Trader now has <total> units of <product>.", replacing <total> with the total amount of the product they have (2 decimal places), and <product> with the product code.

If the amount is negative or zero, or exceeds the amount the trader has in their inventory, output "Could not export product out of market."

If the trader does not exist, output "No such trader in the market."

CANCEL SELL [order]

Cancels the sell order with the given ID and outputs "Order successfully cancelled."

If the order is not in the sell book, output "No such order in sell book."

CANCEL BUY [order]

Cancels the buy order with the given ID and outputs "Order successfully cancelled."

If the order is not in the buy book, output "No such order in buy book."

ORDER [order]

Displays the String representation of the order with the given ID (e.g., "000A: BUY 10.00xABC @ $12.50").

If there are no orders in the market, output "No orders in either book in the market."

If no order has the given ID, output "Order is not present in either order book."

TRADERS

Outputs the ID of all traders in the exchange, each on a new line, ordered alphabetically.

If there are no traders, output "No traders in the market."

TRADES
Outputs the String representation of all completed trades (e.g. “trader1->trader2: 10.50xABC for $1.20”), each on a new line, ordered by completion time (oldest to most recent).

If there are no completed trades, output “No trades have been completed.”

TRADES TRADER [id]
Outputs the String representation of all completed trades involving the given trader, each on a new line, ordered by completion time.

If the trader has not been involved in any trades, output “No trades have been completed by trader.”

If no traders have the given ID, output “No such trader in the market.”

TRADES PRODUCT [product]
Outputs the String representation of all completed trades involving the given product, each on a new line, ordered by completion time.

If no trades have been completed with the product, output “No trades have been completed with given product.”

BOOK SELL
Outputs the String representation of all orders in the sell book, each on a new line, ordered by the time they were processed by the engine.

If there are no orders in the sell book, output “The sell book is empty.”

BOOK BUY
Outputs the String representation of all orders in the buy book, each on a new line, ordered by the time they were processed by the engine.

If there are no orders in the buy book, output “The buy book is empty.”

SAVE [trader-path] [trades-path]
Saves the current trader state as well as log of all trades to the two given files and outputs “Success.” For each trader, the string representation (e.g. “trader1: $100.00 {ABC: 10, NFL: 12}”) is printed on a new line, in alphabetical order of trader ID. For each trade, the string representation is printed on a new line, in temporal order (oldest to most recent).

If there are any issues with saving to file (for example, lacking writing privileges, output “Unable to save logs to file.”

BINARY [trader-path] [trades-path]
Saves the current trader state as well as a log of all trades to the two given files in binary format and outputs “Success.” For each trader, the string representation is written and is then followed by the ASCII symbol for Unit Separator (ASCII 31), with all traders written out in alphabetical order. For each trade, the string representation is written and followed by Unit Separator, all in temporal order.

Examples (1)

```
$ BALANCE trader1
No such trader in the market.

$ INVENTORY trader1
No such trader in the market.

$ AMOUNT trader1 ABC
No such trader in the market.

$ SELL trader1 ABC 10 1.5
No such trader in the market.

$ BUY trader1 ABC 12.5 1
No such trader in the market.

$ IMPORT trader1 ABC 100
No such trader in the market.

$ EXPORT trader1 ABC 1.4
No such trader in the market.

$ CANCEL SELL 0000
No such order in sell book.

$ CANCEL BUY 000A
No such order in buy book.

$ ORDER AFEC
No orders in either book in the market.

$ TRADERS
No traders in the market.

$ TRADES
No trades have been completed.

$ TRADES TRADER trader1
No such trader in the market.

$ TRADES PRODUCT ABC
No trades have been completed with given product.

$ BOOK SELL
The sell book is empty.

$ BOOK BUY
The buy book is empty.

$ EXIT
Have a nice day.
```

Examples (2)

```
$ add trader1 100
Success.

$ add trader2 100
Success.

$ add trader3 100
Success.

$ add trader4 100
Success.

$ add trader5 100
Success.

$ import trader1 ABC 100
Trader now has 100.00 units of ABC.

$ buy trader2 ABC 10 7.5
No trades could be made, order added to buy book.

$ buy trader3 ABC 5 10
No trades could be made, order added to buy book.

$ buy trader4 ABC 15 5
No trades could be made, order added to buy book.

$ buy trader5 ABC 20 7.5
No trades could be made, order added to buy book.

$ sell trader1 ABC 100 6.5
Product sold in part, trades as follows:
trader1->trader3: 5.00xABC for $10.00.
trader1->trader2: 10.00xABC for $7.50.
trader1->trader5: 20.00xABC for $7.50.

$ book buy
0002: BUY 15.00xABC @ $5.00

$ book sell
0004: SELL 65.00xABC @ $6.50

$ inventory trader2
ABC

$ inventory trader1
Trader has an empty inventory.

$ exit
Have a nice day.
```

Examples (3)

```
$ ADD trader1 0
Success.

$ IMPORT trader1 NFL 10
Trader now has 10.00 units of NFL.

$ ADD trader2 0
Success.

$ TRADERS
trader1
trader2

$ SELL trader1 NFL 10 1
No trades could be made, order added to sell book.

$ BUY trader2 NFL 10 2.5
Product bought in entirety, trades as follows:
trader1->trader2: 10.00xNFL for $1.00.

$ EXPORT trader2 NFL 5
Trader now has 5.00 units of NFL.

$ BALANCE trader1
$10.00

$ BALANCE trader2
$-10.00

$ TRADES
trader1->trader2: 10.00xNFL for $1.00.

$ SAVE traders trades
Success.

$ BINARY traders.bin trades.bin
Success.

$ EXIT
Have a nice day.
```

Examples (4)

```
$ ADD trader1 100.55
Success.

$ IMPORT trader1 APL 12
Trader now has 12.00 units of APL.

$ BALANCE trader1
$100.55

$ INVENTORY trader1
APL

$ AMOUNT trader1 APL
12.00

$ SELL trader1 APL 10 1.56
No trades could be made, order added to sell book.

$ BOOK SELL
0000: SELL 10.00xAPL @ $1.56

$ ORDER 0000
0000: SELL 10.00xAPL @ $1.56

$ CANCEL SELL 0000
Order successfully cancelled.

$ BUY trader1 NFL 10 1
No trades could be made, order added to buy book.

$ BOOK BUY
0001: BUY 10.00xNFL @ $1.00

$ EXIT
Have a nice day.
```

Writing your own testcases

We have provided you with some test cases, but these do not test all the functionality described in the assignment. It is important that you thoroughly test your code by writing your own test cases.

You should place all of your test cases in the provided tests/ directory. Ensure that each test case has a .in input file along with a corresponding .out output file (matching the samples provided, which correspond to the four examples above). We require that the names of your test cases are descriptive so that you know what each is testing, e.g. import.in and import.out and we can accurately and quickly assess your test cases. **Note: If you do not format your test case files as explained (where each test case has <name>.in and <name>.out files for input and output), you shall receive 0 for this component.**

Your test cases will be run against a working version of the codebase, which will provide details as to any differences in output as well as the overall code coverage of your test cases as a percentage of statements covered. Note that the code has not been obfuscated in any way (for example with the use of random if statements that are only covered when certain variables have meaningless values). To assist you with determining which branches are yet to be covered, you have also been provided with code that creates a detailed coverage report. To run this aspect, navigate to the root of the project, execute “bash coverage.sh” in the terminal, and then open jacoco/report/index.html in web-view.

Submission Details

Final deliverable for the correctness and manual inspection will be due on the 24th of September at midnight.

You must submit your code and tests using the assignment page on Ed. To submit, simply place your files and folders into the workspace, click run to check your program works and then click submit.

You are encouraged to submit multiple times, but only your last submission will be considered.

Marking

You will only be given valid inputs as part of the automatic test suite. Your program will be checked for errors that a user can possibly make. In addition, we will mark your program against a substantial collection of hidden test cases.

9 marks are assigned based on automatic tests for the correctness of your program. This component will use hidden test cases that cover every aspect of the specification. Your program must match the exact output in the examples and the test cases on Ed.

4 marks are assigned based on the code coverage of your own test cases. Make sure that you carefully follow the assignment specifications regarding the structuring of your test cases.

1 mark is assigned based on a manual inspection of the style. Style will be assessed based on the conventions set out in the Google Java Style Guide (https://google.github.io/styleguide/javaguide.html)

Additional Notes

Use *String.format()* to get a String representation of a double to a specified number of decimal places.

To sort a list of Strings lexicographically, use *list.sort((x, y) -> x.compareTo(y));*

Academic declaration

By submitting this assignment you declare the following: