

COMP3506/7505 Assignment 1

This assignment is worth **15%** of the total course marks.

Administration

Due 5pm Friday 2 September 2011 (Week 6)

Assignments will be returned during tutorials. Refer to the course profile for policy regarding late submission of assignments.

School Policy on Student Misconduct

You are required to read and understand the School statement on student misconduct, available at:

<http://www.itee.uq.edu.au/index.html?page=138114>

In particular, you must familiarise yourself with UQ's definition of plagiarism, and understand the School's policies regarding student plagiarism.

Submission

The assignment must be submitted through the ITEE online assignment submission service:

<http://submit.itee.uq.edu.au>

All parts of the assignment must be in a single zip archive, created by exporting the modified project from Eclipse. Any written-answer questions must be answered in a separate PDF document named "report.pdf" and placed in a "doc" directory in your project. To then create the archive:

- Select the File → Export... menu item
- Select General → Archive File (as the export destination)
- Enter the destination file name, which must be Student<your_student_number>.zip with <your_student_number> replaced by your full eight-digit student number.

Export the project only **after** you have completed all parts of the assignment. Please ensure that your exported archive file contains all of your work before submitting it.

Coding Style

Your code must be formatted using the built-in Eclipse coding style. Open the Eclipse "Preferences" window, go to Java → Code Style → Formatter and under "Active Profile" make sure that "Eclipse [built-in]" is selected. You should reformat your code to follow this coding style before submission, by selecting Source → Format or Source → Format Element.

Resources and restrictions

You **may not** use any classes from the Java Class Library except: those in the *java.lang* package, and *java.util.StringTokenizer*. You **may not** use any code that you did not author, except for code provided to you by us.

To complete this assignment you must first download an Eclipse archive file from:

<http://www.itee.uq.edu.au/~comp3506/Resources>

Import the downloaded archive into a new Eclipse workspace using the “Existing Projects into Workspace” option:

1. Select the File → Import menu item
2. For the Import Source, select General → Existing Projects into Workspace and click “Next”
3. Select “Select archive file”, “Browse”, and select the archive file and click “OK”
4. Select the available project and click “Finish”

Introduction

You have landed a summer research assistant position at a university, writing framework code to support modelling of stock market dynamics. You don't need to support the more complicated bookkeeping aspects of a market, but merely to have a system that tracks submission and fulfilment of buy/sell orders. As a way to get things started, your supervisor has sketched out some basic requirements of this system.

The system must be able to:

- Read sequences of buy and sell orders from a flat file format
- Store the sequences of buy and sell orders
- Simulate transactions

A buy order looks like:

```
buy ABCD 100 $1.99
```

which means “Offer to buy 100 units of the stock identified as ABCD at a price of \$1.99 per unit”. A sell order looks much the same:

```
sell EFGH 50 $2.99
```

meaning “Offer to sell 100 units of the stock identified as EFGH at a price of \$2.99 per unit”.

Simulating transactions involves matching potential buy orders against potential sell orders, and processing the orders. A buy and a sell order *match* if they have the same stock code, and the sell order unit price is less than or equal to the buy order unit price; the number of units being sold is not relevant to making a match, only to the transaction as a whole. If the selling price is lower than the buying price, the stock is transferred at the selling price. Processing an order

involves determining whether one or both of the buy or sell orders are completely fulfilled, removing fulfilled orders, and re-adding any remaining order to the relevant order set.

For example, the commands:

```
buy ABCD 20 $10.00
sell ABCD 100 $20.00
```

would not result in a match since the selling price is larger than the buying price. However, the commands:

```
buy ABCD 20 $10.00
sell ABCD 100 $5.00
```

would result in a match. In this case, 20 units of stock ABCD will be transferred at a price of \$5.00 per unit, and this transaction will be recorded in the transaction set. Both orders would be removed from their order sets, and since 80 stocks remain for the sell order, the sell order will be modified and re-added to the sell order set; in this case, this results in a new sell order of:

```
sell ABCD 80 $5.00
```

being added to the sell order set.

Buy orders and sell orders are both processed in a first-in, first-out manner. When an order is partially fulfilled, sell and buy orders are treated differently: partially-fulfilled buy orders are re-added to the buy order set, but are treated as new orders (i.e., they will be matched after existing buy orders); partially-fulfilled sell orders are re-added to the sell order set as the first in line (i.e., they will be matched against first on the next transaction attempt).

A single transaction attempt takes place as follows:

- A buy order is obtained; buy orders must be processed sequentially in order of being added (i.e., initially, in the order that they were read from the file)
- The system attempts to match sell orders one-by-one in sequential fashion, in the order that they were added; matching is defined above. A sell order that does not match the current buy order is re-added to the sell order set as the last element in the set (i.e., it will now be matched after all other existing sell orders)
- If there is a matching order pair, orders are processed. If there is not a matching sell order for the current buy order, the buy order is re-added to the buy order set as the last element in the set (i.e., it will now be matched after all other existing buy orders)

Your job is to implement parsing of buy/sell orders and to efficiently implement the transaction logic. A single simulation involves repeatedly performing transactions until no more transactions are possible on the sell and buy sets.

You will need to decide upon data structures for storing the buy and sell orders, taking into account the computational and space complexities involved. This system will eventually be used for large-scale simulation of market transactions, so efficiency considerations are important in a big-O complexity sense, but also in terms of a constant factors.

Available code

There are four implementations of data structures that have been provided to you: array-based and linked list-based implementations of Vector and Queue data types, each of which has their own strengths and weaknesses.

Part 1 – Parsing Input (3 marks)

Implementation of the `parseInput` method

You must implement the `parseInput` method in the `Agent` class. This method will take one `String` argument as input, which is the name of a text file. The method will then open the given file, and parse buy and sell commands. Each command will be on a new line and of the form:

```
<transaction type> <stock name> <quantity> $<price>
```

Where `<transaction type>` is either `buy` or `sell`, the `<stock name>` is exactly 4 alphabetical characters, `<quantity>` is an integer and `<price>` is a double.

The `parseInput` method must read each line from the file and convert the command into a `Stock` object. The `Stock` must then be added to the appropriate buy or sell set.

The `parseInput` method must return 0 if there are no errors in the parse and all stocks were added to their respective sets successfully. If a command is not well-formed, then the method must print to standard error the following:

```
Command on line <line number> is not well formed.
```

where `<line number>` is the number of the line where the command is. The method should keep attempting to parse all lines within the file and store the results, but it must return -1 to signal that there was an error during parsing.

You may assume that there is at most one command on each line, but there may be blank (i.e., no command) lines. If the line has no commands, you must skip it (and not register an error). You must keep parsing until there are no more lines to parse in the file.

See “Resources and restrictions” above. You may choose the data structures for each of the buy and sell sets from the implementations provided in the `datastructures` folder. You may also use any classes that were given to you in the imported “COMP3506-A1” project.

You will find the `Agent` class within the `studentXXXXXXXX` package. Please replace `XXXXXXXX` with your student number. For example, if your student number is 12345678, you would rename the `studentXXXXXXXX` package to `student12345678`. **Any additional classes you write must be within this package.**

Hint: You should read Part 2 before deciding upon the data structures for your buy and sell sets. You can set the data structures you are using in the constructor of the `Agent` class. The operations performed in Part 2 should have some influence on your decisions.

Implement the `printQueues` method

As a debugging and verification tool, you must implement the `printQueues` method in the `Agent` class. This method may be called at any time, but would typically be called after input is parsed. It returns a string describing the current set of buy and sell orders, in the order in which they will be processed. All buy orders are listed first, followed by all sell orders:

```
buy ABCD 20 $10.00
buy ABCD 80 $8.00
buy GOOG 100 $50.00
sell ABCD 100 $20.00
sell GOOG 10 $2.00
```

Part 2 – Processing Transactions (7 marks)

Implement the `exchange` method

Now that you can populate your buy and sell order sets, you need to implement the actual transaction logic in the `exchange` method in the same `Agent` class as above. This method tries to match stocks between the buy and sell orders according to the requirements outlined in the introduction section above. The method should iterate through buy orders in the order in which they were received, and try and match them to sell orders as described above, recording each resulting transaction. The `exchange` method finishes when no more buy/sell orders can be matched.

In implementing this method, carefully consider the most appropriate abstract data types to use for each component, including any expected differences in constant factors. You can set the data structures you are using in the constructor of the `Agent` class.

Implement the `printTransactions` method

In order for your simulation tool to provide the results of the recorded transactions, you need to implement the `printTransactions` method in the `Agent` class. This method should be called after input is parsed and the `exchange` method has been called. It generates a string describing the series of successful transactions, specifying the stock, the number of units purchased and the price at which the stock was sold, in the form:

```
ABCD 20 $5.00
```

meaning that 20 units of stock ABCD were sold at \$5.00 per unit. The set of transactions must be printed in the order in which the transactions happened. Each transaction must be on a new line, e.g.:

```
ABCD 20 $5.00
EFGH 10 $2.99
```

Test your methods

A set of JUnit test cases can be found in the `test` package. You should check that your implementation produces the correct outputs for these cases. Note that these are *not* the full set of behavioural requirements, but are instead just a few test cases to help you out. You are advised to add more test cases to test your code thoroughly.

Part 3 – Analysis (5 marks)

Analyse performance

In order to verify that your system is reasonably efficient, you must measure the performance of your system. To do so, write a main class (or copy the provided one) that reads a set of buy and sell orders from a file and measures the time taken for your `Agent's exchange` method to run.

Hint: One simple measurement option is to call `System.currentTimeMillis` before and after each exchange.

Choice of buy and sell set data structures

Using **less than 300 words**, outline why you chose the particular data structures you used for the buy and sell sets used in Part 1 and 2. Discuss the advantages and disadvantages of your choices, focusing on the time and space complexities and the measured performance. You *may* wish to include measured performance for using the alternative data structures.

Part 4 – Improvements (COMP7505 students only – 5 marks)

Improving the `ArrayQueue` structure

The current `ArrayQueue` implementation moves the elements of the array forward as the front element is removed in the `dequeue` operation. Copy the contents of the `ArrayQueue` class and produce a new `ImprovedArrayQueue` class. Modify the `dequeue` and `enqueue` methods so that these operations can be performed in $O(1)$ time.

Replace whatever queue-based data structure your code was previously using with the `ImprovedArrayQueue` class.

Does the `ImprovedArrayQueue` significantly increase or decrease performance relative to the other implementations? Justify your answer using either performance measurements (e.g., timing details on a large data set, shown as a histogram), or a short (no more than 200 words) explanation.