CS1003 Week 3 Practical: File Processing

This practical is worth **20%** of the overall coursework mark. It is due at **9pm** on **Thursday 15th February (week 3)**. As for every practical, you should arrive in the lab having prepared in advance, by studying this specification and reviewing relevant course material. If you don't do this you will waste time during your lab sessions.

Skills and Competencies

- developing robust software to manipulate data stored in files
- choosing an appropriate way to represent data in memory
- choosing appropriate classes and methods from a file API
- identifying and dealing with possible error conditions
- testing and debugging
- writing clear, tidy, consistent and understandable code

Setting Up

On this module we intend to use the automated checker `stacscheck` for most assignments. You should already be familiar with using it from CS1002 last semester. As you know, using the automated checker means that we have to impose some restrictions on naming of Java files, Java classes and where to put your files. For this practical, your program must have a `main` method in a class called `W03Practical` which is in a file `W03Practical.java` that is located inside your `src` directory in your `W03-Practical` directory.

The following steps are provided to help you set up your practical directories in such a way that they will be compatible with the automated checker. These steps are similar to those you carried out last semester on CS1002 at the start of practicals:

- Log in to a machine booted to Linux
- Check your email in case of any late announcements regarding the practical.
- Launch ***Terminal***.
- If you do not already have a *cs1003* directory for your CS1003 coursework, you can use the command `mkdir cs1003` to create it and then move to it using the command `cd cs1003`.
- Create a new directory called *W03-Practical* inside your *cs1003* directory and move to it by using the commands `mkdir W03-Practical` and `cd W03-Practical` for example.
- Create a new directory called *src* inside the *W03-Practical* directory to save your java source files using `mkdir src`. All your java files must be stored within this directory.
- Use appropriate `cd` commands (such as `cd src`) to navigate to the *src* directory within the *W03-Practical* directory.
- Create a new Java file *W03Practical.java* using e.g. `touch W03Practical.java`.
- Using *jEdit*, declare the class `W03Practical` and add an empty `main` method. Now save your program.
- Your program should compile using `javac *.java` at the command line, and run with `java W03Practical`. Please make sure this works, even though it won't do anything exciting yet.
- Create a new *LibreOffice Writer* document for your report and save it as *W03-Practical-Report*, inside your *W03-Practical* directory add an appropriate header at the top such as

  CS1003 W03 Practical 012345678 13 Feb 2018

- If you think you need more detailed instructions on how to use the Unix command line, have a look at last semester's CS1002 instructions for the Week 1 practical at: https://studres.cs.st-andrews.ac.uk/CS1002/Practicals/W01/W01-Practical-Introduction.pdf

Requirements

Write a Java program to process data from a file. For this practical, the data is stored in comma-separated-values (CSV) format. Your program needs to read in data from a given file, perform some processing, and create a new file containing the results. The provided data relates to sales of a UK based on-line store[1].

The format of the input data is illustrated below. The first line is the header line, and the remaining two lines are example lines for one of the data records. Each data record appears on a separate line in the file.

```
InvoiceNo,StockCode,Description,Quantity,InvoiceDate,UnitPrice,CustomerID,Country

536365,85123A,WHITE HANGING HEART T-LIGHT HOLDER,6,01/12/2010 08:26,2.55,17850,United Kingdom
536365,71053,WHITE METAL LANTERN,6,01/12/2010 08:26,3.39,17850,United Kingdom
```

As you can see, an invoice may comprise multiple stock items such that there can be multiple lines with the same invoice number (*InvoiceNo*) but for differed stock items (*StockCode*). The basic requirements do not require you to deal with rounding errors (using a simple `double` type for prices and `int` for quantities is fine) or cancellations of invoices (i.e. where the *InvoiceNo* attribute starts with letter 'C'). You can simply treat an invoice number such as C587654 as a string and print out the invoice details which will most likely result in a negative total for that invoice. You can also assume that data is ordered according to invoice number. For a more detailed description of the attributes, see:

https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/attribute-description.pdf

Various versions of the file, of differing lengths, can be downloaded from:

- https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/data/data-very-small.csv
- https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/data/data-small.csv
- https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/data/data-medium.csv
- https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W03/data/data-large.csv

When working from the lab (especially with the medium and large files), we recommend that you copy these files to the local hard drive (`/cs/scratch/<your-username>`) on your lab machine before proceeding. Running them locally will improve performance and reduce network traffic. Use the following command:

```
cp /cs/studres/CS1003/Practicals/W03/data/*.csv /cs/scratch/<your-username>/
```

where you should replace **`<your-username>`** with your own username.

The Task

You are to write a Java program which will read and process the data (indicated above) and write output to an output file. The overall operation of your program should be as follows:

- The program should be written to expect and use *two* command-line arguments (by making use of the `args` parameter to your `main` method in your `W03Practical` class).
  - `args[0]` should represent the path of the *input file*.
  - `args[1]` should represent the path to the *output file* where your program will write its output.
- If the command-line arguments are not supplied, then your program should print to `System.out` the line containing usage information as shown below
- Your program should process the *input file* and create an *output file* containing text representing:

[1] Obtained and adapted from: http://archive.ics.uci.edu/ml/datasets/Online+Retail

- the *invoice number*, *stock code*, *description*, *quantity*, and *unit price* for each invoice number and stock code
- at the end of all stock items on a particular invoice, the *total number of items* on the invoice and the *total price* for that invoice number
- a summary at the end of the output file consisting of the *invoice number* and *total price* for those invoices with the lowest and highest positive total price – that is, for this part of the exercise, you are not considering invoices that have a negative or zero-valued total price.
- All prices should be printed with 2 decimal places. You may use the `String.format`[2] method to achieve this.

Below are some examples of the expected output when executing your program from the `src` directory and for some different command-line arguments indicating the functionality that you should provide:

```
java W03Practical
Usage: java W03Practical <input_file> <output_file>

java W03Practical /cs/studres/CS1003/Practicals/W03/data/data-very-small.csv ../output.txt
```

The first invocation above does not pass any command-line arguments to the program and results in the required usage message being displayed in the terminal window. The second invocation does pass the expected command-line arguments to the program and as a result does not produce any console output to the terminal. Instead, the program writes the required information to the file `output.txt` in the `W03-Practical` directory as shown below (assuming the program is executed from the `src` directory that is inside your `W03-Practical` directory).

Content of file `output.txt` for second invocation above:

```
Invoice Number: 536365
Stock Code: 85123A
Description: WHITE HANGING HEART T-LIGHT HOLDER
Quantity: 6
Unit Price: 2.55
Invoice Number: 536365
Stock Code: 71053
Description: WHITE METAL LANTERN
Quantity: 6
Unit Price: 3.39
Number of items: 12
Total Price: 35.64

Invoice Number: 536366
Stock Code: 22633
Description: HAND WARMER UNION JACK
Quantity: 6
Unit Price: 1.85
Invoice Number: 536366
Stock Code: 22632
Description: HAND WARMER RED POLKA DOT
Quantity: 6
Unit Price: 1.85
Number of items: 12
Total Price: 22.20

Invoice Number: 536367
Stock Code: 84879
Description: ASSORTED COLOUR BIRD ORNAMENT
Quantity: 32
```

[2] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#format-java.lang.String-java.lang.Object...-

```
Unit Price: 1.69
Number of items: 32
Total Price: 54.08

Minimum priced Invoice Number: 536366 with 22.20
Maximum priced Invoice Number: 536367 with 54.08
```

The program should produce the output exactly as shown above and such output will be expected by the automated checker. Your program should deal gracefully with possible errors such as the input file being unavailable, or the file containing data in an unexpected format.

Running the Automated Checker

As usual, please use the automated checker to help test your attempt. The checker is invoked from the command line on the Linux Lab Machines in your `W03-Practical` folder:

```
stacscheck /cs/studres/CS1003/Practicals/W03/Tests
```

If all goes well, you should see something similar to below

```
Testing CS1003 Week 3 Practical
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - basic/build : pass
* COMPARISON TEST - basic/Test01_no_arguments/progRun-expected.out : pass
* TEST - basic/Test02_data-very-small/test : pass
* TEST - basic/Test03_data-small/test : pass
* TEST - basic/Test04_data-medium/test : pass
* INFO - basic/TestQ_CheckStyle/infoCheckStyle : pass
--- submission output ---
Starting audit...
Audit done.
---
6 out of 6 tests passed
```

If it is not going so well, here are some things to consider

- If the automated checker cannot be started, then you have most likely mistyped the commands to invoke the checker shown above.
- If the automated checker runs but fails at the build stage, then no other tests can be conducted, so you will have to fix this issue first. Likely reasons for the build failure include: executing the checker from some directory other than your `W03-Practical` directory, specifying an incorrect path to the tests, and your program containing errors such that it cannot be compiled.
- If *Test01* fails, it is because your program is not producing the expected output when no command-line parameters are supplied on execution. Remember to check the length of the `args` array and print a suitable message if it too small.
- If some or all of *Test02 – Test04* fail, then your program is most likely not writing the expected output to the file that is specified on the command line when executing your program. Make sure that you use the `args` array elements correctly in your code.
    - Each of these tests will run your program with specified input files and output files. If you see a line such as:

      ```
      --- submission stderr output ---
      diff: prog-output.txt: No such file or directory
      ```

      your program has not written to the output file specified as a command-line parameter.

- The tests use the Unix `diff` command to print a message describing how a line (or set of lines) in your output should be changed to match the required output. It uses the key symbols **`c`** (for change), **`a`** (for add), and **`d`** (for delete) and also prefixes output from your program with '<' and the expected output with '>'. For example, the following output from the checker:

```
--- submission output ---
1c1
< InvoiceNo: 536365
---
> Invoice Number: 536365
```

  means that your program's output on line 1 needs to be changed to match with line 1 in the expected output, in this case because your program output "InvoiceNo" rather than "Invoice Number".

- Failing these tests could be down to logical errors. Check the logic and debug your program to spot errors and fix them. Try running the automated checker again to verify your fix.

- Please make sure you comment out all your debugging print statements before running the automated checker.

- The final test *TestQ CheckStyle* runs a program called Checkstyle over your source code and uses a style adapted from our St Andrews coding style (informally known as the *Kirby Style*). You may receive a lot of output from the style checker for your program. In order to address these, you can look at the published guide at

  https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/programming-style.html

  The style recommends the use of spaces as opposed to TABs for indentation. Don't worry too much about this.

If nothing is working and you don't know why, please don't suffer in silence, ask one of the demonstrators in the lab. The automated checking system will help you with some basic testing of your program, and you can document your success (or otherwise) with it in the Testing section of your report.

Testing

You are encouraged to create and document your own tests to test how your program deals gracefully with possible errors such as the input file being unavailable, or the file containing data in an unexpected format. To prove that you have tested the above scenarios, paste terminal output from each of your tests into your report in the Testing section. Be clear what each test demonstrates.

Deliverables

Hand in via MMS to the *Week 3 Practical* slot, as single `.zip` file containing your entire `W03-Practical` directory which should contain:

- All your Java source files as specified above
- A report containing the usual sections for *Overview*, *Design & Implementation*, *Testing*, *Examples* (example program runs), *Evaluation* (against requirements), *Conclusions* (your achievements, difficulties and solutions, and what you would have done given more time). You can use any software you like to write your report, but your submitted version must be in PDF format.

Extensions

Please note: It is key that you focus on getting a very good solution to the basic requirements prior to touching any of the extension activities. A weak solution with an attempt at extensions is still a weak solution. Once you have a very good solution to the basic requirements, you may wish to experiment further and could try any or all of the following:

- Alter your program to take additional command-line parameters which act as switches to e.g.:
  - Deal with cancellations indicated by the *InvoiceNo* attribute starting with letter 'C'.
  - Output statistics such as total price and number of invoices, per calendar month or year.
  - Display some aspect of the data in a graphical format.
  - Make your output file into a web page.
- Alter your program to handle data containing commas within CSV fields (escaped using double quotes).

Don't forget to document your extension work in your report.

Marking

Your submission will be marked using the standard mark descriptors in the School Student Handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

However, more specifically, for this practical:

- 1-6: Very little evidence of work, software does not compile or run, or crashes before doing any useful work. You should seek help from your tutor.
- 7-10: A decent attempt which gets part of the way toward a solution, but has serious problems such as not compiling (at the low end), or lacking in robustness and maybe sometimes crashing during execution (at the high end).
- 11-13: A solution which is mostly correct and goes some way towards fulfilling basic requirements, but has issues such as not always printing out the correct values.
- 14-15: A correct solution accompanied by a good report, but which can be improved in terms of code quality, for example: poor method/OO structure, lack of comments, or lack of reuse.
- 16-17: A very good, correct solution to the main problem containing clear and well-structured code achieving all required basic functionality, demonstrating very good design and code quality, good method and class decomposition, elegant implementation of methods with reuse where appropriate, and accompanied by a well-written report.
- 18-19: As above, but implementing at least one or more extensions, accompanied by an excellent report. However, as mentioned above, it is key that you focus on getting a very good solution to the basic requirements prior to touching any of the extension activities. A weak solution with an attempt at extensions is still a weak solution.
- 20: As above, but with multiple extensions, outstanding code decomposition and design, and accompanied by an exceptional report.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

http://www.st-andrews.ac.uk/students/rules/academicpractice/

Hints

Rather than trying to develop a complete solution in one go, it may be easier to produce a series of versions, starting with something very simple. Here is one possible sequence:

1. Write your program to open the very small version of the input data file and prints out its contents to the console, line by line, using calls to `System.out.println()`.
2. Refine the previous version so that it writes the contents of the input data file to a new output file.
3. Refine the previous version so that it generates a file containing each required separate item (*invoice number*, *stock code*, *description*, etc.) from the input data file on a separate line as indicated in the sample output above.
4. The `String.split()` method may be useful for splitting up a line of text from the input file into individual strings.
5. Refine the previous version so that it keeps a running total price and total number of items for an invoice (using a Java `double` type) and outputs the total price for an invoice prior to moving on to the next invoice number as indicated in the sample output above.
6. Refine the previous version so that it keeps track of the invoice number and price for the minimum and maximum priced invoice and any other necessary information, as it reads in the data file and get it output the summary information at the end of the output file as shown in the sample output above.
7. Test your program with missing and badly formatted input files, and add exception handling where necessary to produce informative error messages.

You may wish to make a new class at each stage, so that you can easily go back to a previous stage if you run into problems.

Prior to submitting, please make sure that your most recent version of the program can be compiled and run using your `W03Practical` class.