University of London International Programmes

Computing and Information Systems/Creative Computing

CO2220 Graphical Object-Oriented and Internet programming in Java

Coursework assignment 2 2017–18

Introduction

This is Coursework assignment 2 (of two coursework assignments) for 2017–18.

Part A asks that you demonstrate an understanding of constructors, the `StringBuilder` class, parsing with *String.split(),* writing to and reading from a file, and serialisation.

Part B looks at a client/server system utilising a GUI for user interaction, and how choice of the right data structure can improve performance.

Electronic files you should have:

Part A

- *Admissions.java*
- *AdmissionsFileManager.java*
- *FeeType.java*
- *Gender.java*
- *OfferStatus.java*
- *ProspectiveStudent.java*
- *QualificationsType.java*
- *student_admissions_file.ser*
- *student_admissions_file.txt*

Part B

- *Anagram.java*
- *AnagramClient.java*
- *AnagramClientGUI.java*
- *AnagramFactory.java*
- *AnagramServer.java*
- *AnagramServerEngine.java*
- *Dictionary.java*
- *History.java*
- *Permutation.java*
- *PermutationFactory.java*
- *ServerUtils.java*
- *words.small*

What you should hand in: very important

There is one mark allocated for handing in uncompressed files – that is, students who hand in zipped or .tar files or any other form of compressed files can only score 49/50 marks.

There is one mark allocated for handing in just the .java files asked for, without putting them in a directory; students who upload their files in directories can only achieve 49/50 marks.

At the end of each section there is a list of files to be handed in – **please note the hand-in requirements supersede the generic University of London instructions**. Please make sure that you give in **electronic versions** of your .java files since you cannot gain any marks without handing in the .**java** files asked for. Class files are **not** needed, and any student giving in only a class file will not receive any marks for that part of the coursework assignment, **so please be careful about what you upload as you could fail if you submit incorrectly**.

Programs that do not compile will not receive any marks.

The examiners will compile and run your Java programs; students who hand in files containing Java classes that cannot be compiled (e.g. PDFs) will not be given any marks for that part of the assignment.

Please put your name and student number as a comment at the top of each .java file that you hand in.

You are asked to amend some of the classes given, and hand in your amended files. Please do not change the names of the files you submit; in particular, make sure there is no conflict between your file name and class names. Remember that if you give a file a name that is different from the name of a public class that it contains (e.g. *cwk2-partA.java* file name versus *AdmissionsFileManager* class name) your program will not compile because of the conflict between the file name and the class name, and you will lose marks.

Instructions

1.0 What version of Java to use

This coursework assignment uses some features introduced in Java 7, and a method introduced in Java 8. Please use the most recent version of Java, Java 8.

1.1 Java 7 new features

1.1.1 The diamond operator

The programs you have been given for parts A and B use **the diamond operator**, meaning that the type of an `ArrayList` can be inferred by the JVM. For example, this declaration:

```
    ArrayList<String> details = new ArrayList<String>();
```

can be rewritten as:

```
    ArrayList<String> details = new ArrayList<>();
```

From the left side of the declaration, the JVM infers that the `ArrayList` is of type `String`.

1.1.2 The `Files` class

`Files`, introduced in Java 7, is a class with static methods that operate on files and related data structures; it includes the method ***readAllLines(Path, Charset)***. This method reads all lines from a file, recognises standard line endings, and closes the file after use. The *Dictionary* class in part B uses the Java 8 method ***Files.readAllLines(Path)*** that takes only a `Path` parameter. Bytes from the file are decoded into characters using the UTF-8 `Charset`.

For more information on the `Files` class, see the API:
https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html

1.2 Other features

1.2.1 Platform-specific new lines

Because Windows and Linux have different new line symbols the files you have been given use code for new lines that will work as expected on any platform. *System.lineSeparator()* is used, for example, in the *toString()* method of *ProspectiveStudent. System.lineSeparator()* will return the appropriate new line character for the system it is used on. See the API for more information on the `System` class: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html

When using *String.format(),* such as in *AnagramServerEngine*, "%n" is used for a new line, as *n* gives the platform-specific line separator. See the API for the Formatter class for more information: https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html

1.2.2 `List` interface

You will note that in the files you have been given for part B, `ArrayLists` are declared to be of type `List`, as are methods that return an `ArrayList`. `List` is an interface: http://docs.oracle.com/javase/7/docs/api/java/util/List.html

Explanation below, modified from *Effective Java*, 2nd edition by Joshua Bloch

> You should use interfaces rather than classes as parameter types. More generally, you should favour the use of interfaces rather than classes to refer to objects. If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.

Get into the habit of typing this:

```
// Good - uses interface as type
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

rather than this:

```
// Bad - uses class as type!
ArrayList<Subscriber> subscribers = new ArrayList<Subscriber>();
```

If you get into the habit of using interfaces as types, your program will be much more flexible. If you decide that you want to switch implementations, all you have to do is change the class name in the constructor.

For example, the first declaration could be changed to read:

```
List<Subscriber> subscribers = new LinkedList<Subscriber>();
```

and then all of the surrounding code would continue to work. The surrounding code was unaware of the old implementation type, so it would be oblivious to the change.

1.2.3 The `enum` type

The *ProspectiveStudent* class uses the `enum` type for some of its fields. These fields (*FeeType, Gender, OfferStatus, QualificationsType*) could have been Strings, in which case anything could be entered. Using the `enum` type allows us to restrict the user to only a few very specific entries that are appropriate for the field. For example, *Gender* types will only accept FEMALE, MALE and OTHER. For more information see:
https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Part A

Consider the classes that you have been given. Currently the *Admissions* class will not compile as it tries to use methods in the *AdmissionsFileManager* class that do not exist, and a 10-argument constructor for the *ProspectiveStudent* class that has not been written. Once you have successfully answered the questions below, the *Admissions* class should compile and run. You may test the methods of the *AdmissionsFileManager* class by running the class, since the class has a main method with test statements. This will also test that you have answered Question 1 correctly. Do not change the main method, and also do not change any of the four test methods that the main method runs.

1. The *ProspectiveStudent* class has one constructor that sets the fields of the class to default values for testing purposes. It does not have a constructor that allows the user to set the value of its fields. Write a second constructor that will allow the user to set every field (*i.e.* instance variable) of the class. **[6 marks]**

2. In the *AdmissionsFileManager* class, the *read()* method is intended to return an *ArrayList<ProspectiveStudent>*, containing *ProspectiveStudent* objects made from the data in the *student_admissions_file.txt* file.

   The method reads in 10 lines at a time from the text file, then calls another method, *parseProspectiveStudent(ArrayList<String>)* in order to parse those 10 lines to a *ProspectiveStudent* object. However, the *parseProspectiveStudent(ArrayList<String>)* is not complete, so the *read()* method does not work as intended.

   Complete the *parseProspectiveStudent(ArrayList<String>)* so that the *read()* method works as intended. **[6 marks]**

3. You are given the heading of the *write(ArrayList<ProspectiveStudent>, String)* method. Complete the method. In your answer make sure that the method does what the comment written above the heading says it will do. **[6 marks]**

4. You are given the heading of the *deserialize(String)* method. Complete the method. In your answer make sure that the method does what the comment written above the heading says it will do. **[6 marks]**

5. You are given the heading of the *serialize(ArrayList<ProspectiveStudent>, String)* method. Complete the method. In your answer make sure that the method does what the comment written above the heading says it will do. **[6 marks]**

6. Write an appropriate constructor for the *AdmissionsFileManager* class. **[6 marks]**

Reading for part A
Note that in the list below, the 'subject guide' refers to Volume 2, and *HFJ* refers to *Head First Java*.

- HFJ pages 240–49 (constructors)
- HFJ pages 301–06 (working with dates)
- Subject guide Chapter 2, Sections 2.2, 2.11 and *HFJ* pp.275–78 (static utility classes)
- Subject guide Chapter 3, Sections 3.2, 3.3, 3.5, 3.6 and *HFJ* pp.319–26; and 329–33 (handling exceptions)

- Subject guide Chapter 5, Sections 5.3 and *HFJ* 447, 452–54 and 458–9 (files, parsing with *String.split(),* streams including *BufferedReader* and *BufferedWriter*)
- Subject guide Chapter 6, Sections 6.2, 6.3, 6.4 and *HFJ* 429–38; 441–3 (serialization)

Deliverable for Part A
An electronic copy of your revised classes:

- *AdmissionsFileManager.java*
- *ProspectiveStudent.java*

Reminder:
Please put your name and student number as a comment at the top of your Java files.

PART B

Testing the ClientGUI
You should compile and run the *AnagramClientGUI.java* file. To do this you will first need to compile and start *AnagramServer.java*. You can run both programs from the shell (cmd.exe) but will need to open the shell twice; once to get the *AnagramServer* started; and once to run the *AnagramClientGUI*. Note that the *AnagramClientGUI* has been hard-coded to connect to a local host.

When you start the *AnagramClientGUI* class you should see a simple GUI with a *JButton,* a scrollable *JTextArea* and a *JTextField*. When you click on the 'Connect' button, if you see the message 'ERROR: Connection refused: connect', and you have the *AnagramServer* class running, it may be that you need to tell your firewall to allow Java through it. If this is the case you will have to restart the *AnagramServer* and the *AnagramClientGUI* once you have adjusted your firewall. Whenever you disconnect from the *AnagramClientGUI* you will have to restart the *AnagramServer* from the shell (cmd.exe) as the server is single-threaded and dies once the client has disconnected.

What you should see when you run the *AnagramClientGUI* is the following, on the *JTextArea*:

```
Welcome IP address '127.0.0.1' to the Anagram Server. Available
commands:
ANAGRAM <word>  show all anagrams of the word <word>
PERMUTE <word>  show all permutations of the word <word>
SHOW HELP       show this help
```

The PERMUTE command calls the permutation method, which will make all possible permutations of a word. A permutation is a rearrangement of all the objects in a collection. Suppose we have the word MAT, all possible permutations are:

MAT
MTA
ATM
AMT
TAM
TMA

If you try the command ‘permute mat’ on the GUI, you will get five permutations, as the GUI will not count ‘mat’ as a permutation of itself.

Now suppose our word is TAT, all possible permutations are:

TA**T**
T**T**A
AT**T**
A**T**T
TAT
TTA

In the above, the second ‘t’ is in bold. As far as theory is concerned, a 3 letter word has 6 permutations (incuding itself).

If you try ‘permute tat’ you will only see `att tta` as output, as the system will not return duplicate permutations.

An anagram is defined to be '**a word, phrase, or name formed by rearranging the letters of another**'. In the system you have been given, we are only considering words and names, not phrases, as anagrams of a word.

Try 'permutation mitre' followed by 'anagram mitre'. You may have to scroll across to see all of the permutations. You should find that from the list of unique permutations, the anagram command returns only those that are words in their own right.

When testing the GUI you should note that the History class allows you to use the arrow keys to retrieve your old commands to save typing.

Please complete the following tasks:

1. Explain why the *Dictionary*, *PermutationFactory* and *AnagramFactory* classes all implement the Singleton design pattern.

   Give your answer by writing a comment at the top of each of the above three classes, identifying: (1) **how** the pattern is being implemented by the class; and (2) **why** the pattern is being implemented in that particular class. **[5 marks]**

2. Anagrams and permutations are listed as one continuous line on the GUI, such that the user may have to scroll across to see them all. Can you alter the system so that they are listed one to a line? For example, instead of `att tta` output as the permutations of 'tat', the output would be:

   ```
   att
   tta
   ```

   After you have implemented your change, the user may now have to scroll down to see all permutations/anagrams.

   Make sure to use platform independent line separators, as described in section 1.2.1 above. **[2 marks]**

3. On the machine used to test the GUI, looking for anagrams of the 10 letter word 'manchester' took 4 minutes and 15 seconds. Longer words were not tested as they were expected to be even slower. Your system may be faster or slower but you should still find that it takes minutes to find the anagrams of a 10 character `String`.

   It is possible to use a different Java data structure in one of the classes *Dictionary*, *PermutationFactory* or *AnagramFactory* such that finding anagrams is much faster. On the computer used to test the GUI, finding anagrams of 'manchester' (or more exactly, finding that there are not any anagrams of 'manchester') took 11 seconds after the improvement was made.

Make the improvement and, in a comment at the top of the class you have made it in, explain why the data structure you have chosen is an improvement in terms of speed over the original data structure. In a second comment, quantify the improvement you have made in terms of time taken to return anagrams of a particular word. You may use ‘manchester’, some other 10 letter word, or a longer word as your test data.

The data structure used to make the improvement is one already defined by Java, so you should be looking in the API for potential solutions. **[5 marks]**

Reading for Part B

- Chapter 7 of Volume 2 of the CO2220 subject guide, pages 63–66 (clients and servers).
- *Head First Java* pages 473–85 (clients and servers)
- https://en.wikipedia.org/wiki/Singleton_pattern
- *Head First Java* pages 294–300 (formatting output)
- https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html (documentation of the `Formatter` class including lists of arguments for formatting output)
- The Java API

Deliverables for Part B

Please submit an electronic copy of the following:

- *The .java file where you have implemented your answer to Question 2. (Note this should be one of the files that you were given with the coursework assignment).*
- *AnagramFactory.java*
- *PermutationFactory.java*
- *Dictionary.java*

Reminder:
Please put your name and student number as a comment at the top of your Java files.

MARKS FOR CO2220 COURSEWORK ASSIGNMENT 2

The marks for each section of Coursework assignment 2 are clearly displayed against each question and add up to 48. There are another two marks available for: giving in uncompressed .java files; and giving in files that are not contained in a directory. This amounts to 50 marks altogether. There are another 50 marks available from CO2220 Coursework assignment 1.

Total marks for Part A	[36 marks]
Total marks for Part B	[12 marks]
Mark for giving in uncompressed files	[1 mark]
Mark for giving in standalone files; namely, files **not** enclosed in a directory	[1 mark]
Total marks for Coursework assignment 2	**[50 marks]**

[END OF COURSEWORK ASSIGNMENT 2]