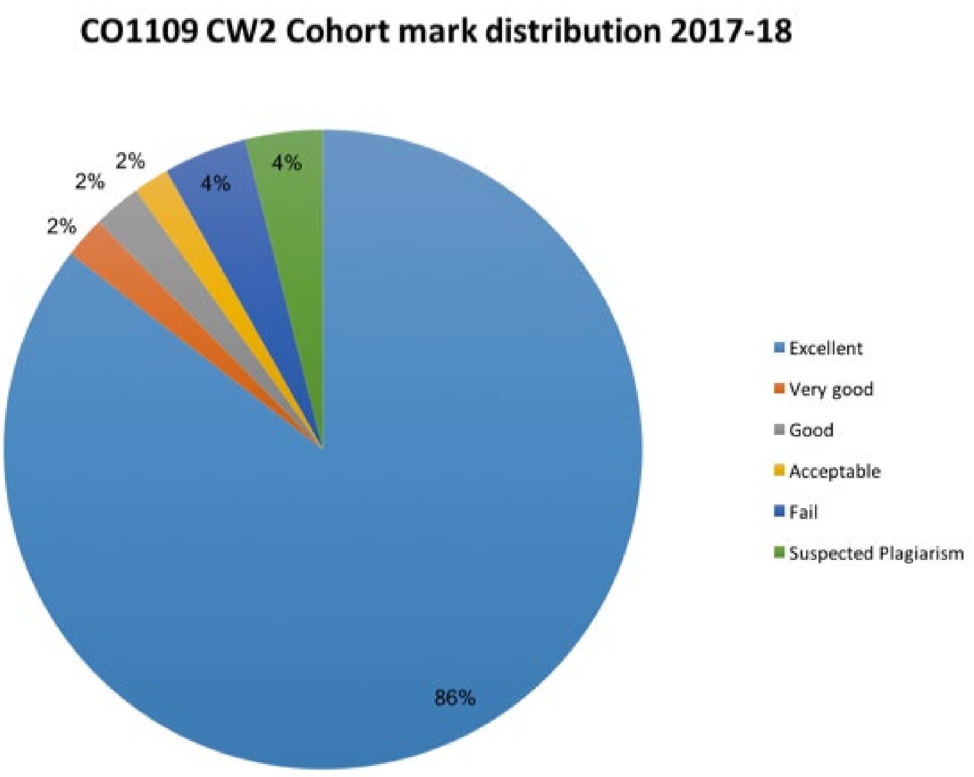
Coursework commentary 2017–2018

CO1109 Introduction to Java and object-oriented programming

Coursework assignment 2

On the whole, this was attempted well, with the majority of students gaining excellent marks.



Part A

Model answers

Please note that the following model answer is given with this commentary:

- *DictionaryMethods.java*

The following files are also given so that students can run and test the model answer:

- *smallDictionary.txt*
- *vSmallDictionary.txt*

The assignment

Students were given the files:

- *DictionaryMethods.java*
- *smallDictionary.txt*
- *vSmallDictionary.txt*

The text files were to be used by the *DictionaryMethods* class. When the *DictionaryMethods* class started an `ArrayList<String>` variable called *dictionary* was made by reading in the words from the *smallDictionary.txt* file given to students. Students were asked to amend some of the methods in the *DictionaryMethods* class and to write four new methods.

Question 1

The *findWord()* method searched for a word in the *dictionary* `ArrayList`, telling the user if their word was found with the message "Word entered is in the dictionary", but giving no output if the search `String` was not found. Students were asked to add the search `String` to the 'found' message, as well as adding a 'not found' message. Almost all students achieved full credit for this question, with a few losing marks because they did not include the search `String` in the found message, and/or in the 'not found' message.

Question 2

Most students gained full credit for this question. Students were asked to test the *display()* method, and to note that it was quite slow. They were asked to rewrite the method to be faster. The method took a second or two to make a `String` by concatenating all the words from the *dictionary*, using this statement in a `for` loop:

```
text += dictionary.get(i) + System.lineSeparator();
```

`String`s are immutable, meaning that once made they cannot be changed. Hence appending `String` *x* to `String` *y* will mean that a new `String` *y* object is made, and the former `String` *y* object has no variable referencing it. The orphan `String` still exists in memory until the JVM disposes of it and frees the memory space. Since there are 45,000 words in the dictionary that is potentially quite a lot of wasted memory. This is probably not a problem with such a small program, but something to bear in mind, because the programs we write will not always be small.

Changing the statement in the `for` loop to: `System.out.println(dictionary.get(i))`

meant that the output started immediately, although it was not necessarily any faster to display the entire dictionary. Using a `StringBuffer` would also be a good solution, since `StringBuffers` are not immutable, and are often used in loops with a lot of `String` concatenation to save memory.

```
private void display() {
     StringBuffer text  = new StringBuffer();
     for (int i = 0; i < dictionary.size(); i++) {
        text.append(dictionary.get(i)+System.
lineSeparator());
     }
     System.out.println(text.toString());
}
```

Only a very few students gave an answer using `StringBuffer`, and it was not necessary for full credit, but was an excellent solution.

Another solution was simply to print the entire dictionary at once with `System.out.println(dictionary);`. This will use the *toString()* method for an `ArrayList`, which will print the contents of the `ArrayList` enclosed in square brackets, with each item separated by a comma and a space. It looks ugly, but is very fast. Another very fast solution, given by just one student, was to use the `String.join()` method (a new `String` method since Java 8) to make a `String`, and then print it. The student who gave this solution noted in a comment that they had searched the internet for a solution, and had found one on StackOverflow, giving the link. This is absolutely fine, StackOverflow is a good place to find developers addressing questions from less experienced programmers. Note that it would not be acceptable to ask a coursework related question on StackOverflow.

Question 3

Students were asked to write four new methods that searched the *dictionary* `ArrayList` looking for matches with partial words, or that searched for words of a certain length. They were asked to use the following names for their methods:

- *contains()*
- *startsWith()*
- *endsWith()*
- *thisLong()*

The Java `String` API was given as reading for this question. Students were expected to consult the API for `String` methods that would help them to answer the question, and most did, using the `String` methods *contains(), startsWith()* and *endsWith()*. The *thisLong()* method needed to use the `String.length()` method in order to find all `Strings` of a certain length. Most students answered this question well, using `String` methods as expected. The searching was normally implemented successfully, with some common errors in the messages to the user.

Mistakes with searching

There were a few mistakes with searching, but they tended to be quite individual, such as: ending the search after finding the first match; only allowing searching if the word is in the dictionary (using the `ArrayList` method `contains()` to check this); forgetting that two of the methods are looking for partial matches, so could still return valid output even if the search `String` is not in the dictionary as a complete word; or ending the `for` loop expression with a semi-colon, meaning that there are no statements included in the loop, and hence effectively no search loop.

Common errors in the messages to the user

- The messages to the user did not contain the search `String`. The example output given with the coursework instructions included the search item in 'not found' messages, and students were expected to take the hint. This mistake was very common.
- There was no message to the user when the search item was not found.
- There was no message to the user when the search item was found.
- With every iteration of the loop, if the search item did not match the current word from the dictionary, then the 'not found' message was printed. Either poor testing or the student assumed that the user would want to see the same message thousands of times, and would enjoy hunting among these messages for the 'found' output.
- The 'not found' message is printed at the end of each method even if a search item was found, either because no effort has been made to implement not printing the message when a match is found, or due to a logical error.

Mistake: The *nextInt()* method

A minority of students (17% of submissions) made a new `Scanner` object in the *thisLong()* method, in order to use the `Scanner` method `nextInt()`. The problem with the `nextInt()` method is that it leaves a hanging new line, hence when the program returns to the main user interaction loop the newline is read in as the user's entry, causing an 'Unknown option' message. The solution to this, seen only once, is to add `in.nextLine()` after `in.nextInt()` in order to eat the new line. Note that making a new `Scanner` object was not necessary, the `Scanner` made in the constructor could have been used.

Mistake: Closing *System.in*

Once a `Scanner` is closed, the input source that it uses is also closed. For example the *readDictionary()* method makes a `Scanner` object with a `FileReader` as the input source, hence closing the `Scanner` object also closes the `FileReader`. It is important to note that once `System.in` is closed, it cannot be reopened. All of the students making a `Scanner` in the *thisLong()* method gave their `Scanner` objects `System.in` as the input source, and some of them closed their `Scanner` object at the end of the method. Once control returned to the main loop the `Scanner.nextLine()` method in *getUserInput()* would throw a `NoSuchElementException` and the program would end. This was because the `Scanner` object made in the constructor used `System.in` as its input source, so with `System.in` closed, the `Scanner` object could no longer read any input.

String.subString() and code reuse

There was a small minority who did not use the existing `String` methods, and instead wrote their own. Quite challenging to do, and against the spirit of Java, which is very much about code reuse. Students wrote fairly complicated logic, mostly using the `String.substring()` method, which sometimes worked, and sometimes did not. Something very surprising was the minority who, instead of searching the *dictionary* `ArrayList`, instead wrote methods that read in the words from the text file, and searched them. It seemed strange to be doing work that had already been done when the *dictionary* variable was made. It was also a waste of system resources. As our programs and systems get bigger, we need to think more about how we are using system resources, so it is good to get into the habit of not repeating work early.

Question 4

Students were asked to add an extra menu choice to run the *changeDictionary()* method as option 7, and to make the quit choice option 8. Very few problems were seen with this question, and most students gained full credit. A minority made the following errors:

- The student added *changeDictionary()* to the switch, but option 8 does not end the program because the *validateUserInput()* method has not been amended such that it can return an 8 when the user enters 8.
- The student added *changeDictionary()* to the `switch`, has not amended *startLoop()* so that user input of 8 makes the *proceed* `boolean` false and thus ends the loop.
- The student added *changeDictionary()* to the switch, but has not made the other changes necessary which include that *validateUserInput()* should be able to return an 8, and that in *startLoop()* user input of 8 should make the *proceed* `boolean` false.

Question 5

Students were asked to test and correct the *changeDictionary()* method. Students were expected to find that when the method was given a new file to replace the contents of the *dictionary* variable, what actually happened was the new content was appended to the old. Students were expected to look in the `ArrayList` API in order to find an appropriate method to empty the `ArrayList`. The link to the Java `ArrayList` API was given in the reading for the assignment. In the API students would have found the method *clear()* that "Removes all of the elements from this list"[1]. All that was necessary was to add the statement `dictionary.clear();` to the *changeDictionary()* method. Many students did this, but a large minority instead remade the *dictionary* variable, with the statement `dictionary = new ArrayList<String>();` usually in the *readDictionary()* method. Using the `ArrayList.clear()` method would have been more readable and a better use of memory resources.

One issue seen with a few answers was the use of the `ArrayList` method `removeAll()` to empty the *dictionary,* which was very slow; on the examiner's laptop it took 8 minutes for the `removeAll()` method to complete its task of removing 45,000 words from the *dictionary* `ArrayList`. The method was called with a statement such as `dictionary.removeAll(dictionary);` Note that the method is intended to remove from the current object, all elements contained in a separate list. The method is not really intended to be used to clear a list by removing from an `ArrayList` all of the elements that it has in common with itself.

Question 6

Students were asked to amend the class so that an exception would not be thrown by the *readDictionary()* method, if the user were to enter a file name when asked for one by the *changeDictionary()* method, that did not exist. Instead the program should enter a loop asking the user again for a file name, until a valid name was entered.

The *changeDictionary()* method as given to students, passed the file name given by the user to the *readDictionary()* method, that already had a try/catch block. In the try block a `Scanner` was made, with a `FileReader` for the input stream. The catch block told the user that the program was being 'aborted' if the file name was not valid, and after this a `NullPointerException` was thrown and the run of the class ended.

Note that the try/catch was handling the potential `FileNotFoundException`. If the catch block was invoked, then the `Scanner` would not have been made successfully, and would be `null`. The reason a `NullPointerException` was thrown can be found in the `Scanner's` API, which notes:

Unless otherwise mentioned, passing a `null` parameter into any method of a `Scanner` will cause a `NullPointerException` to be thrown.

Hence when the expression `while(in.hasNextLine())` invokes the `Scanner's hasNextLine()` method, an exception will be thrown if the `Scanner` is `null`.

The model answer is simple

Some quite complex answers were seen, when in fact this question could be answered by: (1) adding a statement to the end of the *changeDictionary()* method to tell the user that the dictionary has been changed; (2) enclosing the try/catch in the *readDictionary()* method in a `while` loop; (3) changing the message output by the catch block to ask for new input from the user; (4) read in the new user input in the catch block. This is the solution implemented in the model answer and it means that the try block will try to make a `Scanner` object and open the file, if this fails the catch block will tell the user that their file name was invalid, ask the user to enter another file name and read in the new file name. The `while` loop will repeat these actions until a valid file name is given by the user.

With this solution the method can guarantee a valid file name, and that the `Scanner` will not be null when control moves to the loop that reads the file and makes the *dictionary* `ArrayList`. It may be that some or all of the 21% of students who implemented a solution that caught the `NullPointerException` did not understand that if the *readDictionary()* method was prevented from operating with a null `Scanner`, then no `NullPointerException` would be thrown.

One thing to note about the above solution, is that the message from the catch block in the *readDictionary()* method must work when the object is made (*i.e.* when the method is called by the constructor) and when the dictionary is changed. For example, having the try block print the message to say that the dictionary had been changed, would not make sense to the user when the program started. A number of students deleted the error message from the try block completely, meaning that no error message would be seen on start up if the dictionary file was missing.

Recursion, and methods calling each other repeatedly

Some answers were seen that were very much more complicated than the model answer. Some were seen where the *changeDictionary()* and *readDictionary()* methods passed control between themselves until a valid file name was entered, and others that used recursion for their file name validation loop. Both implementations can be very hard to read, and can lead to multiple copies running of the same method, which can compromise the user's experience and drain the memory. In addition complicated implementations need extensive testing to cover all possible scenarios before one can be certain that they work in every circumstance. For example, one implementation seen was that the *changeDictionary()* method would call *readDictionary()*, and if the file name was invalid then the *readDictionary()* method would call the *changeDictionary()* method. The *changeDictionary()* method would ask the user again for a valid file name, then it would call *readDictionary()* with it, opening a second copy of the method. If the second file name was valid, then the second copy of *readDictionary()* would complete successfully, after which control would return to the first copy of *readDictionary()*. The first copy would then have completed one iteration of its `while` loop asking for a valid file name, with an invalid file name, so the `while` loop in the first copy of *readDictionary()* would iterate again. Then once again *changeDictionary()* would be called, and, given valid input, would call another copy of *readDictionary()* which would complete successfully. Then control would return to the original copy of *readDictionary()* which would have an invalid file name, and so the `while` loop would iterate again and this would go on and on.

Implementing the loop to request a valid file name in the *changeDictionary()* method with recursion meant that the method would call itself again if given invalid input. Most of these students were using the `NullPointerException` thrown by *readDictionary()* if given an invalid file name, as a reason to call *changeDictionary()* recursively. One very common side effect of using recursion seen in testing student programs was that if the user entered an invalid file name at least once, then the loop to request a valid file name would appear to work, but when the user chose to quit, the program would throw an exception. This was because as the *changeDictionary()* method backed out of the recursion, all copies of the *changeDictionary()* method called by the recursion had to complete before the program could end, and would run statements that would call the *readDictionary()* method with an invalid file name.

Restricting valid input

Some students lost credit by rejecting any file name that was not *vSmallDictionary.txt*, which of course, would mean that once the input file was changed to *vSmallDictionary.txt*, it would not be possible to change it again. Others enforced that the file name could only be *vSmallDictionary.txt* or *smallDictionary.txt*, but this was penalised in the marking as the user should be able to enter any valid file name.

Part B

Model answers

Please note that the following model answers are given with this commentary:

- *Finalist.java*
- *ProcessDegreeMarks.java*

The following files are given so that students can run and test the model answers:

- *FinalistComparator.java*
- *finalMark.txt*

The assignment

Students were given the files:

- *Finalist.java*
- *FinalistComparator.java*
- *ProcessDegreeMarks.java*
- *finalMark.txt*

The *Finalist* class has fields for a student ID, a degree mark, a class of degree based on the degree mark, and borderline status (true or false). The *finalMark.txt* file held data to make 8 *Finalist* objects. The *ProcessDegreeMark* class had a method, *FinalistsInList()*, to read the text file and make an `ArrayList`, called *finalists,* of *Finalist* objects. The *ProcessDegreeMark* class had some static methods for manipulating `ArrayLists` of *Finalist* objects, and the *FinalistsInList()* method was written in order to make an `ArrayList` to test these methods with. Students were told not to change the main method, which contained test statements, in fact 3 did, but out of 333 submissions that is less than 1%.

The *FinalistComparator* class was used by the *finalistsToFile()* method to sort the `ArrayList` in order of degree mark. Students were not expected to make any changes to the *FinalistComparator*, and none did. Comparators are used for sorting complex objects, as they allow the developer to specify which field or fields of the object to sort by.

Students were asked to make some changes to the *Finalist* class, and to amend some of the methods in the *ProcessDegreeMark* class as well as writing new ones.

Question 1

The *Finalist* object has four instance variables, or fields. The constructor of the *Finalist* class takes the `String` *id* and `double` *degreeMark* as parameters, and then calls methods to calculate values for the `boolean` *borderline* and `double` *degreeClass* variables. The four instance variables have private access, meaning they can only be accessed from within their own class; this is standard Java practice.

Access to instance variables is normally granted through public instance methods called setters and getters, in order to protect the fields from unexpected updates. Getters are so called because they get the value of the instance variable and return it; setters change the value of the instance variable. There is a Java naming convention for these methods, the method name starts with set or get as appropriate, and ends with the name of the variable it is setting or getting, but with the variable name starting with an upper case letter. The *Finalist* class given to students had a getter for the *id* variable, and following naming convention this was called *getId()*. Students were asked to write getters for the other instance variables, and told that the *Finalist* class needed no setters (despite this a few students wrote setters anyway).

It was good to see that most students followed the Java naming convention for getters. It is important to get into the habit of doing so, since it helps developers to know how to use any class that you write. Others will expect your class to have getters, and will assume that they know what your setter methods are called, so mistakes with names could be problematic for anyone trying to use your class. Having said that, some called their getter for the *borderline* variable *isBorderline()* rather than *getBorderline()*, and this was acceptable as the convention allows that methods that return a `boolean` may be called by 'is' rather than 'get'.

Most students gained full credit for this question, however a number of students made minor mistakes in the names of their methods, such as *getDegreeeClass()* (note three 'e's in Degreee). Since it is important to follow the naming convention exactly, these students lost credit.

Question 2

Students were asked to change the *toString()* method in the *Finalist* class, so that if a candidate was borderline (defined in the class to be less than 1 mark from a degree mark boundary) then the *toString()* method would include a statement about their borderline statement in the `String` returned.

From the model answer, you can see that this question can be answered quite simply, however some students lost credit by using logic to include the borderline status that was both complicated and wrong, such that the statement was never included in the output. Others lost marks by using no logic at all, just adding the statement 'Candidate is BORDERLINE' to the output of the *toString()* method with no attempt to only include the statement when appropriate, while some used faulty logic or syntax, such that the statement was included with all output, for example:

```
if(borderline = true) //add the borderline statement
```

The single '=' is assignment, not equality testing.

Another common error was having the *toString()* method print the borderline statement with a `System.out.println()` statement. These students got some credit if their logic was correct, such that the statement was only ever printed for borderline candidates, however lost credit because it made the output confusing – for example when saving *Finalist* objects to a file, the borderline message is printed to standard output, but it is not saved in the file. Another mistake seen more than once was that if the candidate was borderline then only the borderline message was returned by the method.

Question 3

Students were asked to change the *findFinalistID()* method to print a 'not found' message, when a *Finalist* object with the given ID was not found in the *finalist* `ArrayList`. The surprising thing about this question was that no student noticed the mistake in the method, in that the `for` loop to search the `ArrayList` searched the array starting at the item indexed by 1, hence the item in position 0 could never be found. This would mean that the method would not find the object with the *id* 62138, the ID of the first object in the `ArrayList`. This also shows poor testing on the examiners' part, since this error was not found by the test statements in the main method.

The most common error made by students was in having a message to the user printed with every iteration of the loop. Hence if the item was found then the user was told their search item was found, and if it was not found then the 'not found' message was printed. This made the output confusing, and in any case the user only needs to be told once if a particular item is found or not.

Apart from mistakes made by the examiner, student mistakes included:

- not including the ID number searched for in the 'not found' message
- including the wrong ID number in the 'not found' message, usually the last ID number from the list just searched, so the user is told that an ID number that definitely exists, does not exist
- the 'not found' message is never output due to a bracketing/logic error (there are no brackets around statements that should be included in the `for` loop, hence only the first statement is included in the loop)
- outputting the 'not found' message inappropriately. Either (1) the 'not found' message is output with the found item, but is not output when an item is not found, or (2) the 'not found' message is output both when an item is found, and when it is not because the logic to determine output is faulty in some way. Some students need to note that the expression `if(!finalists.contains(s))` will evaluate to `false` every time when *s* is a `String`, because a `String` is not a *Finalist* object. You can check this by looking up the method in the `ArrayList` API, where you will find that the method `contains(Object o)` takes an object and returns true if and only if that object is in the `ArrayList`, and throws no exceptions to tip off the user that their `Object` is inappropriate for that ArrayList, see https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#contains-java.lang.Object-

Question 4

Students were asked to write a method, *findFinalistClass()* that found and printed member(s) of the *Finalist* class with a particular class of degree, printing a 'not found' message if no matches were found in the *finalists* `ArrayList`. All errors seen were in the messages to the user; the searching was always implemented correctly. Some minor mistakes were seen in the message to tell the user the search item had been found, such as not including the class of degree searched for with the messages to the user, or just telling the user the search term had been found without displaying the details of the found objects, but by far the most common errors were displaying too many 'not found' messages and not displaying a 'not found' message at all.

'Not found' messages were not displayed either because the student had included no logic to tell the user when the search item was not found, or because of errors such as failing to enclose in brackets all the statements that are intended to be undertaken with each iteration of the search loop. Another error was caused by not including statements for an `if` expression in brackets, meaning that every item in the list was displayed. 'Not found' messages were also displayed inappropriately with found items, at the same time as not being displayed when the search item was not found.

Much more common than not displaying a 'not found' message, was displaying the message multiple times, usually because each iteration of the search loop had an `if/else` expression to either print the 'not found' message, or output that a match has been found. Hence each iteration of the loop printed out a message to the user, that the user only needed to see once, if at all. This a very easy mistake to find by testing, and the test statements in the main method would have revealed this error, had students run the class and read the output.

Question 5

The method *finalistsInList()* makes an `ArrayList` of *Finalist* objects, reading data to make the *Finalist* objects from a text file. In the class given to students the 'throws Exception' in the method heading meant that the method was passing any exception handling to a calling method. The main method also had 'throws Exception' which meant that if the *finalistsInList()* method threw an exception when run by the main method, the program would stop. Students were asked to add exception handling to the *finalistsInList()* method. One way to find out what possible exceptions could be thrown by the method would be to remove 'throws Exception' from the method heading and compile. The compiler would report:

```
error: unreported exception FileNotFoundException;
must be caught or declared to be thrown

Scanner in =new Scanner(new FileReader(s));
                         ^
```

Students could either add try/catch blocks to catch the specific `FileNotFoundException`, or could have their catch block deal with an `IOException`, or just an `Exception`. Exceptions are polymorphic, which means that they can be handled by handling one of their super types. If you consult the API for the `FileNotFoundException`, https://docs.oracle.com/javase/7/docs/api/java/io/FileNotFoundException.html, you can see that its parent class is `IOException`, and `IOException's` parent class is `Exception`.

Another way to determine what exceptions may be thrown by a method, is to look up the objects and its methods in the API. In this case we would look up the `Scanner` and `FileReader` constructors, and we would discover that they could both throw a `FileNotFoundException`.

This question was attempted well on the whole, with only a minority making mistakes, as follows:

- Some students could have included a more informative message to the user from their catch block, an error message such as 'An exception occurred' does not tell the user anything that they do not already know. It would be better to output something along the lines of 'There was an error opening the file.' However outputting something from the catch block was better than nothing; some students gave no error message from the catch block at all.
- A common error was the `NullPointerException` error, where the `FileNotFoundException` was handled by putting just the statement `Scanner in =new Scanner(new FileReader(s));` in the try block. Once the catch block had concluded, the `while` loop to read from the text file would start. The problem is that at this point the `while` loop may try to use a `Scanner` object that will be `null` if the file does not exist or cannot be found. Hence, if the file name is invalid a `NullPointerException` will be thrown because the `Scanner.hasNextLine()` method will be invoked with a `null Scanner`. This is why in the model answer the `while` loop to read from the text file is included in the try block, so that if the `FileNotFoundException` is

thrown, then control will move to the catch block and the `while` loop will not be executed. Note that a `NullPointerException` is an unchecked exception, in that the compiler does not check to see if it has been thrown or caught. There will be more about checked and unchecked exceptions in CO2220.

Question 6

The *finalistsToFile()* method given with the *ProcessDegreeMarks* class did two things, one was to make a copy of the `ArrayList` it was given as a parameter, and sort it using the *FinalistComparator,* the second was to save the sorted `ArrayList` to a file. Since it is not good object-oriented practice for methods to have two major tasks, students were asked to divide the *finalistsToFile()* method into two methods. One should sort, and one should save the `ArrayList` to a file. The new methods were to be called *sortDegreeMark()* and *finalistsToFile2()*.

The most common error was made in the *finalistsToFile2()* method by the majority of students, who did not write any logic to check whether or not the `ArrayList` was empty before saving it. Other errors included:

The finalistsToFile2() method

- Writing an otherwise correct method but not handling the potential `FileNotFoundException`
- Writing the entire `ArrayList` to the file, with each iteration of the loop that is writing to the file. Since loops to write to the file would normally iterate based on the size of the `ArrayList`, this would mean that the entire contents of the `ArrayList` would be written to the file 8 times.
- Saving only one item into the file by closing the file at each iteration of the loop.
- Saving nothing into the file (1) because the `ArrayList` saved is empty due to a logic error or (2) because there is no attempt to save to a file, and the method prints the `ArrayList` items to standard output instead.
- Less than one percent of students made this error, but it was a completely new error as far as the examiners were concerned, in that the method has the statement `System.out.close(),` which immediately ends all output from the program. It is hard to think of a good reason to use `System.out.close()` in your programs. Presumably the students concerned intended to close the file.

In their error handling, a few students (6% in fact) used try/catch/finally. A finally block comes after a try/catch block, and tells the method that this is something that has to be done whether control has passed to the catch block or not. In this case most students used the finally block for closing the `Scanner`, which is very much a standard use for a finally block. A few first checked if the `Scanner` was not null before trying to close it, which seemed like a very good idea.

sortDegreeMark()

In the *sortDegreeMark()* method mistakes were rare, with only very few seen, the most common of which was to sort the `ArrayList` given to the method as a parameter, rather than making a copy of the `ArrayList` parameter and sorting the copy, which was what the question asked for.

Question 7

Students were asked to write a method called *findAndSaveFinalistClass()*. The method would search for *Finalists* with a particular class, given by the user. Results would be added to a new `ArrayList`, which would then be saved into a file using the *finalistsToFile2()* method. Any student who successfully answered this question is to be congratulated, since this was the question that gave rise to numerous and varied errors, so completely correct answers were rare. The best answers made the `String` input parameter upper case before searching for matches.

The most common mistake was in saving to the text file with every iteration of the search loop, so that with each iteration the `ArrayList` of matches was saved, and anything already in the file was overwritten, a waste of resources. All found items should be added to a new `ArrayList` of matches, that should then be saved *just once* (*i.e.* not in a loop) using the *finalistsToFile2()* method. Other errors with saving included: making no attempt to save to a file; saving an empty `ArrayList` to the file, either because there was no logic to prevent saving an empty list, or because the logic was attempted but failed; and saving the `ArrayList` given as a parameter, rather than the new `ArrayList` of matches.

Other mistakes concerned the file name to save the `ArrayList` of matches into. One mistake was in making the file name too specific, by calling it, for example, *FIRSTOnly.txt* or *FIRSTFinalists.txt*, when really what was wanted was a file name partly constructed out of the search `String`, as in the model answer. Another error was in not giving the file name an extension, which in this case should have been '.txt'.

[1] *See* https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#clear-- *accessed 15 March 2019*

Question 8

Students were asked to add an appropriate constructor to the *ProcessDegreeMark* class. Since the class has only static methods, it would not be appropriate to make an object of it. Classes that contain static methods for doing some useful tasks with other objects are known as static utility classes. Sometimes they are given a private, empty constructor. Since the constructor is private it cannot be accessed from outside the class, therefore it is not possible to make an object of the class.

Java has four access modifiers for classes, constructors, methods and variables: private, protected, public, and the default (package). The default access modifier is given to items that do not have an access modifier, and means that the item can only be accessed within its containing package. Public means, as you would expect, it is open to all, private means an item can only be accessed from within the class, and protected means that the item can be accessed within its containing package, and in any sub-classes. Hence private is the right access modifier to prevent instantiation of the *ProcessDegreeMarks* class. This question could be answered simply, and correctly, with `private ProcessDegreeMark(){}`

Despite the reading given for this question (see www.javapractices.com/topic/TopicAction.do?Id=40) that clearly explained the concept of a private constructor, there were quite a number of students who wrote a public constructor, and an even larger number (28%, or more than a quarter of submissions) who made no attempt at this question. There were also a small number of students (2.4%) who made *ProcessDegreeMarks* an abstract class to prevent instantiation. Since the major use of an abstract class is to provide shared implementation of common behaviour for child classes, and since it would not be appropriate for a static utility class to have child classes, the *ProcessDegreeMark* class should not be abstract. This is particularly true as adding a private constructor to the class is a much simpler way to achieve the same goal, and the simplest way is usually best as it is likely to have less unpredictable side effects.

Conclusion

While both coursework assignments were answered well on the whole, the number of small errors that could easily have been identified by testing surprised the examiners. Students need to understand that when your work compiles, it does not mean that it works exactly as you expect it to; even the most experienced developer can be tripped up by errors in their logic.