Coursework commentaries 2015–2016

CO1109 Introduction to Java and object-oriented programming – Coursework assignment 1

General remarks

In each coursework assignment, students were given one mark for handing in their Java files not in a directory and one mark for handing in uncompressed files. These instructions are so easy to follow that it is quite surprising that there was a minority of students who did not follow them. The examiners are at a loss as to how to make the instructions for deliverables more explicit. In addition, some students lost marks because of simple and avoidable mistakes with the files that they gave in, as follows:

- They submitted class files **only** (the examiners are unable to give any marks for class files).
- They submitted files with a mismatch between the file name and the class name, usually because of capitalisation (Speak.java vs class speak), hence the files would not compile.
- They submitted a pdf with the Java files copied into it, but no Java files. The examiners will give a mark of zero to students who do not follow the submission instructions (which asked for .Java files).
- They gave in files that would not compile, sometimes because of multiple errors, sometimes because of simple things like failing to close a comment. If a file will not compile, the examiners cannot test it and thus cannot give any marks for it. Please remember to check your work.

Comments on specific questions

Coursework assignment 1

Please note the following sample answers are provided with the report on coursework assignment 1:

Part (a): Speak.java

Part (b): InvoiceCreator.java

Part A: Speak.java

Question 1: getRandomString.java

Students were given a class *Speak*, consisting of some static `String` arrays and asked to write a static method that would take a `String` array as a parameter and return one element chosen at random from the array. This task was designed so that students would learn how to write a method that would work for arrays of different lengths.

The question told students that the method should be of type String, which meant that the method should return a `String`. A significant minority of candidates instead wrote an `int` method that returned a random number, and then used that number to choose the array elements in their *getRandomSentence()* method. Some of these students did not make sure to generate a random `int` between zero and the length of the array minus one and so they lost marks both for not returning a `String` and because their testing did not identify that their method could not return certain array elements (or certain array index numbers, to be more precise). A few students lost marks for writing void methods (i.e. methods that returned nothing). Many of these students printed the random array element to standard output with *System.out.println()*. All then had difficulty in successfully completing the second question.

The *getRandomString(String[])* method should take an array variable and return an element from the array chosen at random. Since the array parameter taken by the method could be of different lengths, the most challenging part of writing the method was making sure that every element of any length array had an equal chance of being chosen. This was achieved by generating a random number that was between zero and the length of the array minus one. That is, the length attribute of the array had to be used. For example, if the array parameter for the method is called *arr*:

```
Random r = new Random();
int randomW = r.nextInt(arr.length);
```

In the above, the *Random* class generates a pseudo-random `int`. The *Random* class also has methods to generate pseudo-random doubles, floats etc. The *nextInt()* method is overloaded; without a parameter it will generate any positive `int` value, up to the limit on `int`. With a parameter it will generate a number between zero and one less than the given parameter. So for example `r.nextInt(5)` will return a number between 0 and 4, including 0 and 4. It cannot return the number 5.

While a few students need to note the above information about the *Random* class, the majority of students successfully answered this question and gained full marks.

Question 2: getRandomSentence()

Students were asked to write a method that used the *getRandomString(String[])* method to construct a random sentence using each of the static `String` arrays in the *Speak* class once. Most students had no trouble writing this method. Again the method should have been of type `String` and most students understood this, but a small minority lost marks for writing a void method.

Question 3: The main method (testing)

Students were asked to write a main method with at least one test statement for the *getRandomString(String[])* method. Students were free to write as many test statements as they thought necessary.

Most students gained full marks for this question, although a significant minority lost marks for user interaction. User interaction is a bad choice for testing; it does not help you to be systematic and to make sure that you have covered every possibility in the way that writing down your test statements does, since, once they are written down, you can review and amend them.

The examiners were expecting a loop in order to generate many random sentences and/or to test the *getRandomString()* method, so that enough data could be generated to be sure that each and every array element could be output by the class, most particularly the first and last elements of each array.

Question 4: Explanation of testing

A large minority wrote no test comment at all. The examiners expected students to explain the number of tests they had run and what they were looking for in these tests (i.e. what they would consider to be a successful test and what they would consider to be unsuccessful).

Students were expected to identify the major issue as being to return every element of each array, particularly the first and last elements, and lost marks if they did not explain how their testing had demonstrated that every array element could be chosen. In fact, only 1 in 5 students explicitly tested that the start and end values of every array could be chosen by the *getRandomSentence()* method. A few students did very elaborate testing, probably too elaborate, but even so, failed to test that the start and end elements of every array could be chosen. Among those who did test start and end values, a subset also tested to make sure array elements could not go out of bounds thus giving a run-time error, which was good.

A minority of students had not got their random number generation completely right; sometimes meaning that the first element of any array given as a parameter could never be chosen; but more often meaning that the last element of at least one of the arrays could never be chosen. All of these students lost marks by not identifying these issues in their testing. A few students wrote a method that could choose all possible array elements, at the expense of sometimes generating a random number larger than the size of at least one of the arrays, thus giving an occasional run-time error. These students lost marks for not identifying this issue in their testing.

One thing that a large minority of students tested for (again about 1 in 5), was to see if each element had an equal chance of being chosen; that is, they were testing **randomness**. These tests were done using a loop, counting the number of times each array element was chosen and considering the test passed if each element of an array appeared an almost equal number of times as the other elements of the same array for a reasonably large test sample (say 1,000). This amounted to testing Java's methods for generating pseudo-random numbers. If Java's random number generation does not work, that is not something that students can fix. It might be an interesting exercise in itself, but not at the expense of testing for the errors that developers can fix, that is your own logical mistakes.

A good answer might have said:

> I tested the *getRandomString()* method by calling it in a loop, to check that all array elements could be returned; in particular, the first and last element of each array. I ran the loop until at least the start and end values of each array had been generated once. I assumed that if the start and end values could be returned by the method, then all intermediate values could be too. I tested the *getRandomSentence()* method just once, to make sure that the words were being output in the correct order and the sentence was grammatical. Since I had already tested the *getRandomString(String[])* method many times and found no errors, I did not think that more was needed. I did not find any errors in my testing, including that I did not get any exceptions thrown despite running the methods many times. Since this assignment did not ask me to do any validation checking, I did not test to see if the *getRandomString(String[])* method accepted a null or empty array.

Conclusion

Most students had no problem writing the methods and test statements. By far the most common error was failing to understand that testing should find out if all elements of the arrays could be randomly generated, particularly the start and end values.

Part B: InvoiceCreator.java

Part (b) asked candidates to complete the *InvoiceCreator* class, such that given a text file as input, it produced a formatted invoice as output, using the information in the file. There were six methods for students to write.

Question 1: getFromUser(String)

The vast majority of students gained full credit for writing this method. A very small number lost marks for compilation errors. Another small minority lost marks for making their loop frustrating and slow for the user (e.g. once the user has entered the information requested, asking to press enter to continue before the program would move on). Some students did not make the question to the user a parameter of the method, such that a different question could be asked each time the method was run. Instead the question(s) were put inside the method meaning that either:

- all of the questions were repeated each time the method was run, making the loop confusing for the user (i.e. the user is asked 'What is the job title?', 'What is the hourly rate?' and 'What are the hours worked?' every time the method is run, even though only one of these questions can be answered at a time).
- the same question was asked every time ('What is the job title?'), even when different data was required from the userwere required.

Question 2: getAddress()

More than 60 per cent of students limited the number of address lines that users could enter. In fact, these programs enforced that an address had to be a certain number of lines, the user could not quit before entering the chosen number of lines (usually three, often four). The worst were those programs that only allowed one address line, clearly not a workable solution in the real world. The best answers allowed the user to enter as many address lines as necessary and signal when they had finished; for example, by entering a blank line.

Some methods, while not limiting the number of address lines, asked the user to indicate at every line whether or not they wished to continue, by typing 'y' or 'n' say. Most users would find this frustrating.

Question 3: getIntFromUser(String)

This method was the source of an issue that many students could not fix successfully – the *nextInt()* method of the Scanner class can leave a hanging new line. In practice this meant that running *getIntFromUser(String)* followed by *getFromUser(String)* would result in *getFromUser(String)* returning the rest of the line that the `int` read by *nextInt()* was part of, which would probably be the new line character but would not be what the program was expecting.

One way to deal with this is to read in the new line at the end of the method, for example:

```
i = in.nextInt();

in.nextLine();

//this eats the hanging new line, ready for the next use
of in.next()
```

Another option is to read the whole line (*with nextLine()*) as a *String* and use *Integer.parseInt(String)*. See http://stackoverflow.com/a/13102066

Too many fixed the problem by making a new *Scanner* object in the loop after the *getIntFromUser(String)* method had been called for the final time in that iteration. Making a new *Scanner* object is an unnecessary use of system resources. Of course this does not matter in a small program, but it is a bad habit to get into. It was disappointing to see the number of students who were defeated by this and chose a quick work around, rather than, for example, reading the API and trying to work out what was going on.

Question 4: getVat(int)

The examiners intended that students would write a method *getVat(int)* that used integer arithmetic. Most students successfully wrote this method; unfortunately, a few fell into the trap of integer arithmetic, with their methods returning zero. To illustrate this, suppose one wishes to calculate 20% VAT on £86? We could do this:

86 / 100 = 0.86

0.86* 20 = 17.2

In integer arithmetic this would be:

86 / 100 = 0

0* 20 = 0

This is because integer division returns the number of times one number divides into another and the remainder is thrown away. Because 100 divides into 86 zero times, the result of 86 divided by 100 is zero. In order to make sure that the calculation gives the expected result, one needs to multiply by 20, then divide by 100, that is:

86*20 = 1720

1720/100 = 17

Many students commented that integer arithmetic is not accurate and working with doubles would have been better. Some students made the point that *BigDecimal* would have been better still. Both were correct, in particular *BigDecimal* is always used in real world currency calculations for its accuracy, but the question was testing if students could successfully navigate the pitfalls of integer arithmetic.

Question 5: addVat(int)

While most students got full credit for answering this question, there was one very common mistake. The method should use the *getVat(int)* method to calculate the amount of VAT, add it on the parameter given and return the total. However, many students calculated the VAT in the method. Doing the same calculation in the *addVat(int)* method that is done by *getVat(int)* method is not wrong, so much as it is bad practice; it breaks the programmer's maxim **DRY – Don't Repeat Yourself**.

Question 6: getDateIn30Days()

Most students wrote this method with no problem at all, achieving full credit for using the *add()* method of the *Calendar* class. Almost all students who attempted this question got full marks for it, very few mistakes were seen, the only common mistakes being adding the wrong number of days (31 instead of 30, for example), and displaying the time as well as the date.

Question 7: StringBuilder

Students were asked if *StringBuilder* would have been a better choice than *String* for the variable *s* in the *printInvoice(String, String, int, int, String)* method. The examiners were expecting students to write about how *Strings* are immutable, which means that *Strings* cannot be changed, so each time a *String* is concatenated, a new *String* is made, but the old *String* remains and is unreferenced, becoming garbage collectable. This can give an overhead in terms of memory. Hence, when using a loop where concatenation could potentially happen many times, *StringBuilder* would be a better choice, see http://stackoverflow.com/a/4645155. In addition *StringBuilder's append()* method is faster than *String* concatenation, again a small advantage in a small program, but programs will not always be small.

Many students lost marks here for vague and unfocused comments, which lacked clarity and explanation and/or were very short (often just a sentence or two). In fact only 48 per cent of students achieved full marks for this question.

Some students made the interesting point that *StringBuilder* has the advantage over *String* of throwing a *NullPointerException* if null is passed to it, which is good as the invoice should not be null. In fact the *StringBuilder* constructor throws an exception; the *append()* method does not.

A good response is worth quoting in full here:

'StringBuilder would have been a better choice because of two reasons:

1. String objects are immutable (Java Platform, Standard Edition 7 API Specification, n.d.), while StringBuilder objects are mutable (Java Platform, Standard Edition 7 API Specification, n.d.)
    a. Therefore, every time a String is concatenated, a new String object is actually created and referenced. The old String object is dereferenced and never used again.
    b. StringBuilder objects, on the other hand, are mutable, where its contents can change while maintaining the same object reference.
2. StringBuilder.append() operations are faster than String concatenations using String.format()
    a. I created a simple benchmark to compare the joining of the text in printInvoice(), by comparing the System.nanoTime() before and after the concatenation or append operations.
    b. I averaged the result over 1024 runs through NetBeans on an Intel Core i7-5500U processor.
    c. Using StringBuilder to append the text resulted in an advantage of 30249.266898155212 nanoseconds over the provided String concatenation.
    d. (The method used in benchmarking StringBuilder is included in a comment block at the end of this file, where SAMPLE_SIZE = 1024).'

Of course, students did not need to investigate the speed of the append method for full credit, but the depth of understanding and the spirit of enquiry shown here is admirable.

General comments

Many students wrote loops that were not particularly user friendly, making the user take tedious and unnecessary extra steps. One striking example of this was in the loop that asked the user if they wanted to enter more data. In the program as given with the coursework assignment, if the user entered anything starting with 'N' or 'n' the loop would stop, otherwise it would continue. Some students rewrote this to enforce that the user must enter 'Yes' or 'No' exactly; that is 'yes', 'y', 'Y', 'no', 'n' and 'N' were rejected as illegitimate input so that the user had to try again.

A handful of students wrote classes that were much too complicated for a very simple task given in part B; developers always try to keep their code simple and readable and so should students. In the real world, problems quickly become complicated and code hard to read, so getting into the habit of keeping it simple and readable now will help you in future.

Coursework commentaries 2015–2016

CO1109 Introduction to Java and object-oriented programming – Coursework assignment 2

Please note the following sample answers are provided with this report:

Part (a): StringInput.java

Part (b)

- ReceiptAfter.java
- Receipts.java
- receipt info.txt

Part A: StringInput.java

Students were given the incomplete *StringInput* program, and asked to complete it by correcting some methods and writing others.

Question 1: Writing the reverse(String) method

Students were asked to write a method that would reverse a *String* given to it as a parameter. Almost every student who attempted this question received full credit for a successful answer. A very small number of students lost some credit by writing the method such that it asked the user to enter their word again.

Question 2: Completing the alphabetise(String) method

Students were asked to fix the method, so that it treated upper and lower case versions of the same letter as if they were the same character. The majority of students gained full credit for this question, although some common errors were seen. Those students who made the word all lower case and then sorted the characters in alphabetical order lost a lot of credit. The question made clear that this was not what was required.

Apart from this, there were some minor individual errors, including not having any output from the method and printing out all the intermediate steps taken by the method, which the user does not need to see.

Question 3: Writing the randomise(String) method

This was very challenging to write and pleasingly the majority of students got full marks for it. The most common mistake seen was improper randomisation; that is, the method repeated some characters in the word and removed others completely.

Other mistakes could have been identified and fixed with good testing; for example, where the program went into an infinite loop, or did not display the output to the user, or threw an *IndexOutOfBoundsException*. One reason for throwing an exception, seen more than once, was caused by a `for` loop operating over the length of an *ArrayList* containing all the characters of the word. The size of the word (and hence the size of the *ArrayList*) determined the number of iterations of the loop. Within the loop, elements were removed from the *ArrayList*, hence the size of the *ArrayList* would decrease, but the variable containing the size of the *ArrayList* was not updated, so the loop would eventually attempt to access elements that did not exist, giving the exception.

Question 4: Fixing the frequency(String) method

The frequency method had no output, so students were asked to find and fix the problem. The issue is in the following code:

```
if (j == length) {
   System.out.println(array[i]+”\t”+count);
}
```

Since *j* would never equal the length, there would never be any output, although otherwise the method worked as it should. Students were expected to change the condition to:

```
if (j == length-1)
```

While the majority of students found and fixed the error, a few students rewrote the method completely. A common mistake was caused by students implementing a solution that meant that the word was printed as a list of characters, with a frequency count next to each one. If a character appeared more than once in the word, then it had the correct frequency count on its first appearance and a count of zero for subsequent appearances in the list. Students were asked to write a comment about how they had fixed the method and strangely some comments showed no understanding at all, even though the method had been fixed.

Question 5: A method to remove the vowels from a String

Most students did this correctly and gained all marks. The most common error (seen 18 times) was writing a method that only removed lower case vowels from a word. For example, the word ‘Eamonn’, would be returned as ‘Emnn’.

Other mistakes were not being able to remove a vowel if it was the first or last letter of the word, not being able to remove two consecutive vowels, sometimes removing consonants as well as all the vowels, not being able to remove lower case ‘u’s and removing vowels in a loop that skipped some characters and so missed some vowels. All of these mistakes could have been found and fixed with good testing.

One student wrote a method that told the user there was nothing to display, where if a word was entered that was all vowels. This was a good thought.

Question 6: A loop to allow the user to choose more than one menu choice

Question 7: An outer loop to allow the user to enter a new word

Students were asked to write two loops, an inner one being that which allowed the user to repeatedly choose from the menu, with an option to quit. If the user chose to quit, they were taken to an outer loop, where they were asked to choose to end the program or re-enter the inner loop with a new word. While the majority of students gained full marks for their loops, more than a quarter of students had problems with their answers.

The most common reason for losing credit for Questions 6 and 7 was making no attempt; 7 per cent of students did not do this part of the coursework assignment. The user interaction loop that students were asked to write was quite challenging; however, those who managed to complete it did well.

The inner loop

Most issues seen here were minor and had the effect of making the loop more difficult or time consuming for the user. Since the coursework assignment was not testing user-friendliness, the examiners did not deduct marks for any of the following:

- asking the user to press enter to continue after every menu choice has been completed
- asking the user if they wish to continue after every menu choice, entering 'y' to continue and 'n' to stop
- asking the user to re-enter their word after every menu choice.

Note that the first two items in the list above are particularly pointless as the user, in a properly written loop, can choose to quit at any time.

Other errors for which marks were deducted included:

- Getting error messages when there was no error; for example, when the user chose to quit giving the message 'error executing option', even though the program was working properly and the user could quit successfully
- The inner loop rejecting an invalid choice made by the user, but then not being able to recognise the next valid choice entered, but after that recovering and working as it should.

The outer loop

A number of programs would allow the user to enter a new word, but then go back to the inner loop operating on the first word that the user entered, as the variable holding the word had not been successfully updated. More seriously, a few students could not get their outer loops to work at all, sometimes managing to allow the user to enter a second word, but after this the program ended.

There were a handful of programs that threw *IllegalStateExceptions*, in each case because of trying to access the *Scanner* object after it had been closed. This was either because the *Scanner* was being closed every time through the loop, when it should, of course, only be closed at the very end of the program, or because the *startLoop()* method was being run recursively and when the user tried to quit the first copy of the method closed the *Scanner*, then all the other copies would try to close the *Scanner*, which was already closed, leading to the exception.

Other students also had problems with calling the loop recursively. In these cases, when the user tried to end the program, if the user had entered more than one word, the outer loop would go back to the inner loop instead of ending. This was because, when the user tried to end the program, there were as many versions of the *startLoop()* method running as words entered, less one, and these methods would start again at the point of executing one of the options in the inner loop.

These students were brave to try recursion, but needed more thought and testing to make it work.

There were a number of minor errors, which the examiners made no deduction for, again because they only had the effect of making the program less user-friendly.

- After the user chooses to enter a new word, they are asked to confirm that they want to enter a new word by typing 'y' or 'n'.
- If the user says yes to entering another word, the outer loop asks again if the user wishes to enter another word.
- When the user chooses to end the program, the outer loop asks for another word before quitting.

Summary

Many of the errors described above, particularly those with respect to the user interaction loop, could have been addressed with either more thought about making the program user friendly, or more testing to identify and fix errors.

Part B: ReceiptAfter.java

Students were given the *ReceiptBefore* class, which took a text file as input and output a formatted text file. All the statements were in the main method, students were asked to rewrite the class, calling the new class *ReceiptAfter*, breaking the statements up into methods, and writing a constructor for the class. Students were to decide which of their methods should be static, and which should be instance.

Question 1: Writing the ReceiptAfter class with methods

Students were expected to break the class into several methods.

Following this, the most obvious way to break the class into methods was to have a method to read in from the file and populate the *ArrayList*, to have another method to make the formatted receipt from the *ArrayList* and a third method to write the formatted receipt to a text file. In fact 30 per cent of students more or less stopped there. This meant that the method to make the formatted receipt was doing a lot of work and it would have been better to break it up into more methods. The examiners expected this to be done by dividing statements into meaningful chunks, each reflecting a discrete part of the invoice. Naming methods for the task they were doing would make the program easier to read and would make it easier to trace and fix problems. Writing methods that just have one task to do is a well established principle; for more information, see: https://en.wikipedia.org/wiki/Single_responsibility_principle.

How to divide making the formatted receipt into methods was a matter of judgement. For example, many students chose to have a method to make the very first statement on the receipt, usually called *header()*, and a *footer()* method to add the final statement. This made perfect sense, although it is not the approach taken by the model answer. Some then had one method for the rest of the receipt, while others broke it down into a method for cost and payment details and another for the list of items with quantities and prices. There were different approaches and, as long as they made sense, students got full marks.

A minority of students lost marks by putting statements that should be in a method into the main method, or by having some processing of the invoice done in the method to read in from the file. It is also possible to write too many methods, making the program harder to read; for example, having a method for nearly every statement that was part of the formatted invoice.

Methods and encapsulation

Some students chose to have the user call all the methods to make the receipt and write it to the file, which is not very user friendly, as the user really does not need to know how the receipt is made. All the user needs is to be able to tell the object what file to read from, and what file to write to. In addition, this breaks encapsulation; details that the user does not need to know to use the class successfully should be hidden. If the class has a single responsibility (which it should have), then it should only be necessary for the user to run one *public* method in order to carry out that responsibility. Other methods should be private. Making the user run several methods in order to get a formatted receipt means that the user could run them in the wrong order, potentially ending up with an object in an invalid state.

The main method of the model answer has only these statements:

```
ReceiptAfter ra = new ReceiptAfter("receipt info.txt");
ra.write("final receipt.txt");
```

As the user only has to run one method to get the final receipt, the object is easy to use and maintains proper encapsulation.

Question 2: Static and instance methods

One third of students attempting part (b) made no attempt at this question, typically because all their methods were instance methods, although there were a few who wrote only static methods.

The model answer has two instance methods. The first is a method to construct the receipt, that itself calls other methods to make different parts of the receipt. This method is called in the constructor. The second is the method to write to the file, since it makes sense for the user to run the object by telling it where to save the formatted receipt.

In the model answer, the method to construct the receipt is an instance method because it operates on the two instance variables, which cannot be done in a static method. The method calls other methods to construct the receipt and these methods themselves are all static as they do not directly reference any instance variables. The examiners expected to see students constitute all methods that did not directly access instance variables as static. Whether or not the methods that formatted different parts of the receipt accessed instance variables or not, of course, depended on how individual students structured their programs, which varied. Many different, correct, solutions were possible.

Some students gave in a separate class containing their static methods, which was a reasonable thing to do. The model answer has a separate class, with static methods to read in from a text file and store each line of text in a *String ArrayList* and methods to write from a *List* into a text file. The model answer has other static methods, but these are not in the separate class as they are so specific to making and formatting the receipt, that it makes sense to keep them in the *ReceiptAfter* class. The methods to read and write could be used by other classes seeking to read and write to text files with *Lists* and so it does make sense to put them in a separate class.

Common mistakes

- Having instance methods that do not operate on any instance variables
- Not allowing the user to tell the object what file to save the formatted receipt to (e.g. by making this a parameter for the method to write to file).

Question 3: The constructor

In order to write the constructor, students had to decide which variables should be instance variables, since it is the job of the constructor to initialise the instance variables and get the object ready to do its job. The model answer has just two instance variables, a *List* to hold the contents of the file with the raw data for the receipt and a *List* to hold the final, formatted receipt. All other variables are static or local to certain methods. Static variables are the same for every object of the class. Since the header and footer statements are the same for every receipt, it makes sense for these *Strings* to be static. In fact, it makes sense for them to be static and final; that is to say, constant values (the value of final variables cannot be changed once initialised), and that is what they are in the model answer. One thing to note is that the constructor should never, ever be initialising static variables. This, unfortunately, was done by more than one student. The compiler will not flag this as an error (the constructor is implicitly static so can access static variables), but in terms of making an object, it is wrong, very wrong.

The model answer has just two instance variables because of the way the author has chosen to structure the program. Having more instance variables would not necessarily be wrong, but some students had many instance variables, with only one or two initialised in the constructor. That is wrong. The constructor should initialise the instance variables; if your variable is not being initialised in the constructor, then it probably does not need to be an instance variable. Despite different structures being possible, the examiners were looking for a constructor that initialised and populated the *ArrayList* that held the unprocessed contents of the input file. That is to say, the constructor should call the method to read in from the file in order to populate the *ArrayList* with the contents of the file; only a minority of students did this.

The constructor in the model answer has one parameter; the name of the file from which to read in the data for the receipt. Many students also chose to make this a parameter for their constructor, but equally many did not. Not putting the file name as a parameter meant that the user could not choose which file to read in from, which does not make the object very useful from the user's perspective.

Summary of mistakes

- Failing to populate in the constructor the *ArrayList* to hold the raw receipt details (most common mistake).
- Having the constructor initialise a *static* variable.
- Having instance variables that the constructor does not initialise.
- Writing a constructor that does not initialise any instance variables but, for example, writes something to standard output, or makes some more variables, or calls a list of methods.
- Writing an empty constructor. A class with no instance variables does not need a constructor, a class with instance variables should not have an empty constructor.
- Not allowing the user to tell the object what file to read from (e.g. by making the file name a parameter for the constructor).

Question 4: Testing

It was quite acceptable to test by making one object, with one formatted receipt as output. This is because there were no loops or conditional statements in the program meaning that different starting conditions did not need to be tested. It would be reasonable to assume that if the class worked for one set of legitimate inputs, it would work for another. In addition, students were not asked to do any validation, so it was not necessary to test to see what would happen with illegitimate input, such as an empty file.

Most students gained full marks for this question. A few errors were seen, such as overly complicated test statements. Also, some students who did not realise that testing was demonstrating errors, such as a part of the receipt not being made because the method to make it was not being called anywhere.

Question 5: Comments

About 15 per cent of students did not add any comments to the ones already in the *ReceiptBefore* program. Most students who attempted this question received all the marks, although a small number lost marks, because they had problems with their programs that the comments should have documented, but did not. Comments should also document any assumptions or decisions that you have made, for example:

- `//The reason I have decided to make the user call the methods to make the receipt is because the user might want to leave out certain parts of the receipt, such as the header.`
- `//I have decided to make the methods to read and write to the file static methods as the user might want to just write to or read from a file without making an instance of any object.`

General comments

Some students gave in several extra classes. Having a customer class, as a child class of a person class, when a person is just a name, does seem like an over-elaborate solution to a simple problem and would only make sense if the scope of the assignment were larger. While real world problems may grow, once a problem has been scoped, developers do not normally add unnecessary complexity.

Conclusion

The second part of this coursework assignment was quite challenging. There were many ways to approach the problem leading to many different solutions, and a need for students to exercise their judgement. Those who received a good mark for this coursework assignment are to be congratulated.