Coursework commentaries 2016–17

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

General remarks

Stand alone and uncompressed files

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 were so easy to follow that it is quite surprising that there was still a minority of students who did not follow them. In addition, some students lost marks because of the following simple and avoidable mistakes:

- They submitted class files and no .java files (examiners are unable to give any marks for class files).
- They submitted files with a mismatch between the file name and the class name, (e.g. *PartC.java* versus *TaxAdvisor* class name), hence the files would not compile.
- They submitted a PDF with the Java files copied into it. Examiners will give a mark of zero to students who do not follow the submission instructions, which required .java files so that they could be compiled and tested.
- The submitted files 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 examiner cannot test it, and thus cannot give any marks for it.

Comments on specific questions

Coursework assignment 1

Please note the following model answers are provided with the commentary on coursework assignment 1:

Part (a): PartA.java

Part (c): TaxAdvisor.java

Part a: *PartA.java*

You were given the *PartA* class and asked to write some static methods such that the main method of the class would compile and give the example output provided in the appendix of the coursework assignment document. The example output had been generated by running the program with a main method containing some test statements. The statements in the main method were used by the examiner to test your answers, hence you were instructed not to change them.

Question 1: *the censor(String)* method

In the main method, this method was tested three times, with a different `String` each time. The first input `String` was ‘short’, and the output given was 5 asterisks (that is, `*****`). The next input `String` was “longer

the short" and the output was 16 asterisks and the final input was "the longest one we've got" with 25 asterisks output. You were expected to think about how the input related to the output, in this case the length of the `String` input, gave the number of asterisks output. Therefore the *Censor(String)* method should have been written to output a number of asterisks equal to the length of the input `String`. Most students understood this, but a minority lost marks by writing methods that would not work for any legitimate input (that is, a `String`) but only for the input given in the test statements in the main method. For example:

```
static String censor(String s) {
   if (s==”short”) return “*****”;
   if (s==”longer the short”) return
   “****************”;
   if(s==”the longest one we’ve got”)return
   “************************”;
   return s;
}
```

Different correct answers were possible. The following would achieve full credit:

```
public static String censor(String word) {
   int len = word.length();
   String stars = “”;
   for (int i = 0; i < len; i++) {
      stars += “*”;
   }
   return stars;
}
```

Students were expected to deduce from the test statements and the output given that the *censor(String)* method would return a `String`, and the *print(String)* method would print that `String`.

Question 2: the *print(String)* method

A few students lost marks by writing a *censor(String)* method that would print the asterisks from inside the method. Again, these students were attempting to write a method that would give the example output, but that would not work in general. These students then had a problem with their *print(String)* method, as the output that the method was supposed to give had already been given. They solved this by writing a *print(String)* method that simply returned the input `String`; or by writing a method that had no statements in its body, namely:

```
public static void print(String line){}
```

Another approach that gives the example output, but does not to work for any legitimate input, was to have the *Censor(String)* method do no actual work, just return the input `String`. This would mean that each input `String` in turn would then be fed by the test statements into the *print(String)* method. The *print(String)* method is then tasked with giving the necessary output; for example, by having three if blocks, the first one might be as follows, with the others similar (note in the statements below *s* is the input `String`):

```
if (s.equals(“short”)) {
   for(int i=0; i<5; i++) {
   System.out.print(“*”);
}
```

Another approach was to change the main method to give the example output, a solution that received no credit. Some students took this approach to Question 5 also; that is, rather than writing a *print(String)* method and an overloaded *print(String[ ])* method, in each case they altered the main method to give the necessary output.

A few students lost some credit by not having their *print(String)* method print and move to a new line; instead the new line was added by having the *censor(String)* method append a new line to the `String` returned. Students should have understood from the `print("");` test statement and the example output, that the method would print a new line. In fact the method could be written very simply as:

```
public static void print(String line) {
   System.out.println(line);
}
```

Question 3: The *magic8Ball()* method

Students were expected to understand from reading the main method and considering the output that the method should return an element chosen at random from the answers array. A few students lost credit by outputting the element chosen from within the *magic8Ball()* method; some even added a loop to the method so that it would print 20 elements chosen at random.

Most students correctly wrote a method that found and returned one element at random from the array. A common mistake made by these students was not noticing that the final array element could never be returned because of a mistake in making the *random int*. These students also missed that one of the reasons to run the method 20 times was to read the output, and check that every element of the array was represented at least once.

Another mistake seen more than once was moving the *answers* array into the *magic8Ball()* method; these students did not understand that *answers* was a class variable, hence any method in the class could access it.

Question 4: The *drawStars(int, int)* method

Students were expected to understand from comparing the main method with the example output that the simplest (and therefore best) way to write the *drawStars(int, int)* method was that the output of the method should be an array containing a number of strings as given by the second `int` parameter to the method. Each `String` would be the same, being comprised of asterisks, the number of which to be determined by the first `int` parameter to the method. In other words, if the parameters to the method were *x* and *y*, the output would be a `String` array, with *y* elements, each one a `String` of *x* asterisks.

The array output would then be given by the main method to the *print(String[])* method, which would print each array element, one to a line. Hence the output to the user would look like a rectangle made of asterisks.

Even keeping it as simple as possible, the *drawStars(int, int)* method was the most complicated that students were asked to write for this part of the coursework assignment; and numerous mistakes were seen, including:

- Making the drawStars(int, int) method do the printing. Some students implementing this solution made their methods `void` because nothing needed to be returned, but this then caused a conflict with statements

in the main method, which were expecting a `String[]` returned, giving a compilation error.

- Logical errors leading to an `ArrayIndexOutOfBoundsException`
- Logical errors meaning that the method would not work as expected; for example, each `String` in the array would be '*' whatever the input.

Questions 4 and 5: The *drawStars(int, int)* method and the *print(String[ ])* method

Quite often the solution to Question 4 was linked to that of Question 5, but ideally each method should stand alone, and not depend on receiving specific input. For example, the *print(String[ ])* method should print the array contents one to a line. Some students had the *drawStars(int, int)* method append a new line to each `String` before adding it to the array. Then the *print(String[ ])* method would use `System.out.print()` rather than `System.out.println()`. A more complex solution would be to have the *print(String[ ])* method turn the array into a `String` (including the line breaks) and then use `System.out.print()` to print it.

A number of solutions that linked Question 4 and Question 5 in even more complicated ways were seen, including:

- *drawStars(int, int)* makes an array containing only one item, a `String` with line breaks. The *print(String[ ])* method will only print the first line of its array parameter.
- *drawStars(int, int)* makes an array with one item, a `String` with line breaks, and *print(String[ ])* converts the array using *toString()* then prints it character by character (this includes the line breaks) taking care not to print the start and end square brackets given by the conversion.
- *drawStars(int, int)* returns an array with two strings in it, which are integers converted to strings. *print(String[ ])* converts the strings back to numbers and uses them to print the required output, i.e. to decide on the number of asterisks in each line output, and how many lines to print. If *print(String[ ])* gets input that cannot be converted to *ints* it will throw an exception.
- *drawStars(int, int)* returns an array with a line of stars as the first entry and the number of times to print the first entry as the second entry (that is, the second entry is an `int` converted to a `String`). Question 5 converts the second String back to a number and uses it to print the required output. If *print(String[ ])* gets a `String` that cannot be parsed to `int` it will throw an exception.
- *drawStars(int, int)* fills the array with each element being either a single asterisk or a special character such as '|', used to denote a line break. Alternatively, an actual new line character is added as an array element where necessary. The *print(String[ ])* method then prints the contents of the array all on one line until it reaches the special character (or a new line character), at which point it starts a new line. This carries on until all the asterisks have been printed.

There were some simpler mistakes seen in the *print(String[ ])* method (losing a little credit); for example, printing the entire array on line, and printing the first array element as many times as the length of the array, when printing every array element would give the same output and would be more general.

Question 6: Explanation of the `for` loop in the main method

Some students showed no understanding, giving an answer along the lines of 'The `for` loop should find and print 20 strings chosen at random from the *answers* array.' Since this (and similar) answers simply repeated information given with the coursework assignment, students giving such answers received no credit for this question.

Most students understood that the loop had been written in order to provide a sensible test of the *magic8Ball()* method; but not all students understood what a successful test would involve, with quite a number focusing on the randomness of the random number generated. Answers that only discussed the randomness of the random number generation received no credit.

A good answer might have said:

> The `for` loop tests that the *magic8Ball()* method is able to return all six of the elements of the 'answers' array, and does not throw an `ArrayIndexOutOfBoundsException`. By running the method 20 times within the loop we are testing that random number generation in the method can produce every index of the *answers* array. There are two possible problems. The first is not generating every possible array index number such that certain values (most likely the start and end values "Without a doubt" and "Outlook unclear") can never be returned. The second is generating a number that is larger (or possibly smaller) than the array's largest (or smallest) index number, which will give a run-time error. Hence output that contains both the first and last array elements and does not throw an `ArrayIndexOutOfBoundsException` can be considered as a successful test of the *magic8Ball()* method. In fact, 20 might be too small a number to randomly generate all possible outcomes successfully, so running the test again, or increasing the number of iterations of the `for` loop might be necessary.

Conclusion

A handful of students gave in files with numerous compilation errors, and received no marks for part (a). These students would have been better off concentrating on making their answer **work**, even to just one question. It would have been better to give in a file with one working method, than to give in something that attempted all questions but got none of them right. Similarly, a very small number of over-complicated answers were seen. These students had written an extra class containing the methods, rather than just writing them in the *PartA* class. Unfortunately, not all of the methods worked as they should have. It is a good idea to concentrate on getting the basics right first.

Many students struggled to answer Questions 4 and 5, sometimes coming up with answers that were more complicated than they needed to be. Another very common error was failing to understand what the `for` loop in the main method was really testing.

Students were given what the coursework assignment instructions explicitly called 'example' output from the class. They were expected to **deduce** from this that **other example output was possible**, and **therefore other input**. Hence, in their answers, students were expected to write methods that would work in general, and were not expected to write methods designed to produce the particular output given; that is, the general **form of the input was expected to relate to the output in a logically consistent way** that would generalise for any correct

input. For example, consider the first method, *censor(String)*. Students were expected to write a method that would take a `String` and return a `String` composed of a number of asterisks equal to the length of the input `String`. A surprisingly large number of students lost marks by writing methods that worked for the particular example input, but did not work for any input `String`.

Part b: *PartB.java*

Students who answered this method successfully indicated in their comment that they understood that the *nextInt()* method of the `Scanner` class can leave a hanging new line. Without the `scanner.nextLine();` statement after the `num = scanner.nextInt();` statement, the hanging new line would be read into the *name* variable. This means that an exception is thrown when the program tries to access the *name* variable's starting character (using `name.charAt(0)`) since the new line character does not have one that the system can recognise.

Adding the `scanner.nextLine();` statement after the `num = scanner.nextInt();` means that the hanging new line is eaten by the *nextLine()* method, and the class works as it should. Another option would be to read the whole line as a `String`, namely: `num = scanner.nextLine();` and then parse the `String` to an `int` using *Integer.parseInt(String)*.

Only a minority of students gained full marks for this question. Most who lost marks did so because their comment showed no understanding, often simply describing the two different outcomes of the program with and without the `scanner.nextLine()` statement enabled. Some lost just a mark or two for an answer that showed understanding but was confusingly written. A few lost marks for answers that were too brief.

Part c: *TaxAdvisor.java*

In this part of the coursework assignment, students were asked to write a program to implement a simplified algorithm to work out the tax owed by salaried British workers. The simplified algorithm was given, as was some example output.

Question 1

This specified six methods that the final answer should include. Students lost marks here by not implementing the methods asked for (for example, combining two methods asked for into one method; or doing the work of a method in the main method); and by writing methods that did not work entirely correctly.

Question 2

For this question, once again students lost marks simply by not following the instructions given. One mistake was not using class variables. This would cause problems with linking methods, as the outcome of one method might not be available to another method since it had not been stored in a class variable that any method could access. This could lead to running a method more than once, in order to get the same result (for example, calling it once in a method that needs to use its result; and again in order to print the value in the tax breakdown to the user).

Another common mistake was forcing the user to call more than one method in order to get a tax breakdown. Finally, a very simple solution to accepting any reasonable answer to a yes/no question is to first of all send the `String` input to lower case. Then take the first character of

the `String`. If it is 'y' then the user has input 'yes'; if it is anything else, assume the user is saying no. Strangely, a number of students took instead the approach of comparing the answer given to each of the strings 'y', 'Y', 'yeah' 'yes', etc. Quite often the code to determine the answer to yes/no questions was repeated, leading to a violation of the instruction given (Question 3) not to repeat yourself. A good answer might have avoided repetition by writing a method to send yes/no answers to, such as:

```
private static boolean getAnswer (String s) {
   if (s.trim().toLowerCase().startsWith("y")) return
   true;
   return false;
}
```

In some cases, the class would only accept lower case (or sometimes upper case) answers to yes/no questions. Given that it is straightforward to change the case of input in order to test it (as in the above method), this was not acceptable and credit was deducted.

A very common mistake was in putting additional statements in the main method, commonly the tax breakdown to the user, which should not have been there. Since Question 2 said that 'the user should only have to call one method in order to run the entire program' the instruction in Question 4 to write a main method to test the program, should have indicated a main method with just one statement in it.

Some good answers were seen. The best answers did not ask the user questions about their personal allowance if their salary was above the limit for personal allowances.

Other common mistakes

- Not writing methods but having the main method do all the work.
- Not asking the user if they are a manual worker or wear a uniform, but instead asking the user if they have to wash clothes for work – we all have to wash our work clothes!
- Asking if the user wore a uniform to work, but not asking if the user was a manual worker. The examiners assumed that students making this mistake could not work out how to ask two questions about the £60 laundry allowance, but only give the allowance once.
- Assuming that the user can only have one additional personal allowance, for example, assuming that a worker cannot be both blind and wear a uniform to work.
- Confusing the limit for personal allowances (£122,000) with the threshold for additional rate tax (£150,000).
- Assuming that a person cannot earn less than £11,000, and rejecting input below this as invalid. Nothing in the information given justified this assumption.

Note: if you make assumptions, then you must state them!

Most common mistake

By far the most common mistake was inadequate (or no) testing. The examiners tested many programs using the same small set of test data, and quickly found many tax calculations that were simply wrong. Typically, the calculations were right in some cases, but wrong in others, for example:

- the tax allowance did not include additional allowances so was correct when users were not claiming additional allowances, but wrong

otherwise.

- additional (but not £11k) tax allowances were given to salaries too high to qualify, due to faulty logic in the tax allowance method.
- The correct tax was deducted if the worker only had to pay tax at the basic rate, but not if the user had to pay tax in one or both of the other tax bands.

Coursework commentaries 2016–17

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

General remarks

Please note the following model answers are provided with this commentary:

- *Student.java*
- *ProcessModuleMarks.java*
- *marks.txt* (not a model answer but needed to run ProcessModuleMarks)

Students were given *Student.java*, *ProcessModuleMarks.java*, and *marks.txt* (input for the *ProcessModuleMarks* class), and asked to do the following:

Comments on specific questions

Question 1: Write *getter* methods for the *Student* class

All students found this straightforward, with all gaining full marks.

Question 2: The *positionOfNext(ArrayList<Student>,int)* method

Students were asked to complete the *positionOfNext(ArrayList<Student>,int)* method. The logic of the method was quite challenging, demonstrated by the 20 per cent of students who received less than full credit for this question (or even no marks). Various different logical errors were seen. Some of them could have been corrected with better testing, since they partially worked; evidence that the student was on the right track and – with some testing – they might have found and corrected the error.

Question 3: The *displayStudent(ArrayList<Student>)* method

This method was done well on the whole, with only a handful of errors seen; there were no common errors.

Question 4: The *findStudentName(ArrayList<Student>,String)* method

The most common error here was searching for the entire input `String`, rather than attempting to find all student names that started with a particular `String`. Students were told in this question to look at the Java String API, and were expected to look for `String` methods to help them. Many did this, using the `String` method *startsWith(String)*, although a sizeable minority used the `String` method *substring(int, int)* to compare the start of the student name with the search string. This was a bit more complicated, but worked.

A fairly common mistake was to use the `String` method *contains(CharSequence)*. This would mean that the method returned all student names that contained the search string, which would include those that started with it, meaning that the method could return all correct answers, but potentially some wrong answers too.

Several students made mistakes with case insensitivity; for example, writing methods where the name of the current element being compared

to the search string was made lower case, but not the search string itself. Hence, if the search string contained any upper case letters, no match could be found. Another mistake was to write a method that correctly found all students it should, but did not output anything to the user.

Examples of correct answers follow:

```
public static void findStudentName(ArrayList<Student>a,
String s){
   String search = s.toLowerCase();
   for (int i=1;i<a.size();i++)
   if (((a.get(i))).getName().toLowerCase().
   startsWith(search)){
      System.out.println(a.get(i));
   }
}
public static void findStudentName(ArrayList<Student>a,
String s){
   int snsize = s.length();
   for (int i = 0; i < a.size(); i++) {
      String subname = a.get(i).getName().substring(0,
      snsize);
      if (subname.toLowerCase().equals(s.toLowerCase()))
      {
         System.out.println(a.get(i).getName());
      }
   }
}
```

Question 5: The *StudentsInList(String)* and *StudentsInList_2(String)* methods

Students were asked to handle errors in the above two methods. This meant adding try/catch blocks to handle the potential `FileNotFoundException;` and this should also have meant removing 'throws IOException' from the heading of each method. This was done well by some students but many mistakes were seen including:

- Not printing an error message if the catch block was triggered.
- Not handling all `FileNotFoundException` errors, for example, repeating the statement `Scanner in = new Scanner(new FileReader(s));` after the catch block. However, since 'throws IOException' had been kept in the heading of the method, the compiler would consider the exception thrown, and hence would not report an uncaught exception.
- Handling exceptions in one method, but not the other.
- Not successfully recovering from errors as follows:
    - If the catch block is triggered, the entire program is deliberately ended with `system.exit()`
    - If the catch block is triggered then the user is given an error message and the program hangs, waiting for input because after catch block, the method goes into the `while` loop, which tries to read in. **Note:** the `while` loop should be included in the try block.
    - The `Scanner` object has been initialised to null at the start of the method. This means that if the catch block is triggered, the program will end with a `NullPointerException` as the method

goes into the `while` loop after executing the catch block, and the `while` loop tries to use the `Scanner` object which is null. **Note:** the `while` loop should be included in the try block.

- If the catch block for either method is triggered, then the method will return null, causing a `NullPointerException` in any method that tries to use the arraylists. Better to return an empty `ArrayList` so as not to compromise the rest of the program.

Question 6: The *findStudentGrade(ArrayList<Student>,String)* method

Students were asked to write a method that would find all students with a particular grade, and print them to standard output. This was done correctly by the vast majority of students, with only a very few errors of logic seen, including the following:

- The method failed to include the final element in the `ArrayList<Student>` in the search.
- A boolean variable inside a `for` loop becomes true if the search `String` is found, once true the variable stays true. As the `for` loop iterates through the entire list, deciding whether or not to print every student name depending on the value of the boolean variable, this means that from the point in the list where the variable became true, the current element is printed, followed by the rest of the `ArrayList<Student>`, whether or not the grade is the one searched for.

Question 7: Binary search

Students were expected to implement the binary search method – from the CO1109 subject guide – and most did. Surprisingly, some implemented a linear search instead, and thus received no credit for their answers. One very common error seen was to write a correct method, but one that would search an empty array, triggering a run-time error (`IndexOutOfBoundsException`). **Note:** that the method given in the subject guide (Volume 2, page 56) would not search an empty array.

Numerous logical mistakes were seen with this method, mostly idiosyncratic without common errors. Often the result would always return true, or always false, whatever the input. Some methods could sometimes return a correct value, and sometimes not. Other methods would enter infinite loops, or would enter an infinite loop if the item was not in the array, but would terminate successfully if it was. Again, many of these logical errors could have been found and corrected with better testing.

Some mistakes were made because students had simply not engaged with the logic of the method in the CO1109 subject guide, copying it wholesale into their work, and not attempting to adapt it so that it worked as expected in their program.

Question 8: The *StudentsToFile(ArrayList<Student>,String)* method

Students were asked to write a method that would copy each element in an `ArrayList<Student>` into a text file, using the *Student* class's *toString()* method. Exceptions were to be handled, not thrown.

The most common error was writing a loop to write each element of the `ArrayList<Student>` in turn into a text file, but printing the entire arraylist at each iteration. Two different ways of achieving this were seen:

- Converting the entire arraylist into a single `String`, then printing the `String` with each iteration of the loop, using `System.out.println()`. The `for` loop would be set to iterate for the size of

the arraylist. At each iteration of the loop the entire contents of the arraylist are printed on one line.

- The method saves to the file using a `for` loop that repeats as many times as there are elements in the arraylist. However, with each iteration *toString()* is called on the entire arraylist.

Other common errors

- Throwing and not handling exceptions.
- Handling exceptions but without any error messages to inform the user of what has gone wrong.
- Using the *Student* class's getters to print the output, rather than the class's *toString()* method.
- Printing the entire arraylist on one line, rather than each element on its own line as specified.
- Writing statements such as: `System.out.println(student.toString());` Students were specifically asked to read the **whole** of Chapter 9 of Volume 2 of the subject guide. Had they done so, they would have discovered (section 9.9.6) that:
- *If x is an Object with its own toString() method then System.out.print(x) and System.out.print(x.toString()) will behave in exactly the same way.*

Question 9: *toString()*

You were asked to rewrite the *Student* class's *toString()* method in order to label the output to make it more readable. Various correct answers were possible, and almost no mistakes were seen. A couple of students deleted the spaces separating the output – making the output of the *Student* class harder to read, rather than easier.

Conclusion

Many of the errors described in this commentary could have been addressed with more testing. Some very good answers were seen, and these students must be congratulated on successfully engaging with a challenging coursework assignment. In some very good answers, students had rewritten the constructor in the `Student` class to issue warnings to the user if negative values for the *exam* and *cwk* variables were input; a very good idea and a definite improvement! Well done!