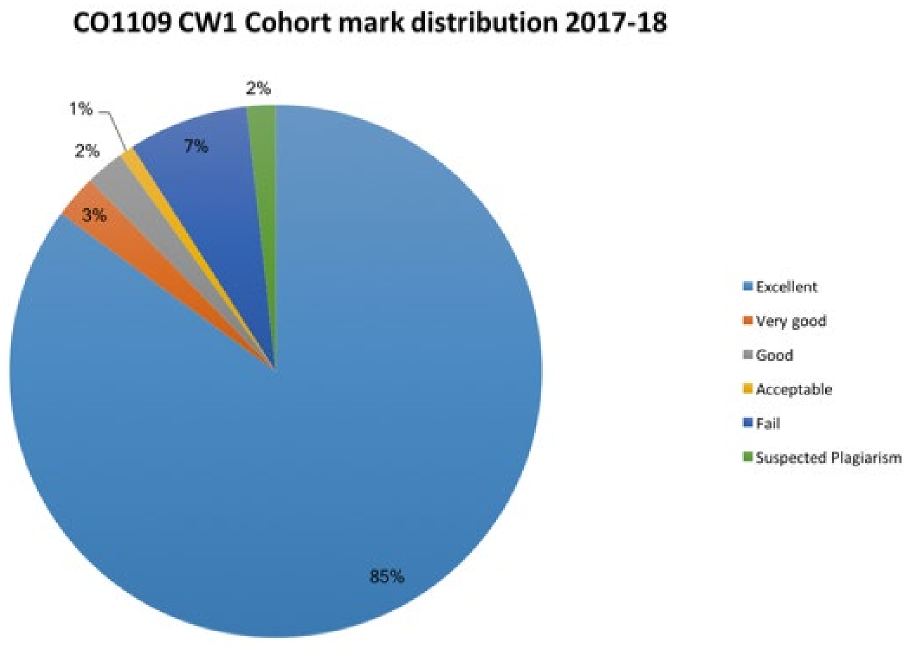
Coursework commentary 2017–2018

CO1109 Introduction to Java and object-oriented programming

Coursework assignment 1

General remarks

On the whole, this was attempted well. Many mistakes were seen collectively in the 352 submissions, and are commented on below, but it should be noted that most students made just one or two errors, and some made none at all, as the chart below shows:



Part A

Model answers

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

- *PartA_Answers.java*

The assignment

Students were given the file *PartA.java*. There were 12 methods in the file, named *loop1()* to *loop12()*. As their names suggest each had a loop, and each did not work exactly as it was supposed to. Students were asked to write comments in the file explaining why each of the methods in the file did not work, and proposing a solution. For each loop the comment should have been divided into separate PROBLEM, REASON and SOLUTION sections. Below we will list some common mistakes; followed by comments on particular answers. See the file *PartA_Answers.java* for model answers, remembering that most problems, even the very simple ones, have more than one SOLUTION.

Common errors in Part A

Students received no marks for giving the PROBLEM definition as something that the compiler told them, such as 'unreachable statement'. In order to receive credit students needed to explain why the unreachable statement was unreachable, e.g. the statement is unreachable as it comes after an infinite loop. In this case, the infinite loop is the problem, not the unreachable statement.

Some students could have achieved higher marks for part A by remembering that there are three types of error: syntax errors that the compiler finds; run-time errors that cause an exception to be thrown; and logic errors that mean our programs compile and run, but they do not do what we think they should do. Some students did not pay enough attention to errors of logic.

In general, part A was attempted well, although many students lost marks for lack of clarity or detail in their comments, particularly for *loop7(), loop8(), loop10()* and *loop11()*. In addition, many students did not understand the PROBLEM with *loop2()* and only a minority of students understood the PROBLEM with *loop12()*.

Below are some comments about the solutions to the 12 problems presented to students in part A, with some common errors.

loop1()

All solutions seen by examiners were correct. Most were straightforward, following the solution in the model answers, though there were a few idiosyncratic, but correct, answers, such as changing the stopping condition to `i > -20`

loop2()

The obvious solution to this was to give the `for` loop some parameters, so that the loop only executed 10 times, and this was the solution that every student gave. However, some made comments demonstrating a lack of understanding. A minority thought that the infinite loop would alternate between outputting `In loop` and outputting `Out of loop`, showing that they did not understand that since there are no brackets enclosing statements after the `for` loop, as far as the loop is concerned only the first un-bracketed statement is inside the loop. This was even more puzzling as the method could have been compiled and run, so that the students could see for themselves what the output was.

Some thought that part of the error was that there were no brackets enclosing one or both of the `System.out.println()` statements, and gave brackets as part of the solution.

loop3(), loop4() and loop5()

All students attempting these problems gave correct solutions. Some could have improved their marks by more clarity or detail in their REASON and/or PROBLEM comments. See the part A model answers for more.

loop6()

For this problem examiners accepted either "the loop does not start" or "there is a compilation error" as the PROBLEM definition. The simplest SOLUTION, which is always the preferred solution, was to make the guard of the `while` loop `true`, i.e. `while (true)`. Then the loop would start and continue until `i` was equal to `j`, triggering the `break` statement. One other solution often seen and accepted was `while (!(i==j))`. This second solution is not as simple, and does not make quite as much sense, since it means that the `break` statement becomes redundant.

loop7()

The PROBLEM was that the loop was infinite. When the loop started, it printed out a message about the values held by two `int` variables, and the screen quickly filled with text as the loop iterated. The REASON was that the loop would only stop when variable `i` was equal to variable `j`, but this was impossible. Most students lost marks for their REASON, because they did not clearly explain why it was that `i` could not equal `j`. Examiners were looking for students to make the point that the bound on the `Random.nextInt(int)` method is exclusive, meaning it is not included in the potential numbers that can be returned. Hence the statement `i = r.nextInt(j);` will mean that `i` can be equal to one of the numbers `0` to `j-1`, but it cannot be equal to `j`. Stating that `j` can never equal 10, or that the statement could only generate a number up to 9 was not enough for full credit. However, examiners did accept the statement "The maximum value r.nextInt(j) can return is j – 1" for full credit.

Different correct solutions were seen, such as:

- change `i = r.nextInt(j);` to `i = r.nextInt(j+1);`
- double the value of `i` before the comparison in the `break` statement
- add `i++;` immediately after the assignment to `i,` and before the `break` statement.

A large minority of students noted that there was a second problem, in that the `System.out.println("i = "+i+" and j = "+j);` statement will not be executed when `i` is equal to `j`, because when `i` and `j` are equal, the loop will have broken before the statement. The solution proposed, of course, was to move the statement to before the `break` statement.

loop8()

Quite a number of students lost marks when answering this question because they stated that the solution was to change the *stop* variable to `true`. It was unclear what was meant by this, as the *stop* variable was initialised at the start of the method to `false`, and made `false` again in the `while` loop. For full credit students needed to clearly write that the solution was to change the value held by *stop* to `true` in the `if` statement in the `while` loop.

loop9()

This was another problem on which students often lost credit for lack of clarity. A large minority of students simply said that the solution was to add `j++;` to the inner `while` loop, without stating that the `i++;` statement needed to be removed at the same time.

Some students also demonstrated a lack of understanding of nested `for` loops, by suggesting that the inner loop boolean expression should be changed to `j<50`, *i.e.* `while(j < 50)` in order to get the inner loop to run 50 times. As the comment given at the top of the method makes clear, the inner loop is supposed to run "5 x 10 times", *i.e.* 5 iterations of the inner loop, repeated 10 times.

loop10()

This method was supposed to draw a 9 x 9 square with asterisks ('*'), but actually drew a triangle. The triangle was drawn with nested `for` loops, the outer loop had the guard `i < k;` where `k` had a constant value of 10, hence the outer loop iterated 10 times, from `i` = 0 to `i` = 9. The inner loop made each line of the triangle (printed below with the value of `i` on each line), by printing asterisks equal in number to the value of `i`. The triangle had only 9 lines because while the outer loop iterated 10 times, on the first iteration the inner `for` loop printed nothing. This was because on the first iteration `i` is zero, `j` is also zero, and the guard is `j < i;` so `j` is not less than `i` and nothing is output.

```
i=0

i=1 *

i=2 **

i=3 ***

i=4 ****

i=5 *****

i=6 ******

i=7 *******

i=8 ********

i=9 *********
```

In order to print a 9 x 9 square, first of all the inner loop guard needed to be a constant value; the guard `j < k-1;` would give 9 asterisks per line. The outer loop guard should be changed from `i < k;` to `i < k-1;` so that only 9 lines are printed. Other solutions are possible, of course, a popular solution was changing the value of `k` to 9, so that the outer loop guard could stay as it was, and the inner loop guard could be changed to `j < k;`.

While all students understood that the guard of the inner loop needed to be modified, the most common mistake made was by students who did not spot that the outer loop would iterate 10 times, so the guard on the outer loop also needed to be modified.

There is also a very simple solution using only one `for` loop, which some students gave, which was to draw a line of 9 asterisks 9 times in a single `for` loop as follows:

```
    for (int i = 0; i < 9; i++) System.out.
    println("*********");
```

To get full credit for the REASON for this problem, students needed to note (1) that `i` was a variable that increased by 1 each time the inner loop was run, and this was why a triangle was drawn; and (2) that in order to draw a square, the inner loop guard needed to include a constant value.

loop11()

This method had a very simple SOLUTION, which many students found. Some did not, and gave much more complicated solutions, full credit was given to any solution that worked. Where most students lost credit was in describing the REASON for the problem, which was quite a challenging thing to understand and explain. The simple solution, found and given by some students, was that `x++;` should be changed to `x--;` The reason for this was to widen the space for drawing asterisks by 2 with each iteration of the loop. One space would come from incrementing *y* (already done in the method) and the other from decrementing *x*. This would mean that an extra asterisk was printed either side of a central point at each iteration, since the condition for printing an asterisk `[((j>=x)&&(j<=y))]` would be true more often, as *x* and *y* would no longer be equal to each other at every iteration of the loop.

loop12()

This was the problem that caused the most difficulties for students. Students struggled to understand and explain what the PROBLEM was, and only a minority gave correct solutions.

The method was searching for a `String` in a `String Array`, and the PROBLEM was that if the search `String` was in the `Array` the method returned true. If not, the method threw an exception. Many students said the PROBLEM was that the method threw an exception, but for full credit they needed to also say that the exception was only thrown when the search `String` was not in the `Array`.

Some students made the comment that the method should return a `boolean`. These students did not understand that the statement `return (a[i]==find);` will return `true` or `false`, which are boolean values. Students should have found from testing the method that it did return boolean values, since if it did not, the method would not compile and run.

Students also suggested that the "==" in the statement `return (a[i]==find);` would not work, and should be changed to `.equals().` These students should also have found in their testing that the "==" operator was working, and could not be the cause of the exception. Note that the `String.equals()` method will return true if two `Strings` are referencing the same value, even if they are not referencing the same object. The == operator, when comparing reference variables, will test to see if they both reference the same object. Hence two objects can be in the same state but not be equal as far as the == operator is concerned. However, with Java `String` literals the == operator is acting like `.equals().` See here for an explanation (scroll down): https://stackoverflow.com/questions/26199587/why-does-sometimes-work-for-Strings

A minority suggested in their solution using `try/catch` to return `false` if the `ArrayIndexOutOfBoundsException` is thrown. This answer did not receive full credit because `ArrayIndexOutOfBoundsException` is an unchecked exception, which means the compiler does not care if we do not throw it or handle it. This is because run-time exceptions are usually due to faulty logic, and the normal answer is to address the logic error that they represent in the code. There is more about checked and unchecked exceptions in CO2220.

Quite a number of students lost marks for their REASON, due to a lack of clarity. Many just said that the variable `i` would be incremented to 9 when the search `String` was not found and hence the error. This was not considered an explanation that demonstrated complete understanding of the problem, and so did not receive full credit. Some students also showed confusion about `Array` indexing, writing that the *greetings* `Array` was indexed from 0 to 9, and that the exception is thrown when looking for array item 10 in the return statement. In fact the *greetings* `Array` has 9 items, indexed from 0 to 8, and the exception is thrown when the return statement tries to access item 9.

One SOLUTION often seen was to change the first part of the guard from `i<a.length` to `i<a.length-1`; nothing else was changed. While many students gave this solution, it was very rare to see comments that demonstrated students' understanding of why this correction worked. In fact many students giving this solution wrote that the error occurred in the loop, rather than in the return statement.

This solution works because if the item is not found, `i` is incremented to 8, and then the loop fails as 8 is not less than the length of the *greetings* `Array` minus one, it is equal to it. Hence the final item in the `Array` is not compared to the search `String` by the `for` loop guard. This means that when the search `String` has not been found, the return statement will be comparing the final item in the `Array` to the search `String`, returning `true` if they are equal, `false` otherwise. So in the *greetings* `Array`, item 8 will not be looked at in the loop, but if the search item has not been found, will be looked at by the return statement.

The short circuit operators

A very small minority of students gave a solution that involved clever use of the short circuit AND operator. Java has four boolean operators, `&&;` `&;` `||;` and `|.` Two of these operators, `&&` and `||,` are known as *short circuit* operators, because they may not check both sides of an expression. The `&&` operator will first check the left side of an expression, if it is false it will return `false` and not evaluate the right side. Hence changing the return statement to

```
return (i < a.length && a[i]==find);
```

would not cause an exception. The `i < a.length` part of the guard would be checked first, and if it was false the method would return `false`. Only if it was true would the second part of the expression, `a[i]==find,` be evaluated. The examiners thought that this was a very elegant solution, but not the most readable.

These students are to be commended for being the only ones to explain that the short circuit operator is the reason why the `for` loop guard itself does not throw an exception. The `for` loop guard is `i<a.length && a[i]!=find;` Once `i` has been incremented to equal to the length of the `Array`, the left side of the expression evaluates as `false`, so the right side (the one that would cause the exception) is not checked and control moves to the return statement. This was an excellent point, although not necessary for full credit.

Part B

Model answers

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

- *CustomerVerifier.java*

The assignment

Students were given the *CustomerVerifier* class, which contained a number of methods with comments to help students to understand those methods. Students were given three tasks:

- Firstly to write a method *UserKnowsRandomCharsFromMemorableWord()*. Students were told that the method could be written with 3 statements plus statement(s) to return a value.
- Secondly students were asked to write a *verify()* method, implementing a quite complex user interaction loop.
- Thirdly students were asked to write a main method to test their class, which meant just a single statement calling the *verify()* method, although a minority made their main method more complicated than this.

Question 1

UserKnowsRandomCharsFromMemorableWord()

The method could be short because there were 3 methods in the class that could be used to do the necessary work: *getDiscreteRandomInts(); charsAt();* and *getMemorableWordCharsFromUser().* Most students wrote their method with statements to call these three methods, a very small minority wrote more complex methods with nested loops.

This method often saw logic errors, sometimes caused by students not understanding the methods they were calling. The *getDiscreteRandomInts()* method took two `int` parameters. The parameters needed to be given as parameters *x*, the number of `chars` from the memorable word wanted for verification purposes, and the length of the memorable word. The method would then return the indexes of *x* random `chars` from the memorable word. Since the question required that the user be asked for 2 random `chars` from their memorable word, the parameters given to the method should have been (2, <length of memorable word>). Mistakes seen here included: giving the method the parameters (2, 2) which meant that only the first two letters of the memorable word could be chosen, although in random order; and giving the method the parameters (2, <length of memorable word –1>) meaning that the index of the final `char` of the memorable word could never be returned by the method.

Question 2

verify()

The most challenging part of this assignment was writing the *verify()* method to implement the user interaction loop. Some students made no attempt at this part of the assignment, others only implemented part of the loop, while others wrote working but overly complex loops, and still others wrote excellent methods. In general those who managed to write a working loop are to be congratulated.

One issue was that many students used recursion to make their *verify()* methods iterate. This is hard to get right, and recursion can be very demanding on the memory. In addition a properly written `while` or `do/while` loop would be much easier to read and hence to test and improve. Despite this most of the recursive *verify()* methods did work, although sometimes, when the user tried to quit, the method at the top of the stack would hand back control to the method that called it, and this method might continue its loop from where it had been left due to an invalid entry, so might other copies of the method further down the stack. This would result in the user being asked for additional entries after they had quit.

Mistake: too complicated

Some students wrote user interaction loops in their *verify()* methods that were too complicated, for example a loop with 4 nested `while` loops, or one with many different `boolean` variables to the point that it was confusing to read and understand. Often these complex loops did not work properly, if written more simply they might have been easier to test and correct.

Mistake: testing

Many of the errors seen could have been found and corrected by the student before submission with better testing, for example the examiners found several loops that only worked given valid entries. If given an invalid entry these loops would either end, or would continue but give incorrect results from that point on.

Question 3

The main method

Most students wrote a main method with a single call to the *verify()* method, while a small minority wrote a main method with a menu for tests, all gained full credit. A few students lost marks for the main method by writing in the main statements that should have been in one of the other two methods they were asked to write.

Good: getPinFromUser()

Some students added try/catch blocks in order to recover from the `NumberFormatException` that would be thrown by the *getPinFromUser()* method if the user did not enter an `int`. This was not necessary for full credit but very good, nonetheless.