CSCU9A2 Practical 2A (Week 3) Spring 2017

Structured Development (30/31 January)

Remember to register your practical attendance at the start of each session:

- **Double click on the My Computer icon, then on Groups on Wide (the V: drive), then on CSCU9A2, and double click on the Register icon**

If you get stuck or need help at any time during the practical, ask a demonstrator.

Remember that these are teaching sessions, and during the practicals you should not be surfing the Web, nor texting nor chatting online with your friends.

Remember: This week's checkpoints must be completed by the end of week 5.

THIS WORKSHEET:

In this practical you will practice structured problem solving as an approach to developing Java programs.

STRUCTURED DEVELOPMENT

Introduction: One approach to *problem solving* that often works is the technique called "stepwise refinement" or "structured development". In this approach, a complex problem is first decomposed (or "refined") into a series of simpler subtasks – the subtasks are themselves not yet fully solved, but if they were completed they would clearly solve the overall problem. We then tackle refining *each of the subtasks independently*: for each we might decide that it is simple enough that we can produce a complete solution immediately; alternatively, we might decide that it is sufficiently complex that the best thing to do is to decompose it into a further series of simpler subtasks. This process continues until there are no subtasks that have not been solved.

The web page at **http://c2.com/cgi/wiki?StepwiseRefinement** is interesting to read, and shows a stepwise refinement of an initial problem "Brush teeth".

Stepwise refinement in programming usually works like this: The goal is to produce a *method* to solve some problem. Assume the problem is reasonably complex, so the method's body is a decomposition of the problem into simpler steps, each of which is *a call of a further method* that will solve a particular subtask. At this point, declarations of those further methods can be added to the program with method headers that match the way the method is called, but with *empty method bodies* (just a pair of {}); for each method we will also have a description of what its purpose is (preferably as a comment in the program). We then tackle each method separately: for each method, perhaps we can design the full Java code for its body reasonably easily, but alternatively, if the method's purpose is sufficiently complex, then we may refine its body to a series of further method calls (and also adding corresponding empty method declarations to the program). When all methods have fully programmed bodies, then the program is complete – it might even have been partially executable at intermediate stages.

You are going to carry out a stepwise refinement of a problem based on one that appeared in Horstmann's lecture slides on arrays and problem solving: *The program reads a student's marks into an array, discards the lowest mark, adds up the remaining marks and displays the average.* [Discarding the lowest mark before averaging is a neat way to ensure that one bad piece of work does not affect the final outcome.]

You can carry out this worksheet using either BlueJ or at the command prompt with Textpad/Notepad, etc. It is up to you!

- Open **My Computer**, and then open the **CSCU9A2 Groups on Wide** folder, and then the **Java** folder. Take a copy of the file **Marks.java** into your own file space in a suitable new folder.
- **Marks.java** will *not* compile as it stands, as it has methods missing.
- Open **Marks.java** and look at what is there:
  - There is a **main** method that asks the user how many marks are to be processed, then declares an array of the correct size to hold the marks. The method body then contains the main processing in the form of calls of three further methods whose names indicate their function: Read the marks, discard the lowest, and calculate the average – and it should be clear that they identify three processing steps that together solve the overall problem. *This method is complete and correct.*
  - At the end of the program is a standard **readInteger** method, with the addition of a parameter that is used as an initial prompt to the user. *This is complete and correct.*
  - In between is the header for one method called from **main**, **readMarks**, but with an empty method body. This method placeholder is here because it is a subtask introduced in the refinement of the **main** method body. At a later stage, its body will be completed.
- Now for some work:
  - Add two further placeholders for the other methods called from **main** – the method headers should match how the methods are called in **main**. The descriptions of those two methods as devised by the designer of **main** are:
    discardLowestMark*: Assume that the given array is full of marks (integers). The lowest mark is discarded by moving all the following marks down one element in the array . This leaves valid marks in all except the last element of the array.*
    averageTheMarksExceptTheLastOne: *Add all the marks in the given array together, excluding the very last array element (as that element contains junk left over from discarding the lowest mark). Return the average as the result.*
    The program might now compile, but doesn't do much yet!
  - There are three outstanding tasks: three methods with empty bodies.
  - Focus on method **readMarks**: You can write the Java for **readMarks**' body quite easily (it's **readArray** from previously). Compile, and **readMarks** should run correctly. [But how can you test that it runs correctly? One possibility is to temporarily add into **main,** after the call of **readMarks,** a short **for** loop that outputs each array element to **System.out.println]**;
  - Now focus on method **discardLowestMark**: That looks a bit tricky, so we'll refine it into two further subtasks (without coding their full details just now). The two steps are: work out where the lowest mark is, then remove it. Insert this into the body of **discardLowestMark**, which introduces two further methods:

```
int position = findLowestMark(marks);
removeMark(position, marks);
```

  - The add two placeholders for the two new methods. You'll come back to them later. Their descriptions are:
    findLowestMark*: Search the given array of marks, and return the index of the lowest mark (or an equal lowest mark).*
    removeMark*: Remove the value at index position in array marks by moving all further values down one element.*

- Now focus on method **averageTheMarksExceptTheLastOne**: You can write the Java for the method body quite easily – a loop to add, and then calculate and return the average (the principle was in a previous practical). Compile and run: you should be able to enter marks and see some kind of average (but, of course, it is not the proper required answer, as the method to discard the lowest mark currently does nothing!).

Remember: Your code must be well formatted and clearly commented.

Checkpoint [Structured Development]:

Show a demonstrator your code up to this point for the Marks program. Demonstrate it running and outputting the average of all marks entered. Explain why your program contains empty-bodied findLowestMark and removeMark methods. Answer any questions they ask you.

- Finally, focus separately on the two remaining methods: **findLowestMark**, and **removeMark**, and program their method bodies. You may need to view CSCU9A1 material for ideas here.
- Compile, run and test carefully.

Remember: Your code must be well formatted and clearly commented.

SBJ 27 January 2017