CSCU9A2 Practical 4B (Week 5) Spring 2017
Testing and debugging 16 February

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

Remember: Week 3's checkpoints (sheets 2A and 2B) must be completed by the end of this week. You have until the end of week 8 to complete this week's checkpoints.

THIS WORKSHEET:

*In this practical you will get some exercise "provoking" and tracking down the bugs and imperfections in a program that **almost** works. It is also a program that uses an array in a special, though quite common, way to hold a collection of numbers that does not necessarily fill the array.*

TESTING AND DEBUGGING

When developing computer software, before delivery to a "customer", we must test what we have produced to make sure that it meets the customer's requirements. This involves the careful choice of test data (and usually patterns of events) to make sure that all parts of the program have been exercised in all circumstances that they must eventually deal with, and have been seen to work properly. If a test reveals some "non-conformance" with requirements, we must then "debug" the program: investigate it to discover where something is going wrong, and then correct it. This practical shows you some of the skills and techniques to be applied in testing and debugging — they may be of use to you in the assignment!

DEBUGGING USING BLUEJ

In earlier practicals you have used additional `System.out.println`s to display information from inside your running program so that you could monitor what was happening — with appropriate choice of diagnostic statements, you could see precisely what path execution was taking through the program, and also what data the program was dealing with. Those techniques are more or less "universal" — they work in virtually any programming language in almost any situation.

However, many IDEs (Integrated Development Environments), and some independent tools too, offer better facilities for monitoring what is happening inside a program that are especially useful if the program is not producing the correct results. BlueJ has such an "interactive debugger". Essentially, the programmer can indicate one or more specific points in a program ("breakpoints") at which execution should temporarily be halted; when the program is halted at a breakpoint a "debugger" window allows the programmer to see where execution has halted, how execution arrived at that point, and what the values of variables are at that point in time. The debugger then allows the programmer to step through execution of the program one statement at a time, to add/remove breakpoints, or to continue the program's execution until the next breakpoint (or the end of the program) is reached. [Other debuggers offer more sophisticated possibilities in addition.]

1. First you need to create a new basic application from an existing file as you have done before:

   In a new folder in your CSCU9A2 working folder: Create a new BlueJ project using the file **Averages.java** from **Groups on Wide (V:)\CSCU9A2\Java**

   The program consists of a main class **Averages**, which shows a text field for entry of integer data items (when **Enter** is pressed), and a text area for displaying results.

   Compile and run the program — *it doesn't work very well yet, so don't worry, and read on*.

2. The program is ***intended*** to work as follows:

   - A collection of up to 10 integers is held in an array (called `data`).
   - The current collection of integers is displayed on the screen (in a text area called **`displayArea`**) by method **`doDisplay`** which is called from the method **`setUpdata`** and from the **`actionPerformed`** method. Most of your work will be in the **`doDisplay`** method.
   - The average *of **only** those integers in the list that fall in the range 10–20 **inclusive*** is displayed (so, for example, if the list of numbers is 11, 9, 13, 125, then the average should be (11+13)/2, that is 12).

- The user can add another number to the end of the collection by typing it into the text field and typing the **Enter** key — so the program also illustrates how it can be an **`actionListener`** for a text field and how **`actionPerformed`** will be called when the user presses **Enter** in that text field.

As given to you, the program is set up with the two numbers 15 and 17 in the collection (see the array initialization in the **`setUpData`** method). **Actually the program contains bugs**, so it *doesn't* do what it is supposed to. *Even if you think that you can see all the problems immediately, there is tuition here in the choice of test data, and debugging techniques, so please "go through the motions".*

Special note about the use of the array: The program uses an array called `'data'` of 10 (`'size'`) integer elements to hold the current list. To provide for the situation where less than 10 numbers are actually being held, the global instance variable `top` is used to indicate the index of the 'last' actual data item held in the array: so, if there are currently four integers in the list, they will be in elements `data[0], data[1], data[2], data[3]` and `top` will contain the value `3`. So, to add another data item to the *end* of the list, the following pair of statements is used:

```
top++;             // Adjust index to next element
data[top] = ... ; // Insert value
```

(This occurs three times in the program.)
For consistency, this means that if there are **no** integers in the list, `top` must be `-1`, which is how it is set up initially. *This way of using of an array is extremely common, and is worth understanding well.*

3. Run the program. One problem should be immediately apparent: we know that the list contains 15 and 17, but the display shows only 15, and the average displayed is 15, not 16 as it should be. Since both the display and the average are omitting the 17, perhaps it is a similar error. Close the running program.

   (a) First, let's tackle the display of the list. It is produced by the first `for` loop in the method `doDisplay`. We really need to know whether, firstly, the `data` array actually contains what we think it contains, and secondly, whether the loop even attempts to display both the items in the list (perhaps both are displayed, but something else clears one from the screen!).

   You can find this out by setting a "breakpoint" at the loop that inserts the numbers into the text area display. When the program is stopped at this point, you will be able step execution through the loop to see exactly what it does.

   Open **Averages** in BlueJ's *editor window*. Down the left hand side is a narrow margin, separated from the Java text by a line. You can set BlueJ to display line numbers in this margin, via **Options** menu/**Preferences — do that now**. Once the program has been compiled, if you click *once* in the margin to the left of a Java statement then a red Stop sign appears, indicating that a breakpoint has been set at that line (clicking *once* on a Stop sign removes it): when the program is running, program execution will halt *just before* executing that line of Java.

   Locate the following `for` loop (line 157 in the program):
   (Hint: Type Ctrl-L and then the line number, and BlueJ will skip directly to that line)

```
X     for (int i = 0; i<top; i++)
        displayArea.append(data[i] + "\n");
```

   Click next to the line marked here with an X (the line numbered 157) to set a breakpoint just as the `for` loop is about to start.

   (b) Before running the program, it will be convenient if you *move the editor window* so that it occupies the *top left of the screen*.

   (c) Now run the program in the usual way. `doDisplay` is first called at the end of `setUpData`, *as the program is starting up*. As Java is executing `doDisplay` it will very quickly reach the breakpoint at line 157: BlueJ's debugger window will appear on the screen, and the line containing the breakpoint will be highlighted yellow in the editor window. Rearrange the windows so that you can see the editor (upper left), and the debugger window (towards the right of the screen is convenient). The start-up process is not yet complete (the JVM has not reached `frame.setVisible` yet), so the application's main window is not yet visible on the screen!

(d) The debugger window contains many small panes. You can adjust their relative sizes by dragging their boundaries. For this practical, the "Threads" and "Static variables" panes are *not relevant* and you can shrink them to their minimum. The "Call sequence" pane shows, at the top, the name of the method in which execution is stopped (useful information), then the name of the method that it was called from, and so on (the lower lines often refer to JVM library code, so are not very useful). The "Local variables" pane shows the parameters and local variables of the current method (there are none at the moment). The "Instance variables" pane shows global variables — it contains many items of data used by the JVM for managing the `JFrame` that are not useful to us, but if you look carefully (scrolling if you need to) then you can find the variables `size`, `data` and `top` that are global instance variables in the **Averages class.**

(e) The breakpoint has stopped the program just before starting the `for` loop: line 157 in the editor window is highlighted yellow indicating that **that** statement is to be executed ***next***.

(f) Look at the Instance variables pane in the debugger window. Scroll to find the information about the `data` array – it should say: `private int[] data = <object reference>` Double-click on that line and BlueJ will open an "object inspector window" to allow you explore and view the contents of the object (an array in this case): You can see that the two values 15 and 17 are there in the elements with indices 0 and 1, which is exactly as expected. Close the inspector.

(g) Now to trace through step by step to see what is sent to the text area for displaying:

Click the **Step** button in the debugger window ***once*** — this causes the debugger to allow your Java program to execute (roughly) one more statement, and to then stop it again for inspection.

Look carefully: The loop body statement `displayArea.append...` is highlighted in the editor window, showing that execution has entered the loop body, and the *Local variables pane in the debugger window* shows that now `int i` has the value 0. That is good: the first array element (15) is about to be added to the display text area (which as yet is still not visible).

When carrying out this kind of process, you need to take it slowly, you must cross check carefully all the time between the highlights in the editor window and the variables in the debugger window, and you need to think about ***what ought*** *to be the case, and see whether* ***it is*** *the case.*

Click the **Step** button in the debugger window ***once***. The highlight in the editor window shows that execution has completed the loop body and has moved back to the `for` loop control line – OK so far.

The next time that you click **Step**, execution *should* move into the loop body to deal with the *second* element of the array (the value 17, which is in the element with index 1). Click **Step** ***once*** and notice what happens in the editor window: Java has exited from the loop *without* carrying out the loop body for the second time! The yellow highlight is positioned at the declaration `int total = 0;` which is the first Java statement ***after*** the loop.

We have seen enough for now. We need to think for a bit. Click on the **Terminate** button in the debugger window to stop the program running.

(h) Here is what we should conclude: The loop only attempted element `0`, rather than both `0`, and `1`. The loop is stopping *one element too early*, and we should *look at the loop repetition* ***test*** *carefully*. The answer here is that the programmer has treated the variable `top` rather like the *size* of an array, and the loop condition is `i<top`, whereas `top` holds the index of the last *actual data item* — so the loop condition should be `i<=top`, so that the last element *is included*! Make this change, and, while you are doing so, notice that the programmer has made the same mistake in the averaging loop a couple of lines further down. Correct it there too.

(i) Compile and run again, without setting the breakpoints again. This time execution runs through the launch process without being interrupted, the main window appears, and both items should be displayed, and the average should be 16. Excellent — *one bug detected, tracked down, and two bugs removed!*

4. Now try running the program and *using the text field* to enter three extra items of *typical, easy input data, with all the numbers within the range 11–19* (that is, well within the range to be considered by the program). For example, enter 16 (then **Enter**), 18 (then **Enter**) and 19 (then **Enter**), giving the list

```
15 17 16 18 19
```

(though appearing in a vertical column on the screen).
The average *should be* 17. **You should *always* work out what the answer *should* be before running the test — so you don't get biased by what you see!** The result for this test should be OK. *Close the program.*

5. So now we can try running the program again with typical, *"not so easy"* input data, with some numbers *outside the range 10–20*: type into the text field 8 (**Enter**), 99 (**Enter**), 16 (**Enter**), giving:

```
15 17 8 99 16
```

What *should* the answer be (work it out by hand)? What answer is actually displayed? Oh dear… time to stop and think again.

(a) The average is wrong, but it is not immediately obvious how the average reported is related to the numbers entered. It would be useful to know what is going on *inside the averaging process* in `doDisplay`. As a first guess we might wonder what the values of `count` and `total` are *at the point where the average is calculated* (since all we can tell at this point is that the average turns out wrong). We can find out what values they have at this point by setting a breakpoint at the assignment (line 172):

```
float average = (float)total/count;
```

Remember that the debugger will stop the program *just before* executing the statement at the breakpoint.

(b) Run the program *again using the test data that failed*. Each time that the program attempts to calculate the average, the debugger will stop the program. The first time is when the program is starting up — we are not interested in seeing anything at this stage, so click **Continue** in the debugger window — this causes the debugger to allow the program to continue running normally until it next reaches a breakpoint. The display with 15 and 17 and their average will appear. Move the program's main window, now visible, to near the bottom left of the screen. Now enter the next item of data, 8, press **Enter**, wait for the debugger, then click **Continue** — the display should show 15, 17, 8 and their average. Then enter 99, press **Enter**, wait, then click **Continue**. Then **slow down**: enter 16 and press **Enter**.

The debugger has stopped the program *just as it is about to calculate its final average* for the data 15, 17, 8, 99, 16. Look carefully at the *local variables* in the debugger window:

What do you observe about the values of `count` and `total` displayed in the debugger window?

`total` seems to be OK (it should be 15+17+16), but `count` has counted *all* of the numbers in the list rather than *just the ones that have been added together* — so it is not surprising that the displayed average is wrong! Hmm. *Time to look carefully at how `count` is calculated – see the next step.*

(c) To help you with this step: After a little thought, we see that the problem is that the step in the second `for` loop body which increments `count` (line 168) is *always* carried out — it should actually be carried out *only* when we add `data[i]` to `total`. Solution: the assignment to `total` and the `count++` must be grouped together with { and } to make a *block*, so that the `if` statement either executes them *as a pair* or not at all. [Note that the statement that increments `count` is correctly indented, as if the programmer knew what they wanted, but, of course, the Java compiler itself ignores all the indentation!]

Make this change to the program, save, recompile **and try the same test data again**. All OK now?

6. So, the program has passed the tests with "typical data". The next step is to try it with *boundary data* — in this case that means including data that is on or near the boundaries of being included or excluded. The special values are 10 and 20 — the program is supposed to *include* these, but *exclude* 9 and 21. Here are some separate test data sets to try to see what happens; *work out for yourself what the average is supposed to be in each case*, ***only then*** run the program once to try each data set and check the results:
   - `15 17 20 5 18` (this tests that the exact upper limit is *included* in the average)
   - `15 17 10 12 0` (this tests that the exact lower limit is *included* in the average)
   - `15 17 9 21 0` (test that values just outside the upper and lower limits are *excluded*)

   Well, how did it work out? There's still a bug, isn't there?

   (a) Now is probably the time to find out exactly what is happening to `count` and `total` inside the `for` loop that adds the numbers up. Add a breakpoint *just before the start of the totalling loop*: line 162 is a good place (the declaration of `total`). This will allow us to click **Continue** if we are not interested in the totalling process, and to click **Step** if we wish to see exactly which numbers are added to the total.

   (b) Run and try the test data from step 6 again. In each test, you can click **Continue** at each breakpoint *before* entering the last number in the data set. *After* entering the last number, click **Step** repeatedly to see the totalling loop at work: Look carefully at which element index values (`int i`) cause the statement
   `total = total + data[i];` to be executed.

   Is the program accepting and rejecting the correct numbers? No, it's not — it appears to be excluding 10 and 20 from the total instead of including them, *so there must be a problem with the `if` test*. Look at the program to find out where the bug is; correct it; recompile ***and try all the test data again – this is very important***.

7. Checkpoint [Test/debug]: Now demonstrate to a tutor your program correctly handling the test cases in step 6, show where you inserted the breakpoint, and explain how stepping helped you to identify the problem, and explain the correction that you made.

8. You may now be happy with your program; you may feel that you have tried everything, and are ready to deliver the program to your customer.

   But **Hold On**, you have missed a *very important* special test case: when **none** of the integers entered is in the range 10–20! Alter the `setUpData` method so that the first two elements of the array contain 9 and 21 instead of 15 and 17.

   Compile and run. Oh dear, oh dear… this is worse than the program simply giving a numerically wrong answer. The answer is plainly ridiculous — your program has placed some junk in `average` and then displayed it on the screen! [By the way: `NaN` stands for "Not a Number"!] That is, the calculation that produced a value to assign to `average` went wrong. Think about why. (Hint: What would the value of `count` have been? What is the problem with dividing by this value?)

   Perhaps we need to fix this problem too. However, you might well reason that this kind of data is simply ridiculous, the program was only designed to deal with sensible sets of numbers — and so you could deliver the program as it is to the customer with a warning in small print. But is that responsible? It is probably better to make the program data-proof (and idiot-proof?).

   (a) Correct this bug too. Hint: you'll have to avoid computing and outputting the average when `count` is zero (it's not too hard once you realise that it needs doing).

   (b) Correcting the 'no values in the range' error also means that you can safely remove the two lines from `setUpData` that set up the initial two test values (they were rather artificial in the first place — only there to avoid this problem at beginning). Compile, run, and *test again with all the test data sets to make sure that your final correction did not undo any earlier corrections*. [This re-testing is called Regression Testing, and is very important.]

9. Finally for the **Averages** exercise, the program ought to be protected from the entry of genuinely ridiculous data. There are two kinds: non-integer entries into the text field, and too many values entered for the size of the array.

(a) Try entering a non-integer into the text field: a word, or maybe a number with a decimal point. After hitting **Enter**, probably nothing will happen in the program window, but an awful message mentioning

```
java.lang.NumberFormatException
```

will appear in BlueJ's terminal window (you might need to scroll up to see the very first line). This will be followed by more lines that are less useful, one of which (probably the 5th) indicates that the exception occurred

```
at Averages.actionPerformed (Averages.java:138)
```

This tells you the method in which the "exception" (error) occurred, and the file containing that method. The 138 is the line number where the problem occurred. It might not be exactly 138 for your program, but it will indicate the line with a call of `Integer.parseInt` on it. It is the call of `Integer.parseInt` that has failed — not surprisingly. You can click on the "`at Averages...`" line and BlueJ will show you the line of Java in the editor window.

If you carry on, and enter a valid integer, you will probably see that a zero has been recorded when you entered the invalid data; that is more or less accidental and benign here. You do not yet know how to solve this problem — it needs the use of an *exception handler* — this will be discussed in a later lecture.

(b) Try entering more than ten integers. After you hit **Enter** for the eleventh, an awful message like this

```
java.lang.ArrayIndexOutOfBoundsException
```

will appear in the terminal window (followed by several more lines that are less useful). The last word on the first line gives a big clue: `'ArrayIndexOutOfBoundsException'` — the program has attempted to access an array outside its valid index range. That is not surprising. One of the later lines tells us exactly where it happened – look at that line. Your program needs to avoid processing the integer entered by the user if `top` already has the value `size-1`. Try to correct this too.

10. Your program should now work perfectly!

SBJ February 2017