

CMSC 113: Computer Science I

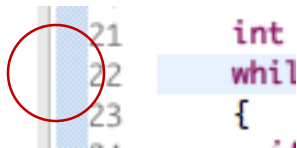
Lab #6: ArrayLists

Part I: Debugging

Before working on problems involving lists, we'll start with a brief tutorial on using the Eclipse debugger. The Eclipse debugger is a powerful feature of Eclipse that allows you to inspect your programs as they run, watching for where they go wrong.

1. From the course syllabus page, download *Imperfect.java* and load the file into Eclipse. (Depending on how things are set up, it may be easiest for you to create your own *Imperfect.java* file in Eclipse and just copy and paste my contents into your file.)
2. Run the program, say with the input of 6. You should see an error (`ArithmeticException`) get reported.
3. This program needs to be debugged! Clearly, the place to be worried about is the loop. So, we want to *break* the program at the beginning of the loop. To break a program means to stop its execution so that we can examine what it's doing, in slow-motion.


To set a *breakpoint*, double-click in the left margin of the editor window:

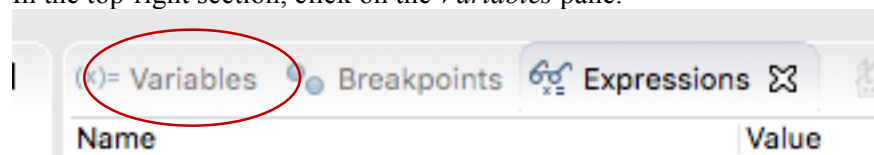


After double-clicking, it should look like this:

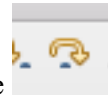


That little blue dot is called a breakpoint.

4. Now, click to debug your program by pressing , to the left of the usual *run* button.
5. Your program will start running as normal, asking you to enter a number. Enter 6.
6. Then, Eclipse will ask you to *Confirm Perspective Switch*. Say *Yes* (and you may wish to tell Eclipse to *Remember my decision*).
7. Eclipse will reconfigure its views. When you see this display, it means you're in the middle of a debugging session.
8. In the top-right section, click on the *Variables* pane.




9. You will see all your variables. For example, you should see that `num` is 6 while `i` is 0.



10. To make your program move forward by one step, click the *Step Over* button. (It's in the normal toolbar toward the top of Eclipse.)

11. You'll see the highlighted line select the `if(num % i == 0)` line. Step once more and you'll see the error message being generated. But now you can answer why this happened: `i` was 0.

12. Now that you've solved the problem, abort the debugging session by clicking the usual  *Terminate* button (near the bottom of Eclipse).



13. Return to the normal display by clicking *Java Perspective* near the top-right corner of Eclipse.


14. Make sure you're editing *Imperfect.java* and fix the problem by starting `i` at 1, not 0.

15. Debug your program again; it will stop at the breakpoint (after you enter in 6 as your number).

16. Press *Step Over* several times. You'll watch the program in action. Study the behavior in the *Variables* view (on the top-right). Is this what you would expect? Do you see what's going wrong? How can you fix this?

17. You'll find two more bugs in this file. Fix this and watch how the program works correctly in the debugger.

A few more debugging tips:

- Next to *Step Over* you'll find , *Step Into*. The difference between these is that *Step Over* tries to get to the next line in the current method, while *Step Into* will move the highlighted line into any methods that are called on the current line. (If the current line does not contain a method call, *Step Over* and *Step Into* behave identically.)
- If you want the value of something that's not a plain variable (say, `num % i`), you can enter the expression in the *Expressions* pane, an alternative to *Variables* in the top-right section of Eclipse.
- When you have a list, Eclipse will allow you to see the list contents by clicking an arrow that will appear in the *Variables* pane. (In this program, the `in` variable has such an arrow, but the information stored in a `Scanner` is not of interest to us.)

The best way to use the debugger is to make every time you click *Step Into* or *Step Over* a tiny scientific experiment. Before clicking, form a hypothesis about what you expect to happen. Then, after clicking, check whether your hypothesis is confirmed. If it's not, you've learned something new that might lead you to your bug.

Part II. Programming with lists

Complete the problems below. Your output does not need to match the sample exactly – the samples are to help you understand the task before you.

1. Write a console program that asks the user to enter numbers. It should continue to accept numbers until the user enters 0. At that point, it reports back all the numbers that had been entered. Here is an example session:

```
Enter a number: 3
Enter a number: 5
Enter a number: -4
Enter a number: 1
Enter a number: 0
You entered 3, 5, -4, 1, and then 0.
```

2. Write a console program that asks the user for the number of values he or she will enter. Collect that number of values. Report back the values entered and their arithmetic mean (average). Here is an example session:

```
How many values? 5
Enter a value: 3
Enter a value: 9
Enter a value: 8
Enter a value: 5
Enter a value: 10
You entered {3, 9, 8, 5, 10}; the average is 7.
```

3. Write a console program that asks the user for the number of values he or she will enter. Collect that number of values. Print out the values entered and then report how many positive values and how many negative values there were. Here is an example session:

```
How many values? 6
Enter a value: 8
Enter a value: 0
Enter a value: -3
Enter a value: -3
Enter a value: 2
Enter a value: 5
You entered {8, 0, -3, -3, 2, 5}; 3 were positive and
2 were negative.
```

4. Write a console program that asks the user for the number of values he or she will enter. Collect that number of values. Report back the values and state the index of the value that is the smallest. Here is an example session:

```
How many values? 7
Enter a value: 78
Enter a value: 35
Enter a value: 90
Enter a value: 96
Enter a value: 56
Enter a value: 83
Enter a value: 77
You entered {78, 35, 90, 96, 56, 83, 77}; the minimum
was stored at index 1.
```

5. Write a console program that asks the user for the number of values he or she will enter. Collect that number of values twice. Report back both sets of values and the set of values that results when corresponding values are added together. This operation is an example of *zipping*. Here is an example session:

```
How many values? 4
You will now enter the first set.
Enter a value: 3
Enter a value: 25
Enter a value: 88
Enter a value: 17
You will now enter the second set.
Enter a value: 7
Enter a value: -25
Enter a value: 2
Enter a value: 17
You entered {3, 25, 88, 17} in the first set.
You entered {7, -25, 2, 17} in the second set.
The zipped sum is {10, 0, 90, 34}.
```

6. Write a console program that asks the user for the number of values he or she will enter. Collect that number of values. Print out the numbers that are perfect. Here is an example session:

```
How many values? 9
Enter a value: 35
Enter a value: 6
Enter a value: 17
Enter a value: 11
Enter a value: 28
Enter a value: 99
Enter a value: 23
Enter a value: 16
Enter a value: 9
You entered {35, 6, 17, 11, 28, 99, 23, 16, 9}.
Of those, {6, 28} are perfect.
```