

CMSC 206: Data Structures

Lab 6: Linked Lists

Part I: Debugging

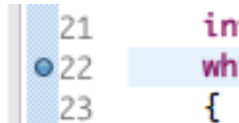
Before working on problems involving lists, we'll start with a brief tutorial on using the Eclipse debugger. The Eclipse debugger is a powerful feature of Eclipse that allows you to inspect your programs as they run, watching for where they go wrong.

1. From the course syllabus page, download *KWArrayListBuggy.java* and *ArrayListTest.java* and load these file into Eclipse. (Depending on how things are set up, it may be easiest for you to create your own files in Eclipse and just copy and paste my contents into your files.)
2. Run the tests. You should see an `ArrayIndexOutOfBoundsException` get reported.
3. This program needs to be debugged! The `ArrayIndexOutOfBoundsException` tells us that it is triggered in the `reallocate` method of `KWArrayListBuggy`. That means that we want to see what is going on inside that method. To do this, we set a *breakpoint*, which tells Eclipse to pause execution of our program at a certain point.

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




After double-clicking, it should look like this:

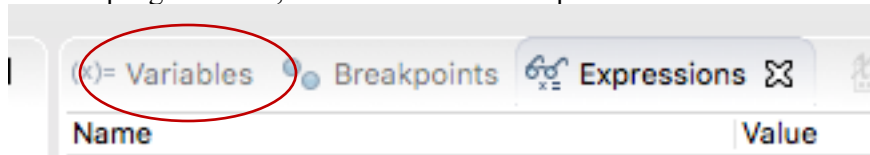


That little blue dot is called a breakpoint. In our case, set the breakpoint on the first line of `reallocate`.

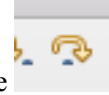


4. Now, click to debug your program by pressing , to the left of the usual *run* button.
5. Your test will start running as normal.
6. Then, Eclipse will ask you to *Confirm Perspective Switch*. Say *Yes* (and you may wish to tell Eclipse to *Remember my decision*).
7. Eclipse will reconfigure its views. 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 `this` is a `KWArrayListBuggy`. If you click on the arrow to the left of `this`, you'll see the fields of the class, informing you that `capacity` is 10, `size` is 10, and `theData` is an array.



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 `for`-loop line. Keep stepping until you see the error. Why does the error happen?

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. Update the code to fix the problem and re-run the test.

15. You'll find several more bugs in this file. Fix these 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, `i + 1`), 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.
- Though it's more advanced, Eclipse supports *conditional breakpoints*, which trigger only when a certain condition holds. Visit the *Breakpoints* tab (next to *Variables*) to set this condition.

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: Linked Lists

This lab is meant as an opportunity to explore linked lists. This lab refers to code in the `SingleLinkedList` class studied yesterday in class.

1. Write a sample main program using `SingleLinkedList` to create a list of strings. Add several strings to it (with `addFirst`), and use the `toString` method to render a string to print.
2. We looked at how `addFirst` works to add a new node to a list. Do the same with `addAfter`; that is, draw a diagram (with arrows, etc.) of a linked list and model how `addAfter` will change the list. Include both a "before" and an "after" in your drawing.
3. Repeat this exercise with `removeFirst` and `removeAfter`, drawing "before" and "after" diagrams for each.
4. Read the code for the two-parameter `add` method. See how it uses `getNode` and `addFirst/addAfter` to add an internal node in the list. Trace through how this would add a new string to the list of strings you created in your main method.
5. Write a new method according to the following:

```
/** Remove and return the item at a known index
 * @param index The index of the item to be removed
 * @return The removed item
 * @throws IndexOutOfBoundsException if the index is invalid
 */
public E remove(int index)
```

This method should use `getNode` and work similarly to the two-parameter `add` method. Drawing diagrams may help you to understand this better.

6. Write a new method according to the following:

```
/** Remove the first occurrence of element `item`
 * @param item The item to be removed
 * @return true if the item is found and removed;
 *         otherwise, false
 */
public boolean remove(E item)
```

How should you be determining whether one item equals another? (Hint: `==` doesn't work.)

Drawing diagrams may help you to understand this better.