CS 61B Lab 7
March 1-3, 2000

Goal: This lab will introduce you to gjdb, the Java debugger.

Copy the Lab 7 directory by starting from your home directory and typing:

```
mkdir lab7
cd lab7
cp $master/lab/lab7/* .
```

The Java Debugger Tutorial

Welcome. You now find yourself in the position of debugging somebody else's bad code in DebugMe.java, which is supposed to compute the partial geometric series

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \qquad \text{(which equals 1.875)}$$

DebugMe.main() begins by calling the recursive method createGeomSeries(), which creates a linked list in which each node contains one term of the above series. Next, main() calls the recursive method listSum() to compute the sum of the nodes in the list. As you may have guessed, it doesn't work. If you compile and run DebugMe you will see that it prints the value 1.0 instead of 1.875.

Some documentation on the java debugger (gjdb) is available in the course reader and is also linked from the class Web page. (The Java debugger provided with Sun's Java Development Kit is called jdb, and it is not very good; Professor Hilfinger modified it and built a better interface for it.) Here is a summary of how you might use gjdb to track down the error in this program.

(1) Compile the program with the debugging switch '-g'.

```
javac -g DebugMe.java
```

(2) Instead of running java, you could run gjdb as follows.

```
gjdb DebugMe
```

When you see the '(gjdb)' prompt, the debugger is ready to go, and you may follow the instructions in the course reader on how to use it. However, rather than run gjdb from a shell, we recommend you run it from emacs. Emacs will interpret the output of the debugger and show you where the debugger is in your .java files, which is extremely useful. To begin, type 'Meta-x gjdb' from within emacs. When asked to fill in the gjdb command, use 'gjdb DebugMe'. You should see a new window appear with the '(gjdb)' prompt.

(3) gjdb is a command-line driven program. The most important command you should learn how to use is the (limited) on-line help; type help at the prompt.

```
(gjdb) help
```

If you see a command that looks like the one you want, but need more information, type 'help <command>', replacing <command> with the command you want to know more about.

(4) To run the program from within the debugger, type "run" at the prompt.

```
(gjdb) run
```

This executes the your program FROM THE BEGINNING. You are only allowed to use the "run" command once during a gjdb run. After it finished, you will need to restart the debugger using 'M-x gjdb' again.

(5) The other most important command is "quit".

When gjdb is stuck, not giving you a command prompt, you may simply kill the emacs buffer in which gjdb is running. To do this, make sure your cursor is in the debugger window and type 'C-x k'. Do this now, since after typing 'run' this program quickly finished and you cannot get the debugger's attention again.

(6) Restart the debugger again. Let's see what's happening within the main function. So that we can do this, we'll set a breakpoint at the beginning of main().

```
(gjdb) break DebugMe.main
```

This tells the debugger to stop when execution reaches the beginning of the main method. (You can also set breakpoints at specific line numbers. If you've written small modular programs, you will rarely find that necessary.)

Start the program by typing 'run'. When execution reaches the breakpoint, emacs will show the DebugMe.java file in a separate window with the current line indicated by an '=>' arrow.

(7) At the beginning of the 'main' function, there's no sign yet of the bug. Execute slowly through the code using either the 'next' command or the 'step' command. These two commands have a crucial difference: 'next' executes one statement in full, including any method calls. 'step' normally executes one statement, but if the statement includes a method call, execution stops at the first line of the called method. In emacs, C-c C-n is shorthand for 'next' and C-c C-s for 'step'.

Type C-c C-n twice while watching the '=>' arrow in the other window. The arrow should now point to the line that declares 'sum' and assigns it a value. This means that we are about to execute that line, but haven't yet.

(8) Let's see if the list 'geomSeries' has been constructed correctly. It is made of ListNodes, each containing an item and a reference to the next ListNode in the list. Let's inspect the local variables and confirm that they have values we expect.

```
(gjdb main[1]) info locals
```

You can also inspect the value of any variable or expression using the 'print' command. If the expression being printed is an Object, the debugger prints the object using the toString method. If the class defines a toString method, it is used; otherwise, a default toString is used.

```
(gjdb main[1]) print geomSeries
```

Since geomSeries is a reference to a ListNode, and ListNode doesn't have a toString method defined, this information isn't very useful. But we can view the ListNode's item as follows.

```
(gjdb main[1]) print geomSeries.item
```

Each item is a Double (from java.lang), which defines a toString method, which prints the item. We can see the second element of the list as follows.

```
(gjdb main[1]) print geomSeries.next.item
```

[1 point] Implement a toString method in ListNode.java that prints the contents of the full list. Recompile the file, restart the debugger, step again to the third line of main(), and print the value of the list from the debugger.

[1 point] Draw a box-and-pointer diagram of the geomSeries data structure, including all ListNodes and items.

(9) Quit the debugger and restart it. This time, set a breakpoint on the createGeomSeries method.

```
(gjdb) break DebugMe.createGeomSeries
```

'run' the program again and print the contents of the variables to see the value of the parameters 'r' and 'N'. Since createGeomSeries is a recursive function, you will want to look at these variables during several invocations of createGeomSeries. To do this, continue executing the program using the 'cont' command, or C-c C-r in emacs. Repeat the two commands 'cont' and 'info locals' until you see each invocation of the recursive function. You should now be able to determine the bug in the program.

[2 points] Fix the bug in the program, recompile it, and run it. Make sure you are getting the right answer.

Check-off

Show your TA or Lab Assistant that DebugMe produces the correct answer, and that you can print a list from within the debugger.

1 point: Show that your toString method makes it possible for you to print a complete list from within the debugger.
1 point: Show your box-and-pointer diagram of the geomSeries list (in the original buggy version).
2 points: Show that your debugged program is producing the right answer.