```
                    CS 61B  Lab 7
                   March 6-7, 2014

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

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

  cp -r ~cs61b/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     1     1
              1 + --- + --- + ---       (which equals 1.875)
                   2     4     8

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 Oracle's Java Development Kit is called jdb.  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 '[-]' 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 '[-]' 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.

        [-] 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.

        [-] run

This executes your program FROM THE BEGINNING.

(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.  If you ever need to restart gjdb, make
sure your cursor is in the debugger window and type 'C-x k'.

(6)  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().

        [-] 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.  Think of 'step' as 'stepping into' a method.
In emacs, C-c C-s or the f5 key is shorthand for 'step', and C-c C-n or the f6
key is shorthand for 'next'.

Type f6 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.
(If you stepped too far, you can start over by typing 'run' again.)
```

(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.

```
main[0] info locals
```

You can also inspect the value of any variable or expression using the 'print' command.

```
main[0] print geomSeries
```

Since geomSeries is a reference to a ListNode, this information isn't very useful. But we can view the ListNode's item as follows. (We can also use 'p' as shorthand for 'print'.)

```
main[0] p geomSeries.item
main[0] p geomSeries.item.toString()
```

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.

```
main[0] p geomSeries.next.item
main[0] p geomSeries.next.item.toString()
```

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

```
main[0] print geomSeries.toString()
```

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

[1 point] Find a simple command (either by the 'help' command or by the gjdb documentation in your reader or online) that allows you to directly print the geomSeries to a deeper number of levels, and thereby learn the structure of geomSeries without calling toString(). Note that this command will print a lot of extraneous information too, like a bunch of "final static" constants associated with the Double class.

(9) Set a breakpoint on the createGeomSeries method.

```
[-] break DebugMe.createGeomSeries
```

'run' the program again. Your program is still set to break at the beginning of main, so type 'cont' (or just 'c' for short) to 'continue' execution. You could instead use f8 in emacs. When the debugger reached createGeomSeries, use 'info locals' 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, 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 your box-and-pointer diagram of the geomSeries list (in the
           original buggy version).
1 point:   Show that your toString method makes it possible for you to print
           a complete list from within the debugger.  Say what other debugger
           command can give you the same information without calling toString.
2 points:  Show that your debugged program is producing the right answer.
```

Postscript

Use the debugger to find the sources of the bugs in your project. Although it takes some effort to learn to use a debugger, it will save you a lot of time this semester if you get accustomed to using it now.