CMPS 11
Intermediate Programming
Lab Assignment 5

Our goal in this project learn to use the `jdb` debugger by correcting a slightly buggy program that generates a form letter to Mom and Dad. Begin by copying the example `LetterHome.java` from the Examples page on the class website to your own account space. Also copy the auxiliary files `letter1.dat`, `letter2.dat`, and `letter3.dat`. Read the (rather long) comment block at the beginning of the program explaining its intended operation. Compile and run it as described in the comments, and then note that the program does not behave as expected. This program reads input from a file specified on the command line – something we will see more of in future assignments. To run the program on the input file `letter1.dat` for instance, do

```
% java LetterHome letter1.dat
```

Long ago program errors of various kinds became known as *bugs*. Go to

http://www.computerhistory.org/tdih/September/9/

to see the first computer bug. *Debugging* is the process of locating and correcting these errors. For complex programs, this can be a slow and expensive process that requires as much effort as writing the program in the first place. Program errors can be categorized into three basic types. A *syntax error* occurs when a program statement does not follow the proper rules of form defined in the language specification. Typically these are misspellings of reserved words or identifiers, or the improper use or placement of punctuation characters, like the semicolon or comma. Syntax errors are usually revealed at compile time. The javac compiler provides fairly clear messages to help the programmer find such errors. *Runtime errors* occur when the program is executed using certain sets of data that result in some illegal operation being performed, such as division by zero. The Java Virtual Machine (JVM) provides descriptive error messages at run time to help track these types of bugs. The third, and most subtle, class of errors is *logic errors*. These are the result of using a faulty algorithm to solve the problem. Some incorrect steps that result in wrong answers are performed, but no error messages are produced to help pinpoint the problem. The only way to detect logic errors is to notice that the program output is simply wrong. Extensive time spent on debugging usually means that insufficient time was spent on carefully specifying, organizing, and structuring the program. If the design is poor, then the resulting program is often a structural mess, with convoluted, hard to understand logic. Devoting careful attention to the design phases helps reduce the amount of debugging that must be done. Even when appropriate care *is* used however, bugs can still creep in.

A time honored method to debug both Runtime errors and logic errors is to place print statements (i.e. `System.out.println()` in Java) at judicious points in the program to track the values of critical variables. The programmer can then detect the point or points during execution where those variables depart from their expected values, helping to pinpoint the error. The programs we've studied so far have been sufficiently simple that we could manually trace each step, following the values of all variables at each point in time, and thereby understanding the program fully. With large complex programs, this becomes impractical or impossible. A *debugger* is a software tool that performs such a trace automatically, allowing one to step through execution at any desired speed. One can follow the precise logical pathway taken by the program through loops and conditionals, and monitor any or all variables along the way. The Java debugger we will use in this assignment is called *jdb*. To use it you must first compile your program using the `-g` flag:

```
% javac  -g  myProgram.java
```

This inserts special instructions into the executable `myProgram.class` that will be used by `jdb` to trace execution. To run `jdb` do

```
% jdb myProgram command_line_arguments
```

This project was designed to read input from a file (and hence the need for command line arguments) because `jdb` cannot (easily) be run on programs that read from standard input. This is because `jdb` itself reads from standard input, and the program can't tell which are debugger commands and which are inputs to the program.

Perform the following steps to complete the assignment.

- Compile the program for later use with `jdb`: `% javac -g LetterHome.java`
- Run the program normally using `letter1.dat` as input: `% java LetterHome letter1.dat`
- Observe that there are at least two errors, since there are two departures from the expected output. (Read the comments in the program to see what that is.)
- Run the program in `jdb` using `letter1.dat` as input: `% jdb LetterHome letter1.dat`. You will see something about `jdb` being initialized, then you should see the command prompt `">"`.
- Type `help` to see a list of debugger commands along with a short description of what each command does.
- Set a breakpoint at the start of the program so that you can step through the program when it is run. A breakpoint is a place in the program where the debugger will pause and await further instructions. The debugger commands `stop in` and `stop at` are used to set breakpoints.
    - `stop in ClassName.MethodName` (sets a breakpoint at the beginning of Method).
    - `stop at ClassName:LineNumber` (sets a breakpoint at LineNumber).

  Thus, to set an initial breakpoint, do: `stop in LetterHome.main`. You will see a message saying that the stop will be set when the class is loaded.
- Run the program by typing: `run`. The program is now running, but temporarily paused at the breakpoint you set above. To see where you are in the source code type `list`. This shows the source code surrounding the breakpoint, and the next instruction to be executed, which is indicated by the symbol `=>`. You can see that execution is halted right before the first instruction in function `main()`, namely `Scanner in = new Scanner(new File(args[0]));`
- Recall that the first program error was that a valid sentence code was not accepted as valid. Type `list 125` to see the code that checks for a valid sentence code. Set a breakpoint here by typing `stop at LetterHome:125`. (It will be helpful to have a listing of the source code with line numbers included. This can be done by running `vi` or `emacs` in a separate window.)
- Run the program up to the next breakpoint by typing `cont`. Type `locals` to see the values of all local variables (i.e. those local to function `main()`) at this time. You can advance the program exactly one step by typing `next`. You can also see the value of a single variable by doing `print variableName`. Examine the values of each of the local variables at this point, and notice that when you step past the breakpoint, execution flows into the true branch of the test for a valid sentence code. Why was the conditional evaluated to be true? Evaluate it yourself given the values of the variables. You should now see the program error. (Hint: the valid sentence codes are 1 through 5 inclusive.)
- Recall that the second program error was the weather modifier, i.e. "foggy" instead of "great". Type `list 140` to see that the `weatherModifier()` method is called on line 142. Set a breakpoint at that line by doing `stop at LetterHome:142`, and type `cont` to continue execution up to the next breakpoint.

- Notice that the program does not hit the breakpoint at line 142. Instead it hits the one at line 125 again. To understand why, look at the input file `letter1.dat`, and observe that the sentence code 2 (for the weather sentence) appears on line 3. Thus line 142 is not executed until the third iteration of the surrounding while loop, while line 125 is executed on every iteration. Therefore we have to hit 125 three times before we can hit 142. We've only hit 125 twice so far. Type `cont` to hit the breakpoint at 125 yet again, then `cont` one more time to finally hit the breakpoint at 142. You can also just type `next` again and again to watch the loop go around, instead of `cont` which always takes you up to the next breakpoint. You can also do `clear LetterHome:125` to delete the breakpoint at 125, and then `cont` takes you directly to 142.
- Type `list` to see that the upcoming instruction is the call to `weatherModifier()`. Examine the value of its argument `modifierCode` at this point by doing `print modifierCode`.
- Recall that the `next` command causes the next instruction in the current method (namely `main()`) to be executed. Thus `next` causes the debugger to step *over* method calls. But this is not what we want, since the error, whatever it is, lies inside `weatherModifier()`. Instead type the command `step`, which follows execution *into* a method. Now do `list` again and observe that we have entered the method `weatherModifier()` and are about to execute its first instruction. Remember, `next` goes over function calls and `step` goes into them. Another variation is `step up`, which advances to the next instruction in the calling function (i.e. the method that called the one we're presently in.) In other words, `step up` causes the current method to finish, then pause right after we return to the calling function.
- Type `locals` to monitor the value of the method parameter `m`. Observe that its value is identical to the function argument `modifierCode` which we examined above, just as one would expect.
- Now do `step` a few times to follow execution within `weatherModifier()`. What path through the conditionals does it take? What path did you expect it to take, given the value of `m` above? You should now see the program error. If not, type `step` a few more times, do `locals` again and note that the value the string variable `word` is "foggy", not "great" as expected. Why is that?
- Run the program a few more times in `jdb` and play with other debugger commands. Type `man jdb` from the Unix prompt to see more detail on how the debugger commands work.

What to turn in

Write a one sentence description of each of the two errors you found in the above exercise, then fix the errors. Compile the corrected program, and run it again on the same input `letter1.dat`. Observe that there is still an error somewhere. Use `jdb` to track down that error, write a one sentence description of it, then fix it. Test the corrected program on `letter1.dat`, `letter2.dat` and `letter3.dat`. Delete the comment block at the beginning of the file and replace it with the your usual identification block. Include in that comment your neatly written descriptions of the three errors found. Submit the altered file `LetterHome.java` to the assignment name `lab5`, then check the submission in the usual ways.