CISC181 Fall 2010 LabXX

- This lab an all subsequent labs will be due Wednesday at 11:55 on Sakai.
- Start to think about people you would like to work with for a project team. In my experience, teams that have members with wildly varying skill levels do no perform well. If you think you might like to work with someone, try designing and coding with them in lab to see if it works well.
- All methods that have return values or side effects (other than printing) must be tested using the checkExpect method. Every method in this lab is testable except paint() (which you'll test by trying out.)

Preparation (do not submit for grading)

1. A *bit* is one binary digit. The binary number system (the basis for all modern computing) is based on only two digits, rather than the ten digit systems used by people. The value of a bit can be either 0 or 1, and as such can be represented in Java by a boolean. An array of booleans is also known as a *bit string*. Create a class called BitString which contains an array of boolean. All data on a computer is stored as bit strings. Since java uses 31 bits[1] to represent an integer, make the constructor of your BitString initialize the boolean array to a new array of size 31, and set each element to `false`.
2. Add a method `intValue()` to your BitString which returns its integer value. Look up the binary numeral system on Wikipedia to see how to convert a number from base-2 (binary) to base-10 (decimal.)
3. Add another constructor to your BitString which takes an integer and initializes the BitString to that integer's value. Again, use Wikipedia to find the conversion method.
4. Write a `toString()` method for your BitString. The string representations should of course be in binary. Your string should *not* contain any leading 0's. For example: the array {`false, true, false, true`} would give the string "1010", but the array {`false, true, false, false`} would give "10". An array of all `false`s should return the string "0".

Programs (to be graded)

1. In this lab, you will implement *Conway's Game of Life.* You should start by reading about it on Wikipedia. Create a class `CellularAutomaton` using a 2-dimensional boolean array (or *bit matrix*) to represent the grid.
2. You'll be implementing your grid as a *torus* (donut shape) rather than an infinite plane. Think about how to properly handle the edge cases (*e.g.* the row directly above the very top row of the grid should be the very bottom row.) and write an `up` method which takes a row number and returns the row number directly above it. Then write similar methods for `down`, `left`, `right`.

[1] Actually it uses 32, but the high (thirty second) bit is used to determine the sign (whether the integer is positive or negative.)

3. Write an `update` method for your `CellularAutomaton` which updates the grid to the next state using Conway's rule. Use the methods you defined in the previous step to make your life easier. Remember also that you'll need a way to keep track of the current state of the machine while building the next state!

4. Add a `setInitialConfiguration` method to your `CellularAutomaton` which takes a LinkedList of `Points`[2] and sets all the Points in the list to `true` (alive) in the grid. The rest of the grid should be set to `false` (dead.)

5. Download the GameOfLife and AutomatonView programs from the resources directory and add them to your project. Modify the `AutomatonView` class to have a `CellularAutomaton` member as well as two colors - one color for cells in the alive state and one for cells in the dead state. Modify the constructor so that it initializes all of these from parameters. You should also modify the size of the canvas so that its height and width are each *three times* the height and width, respectively, of the automaton.

6. Modify the `paint()` method of your `AutomatonView` to display the current state of your automaton. Represent each cell in the grid as a 3×3 filled rectangle. Use the appropriate colors for alive and dead cells. *N.B.* you should *not* do any updating of the machine in `paint()` - you should merely paint the current state of the machine.

7. In GameOfLife.java create a new `CellularAutomaton` and set its initial configuration to the provided list `init`. Modify `cavnas`' constructor to take your new `CellularAutomaton` plus two colors of your choice. Add a call to your automaton's update method in the event loop. Now compile and run GameOfLife and watch what happens.

8. Make a copy of GameOfLife.java and modify it so that your machine size and initial configuration are read in from a file passed in as a command line argument. The first number in the file should be the width of the grid, the second the height. The remaining numbers comprise the points which make up the initial configuration. Download the two sample files (gosper.txt and rpentomino.txt) from the resources directory and try running your program with each of those. You should see some very interesting results. Feel free to create your own initial configuration files and try running your program with those!

You should have a bunch of Java files to submit on Sakai. Submit one script of your Console showing all tests executing, and any other docs required by your TA. You may wish to make a new package for some files; if you do, remember to import the Library as before so you can use Tester.

[2]See the Java API documentation for the Point class.