Homework 5

The "Game of Life" is a classical computer game that takes place in an infinite two-dimensional space consisting of *cells*. Each cell can be either alive, or dead. Each cell has a neighborhood consisting of the 8 immediate neighbor cells that surround it.

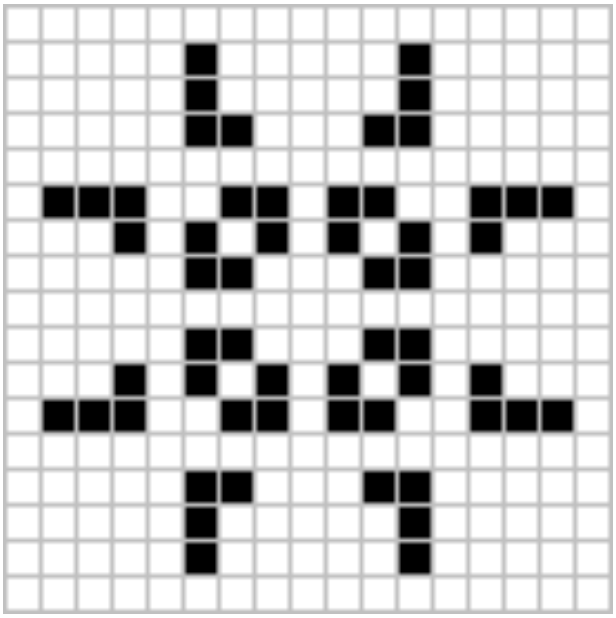
The cells evolve over an infinite sequence of generations. In each generation, each cell computes how many live cells exist in its neighborhood (not counting itself). This information is then used to determine whether the cell will die, become alive, or remain unchanged, in the next generation. The evolution rules are:

- A live cell with fewer than two live neighbors dies (for lack of company).
- A live cell with two or three live neighbors lives on to the next generation.
- A live cell with more than three live neighbors dies (for lack of food).
- A dead cell with exactly three live neighbors becomes a live cell (no biological justification, but helps to keep the game interesting; Also, because the game does not model reproduction, we need some rule to keep the population alive).

In each generation, these rules are *evaluated* (as we say in computer science) and applied for every cell. Note that within a generation, no change happens; all the changes occur when the system moves from one generation to the next.

The player of the game has nothing to do except for supplying an initial cell configuration and then sitting back and watching how the cells evolve from one generation to the next. The game is played forever, or until the player becomes tired of watching and stops it.

Some of the evolutionary patterns that the game generates are boring, some are interesting, and some are quite striking. Here is an example of an interesting pattern:

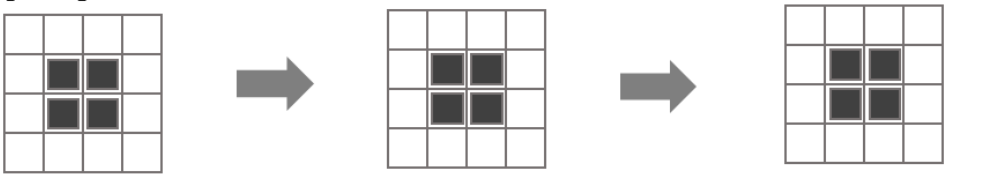


Here again are the rules of the game:

- Any live cell with fewer than two live neighbors dies.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies.
- Any dead cell with exactly three live neighbors becomes a live cell.

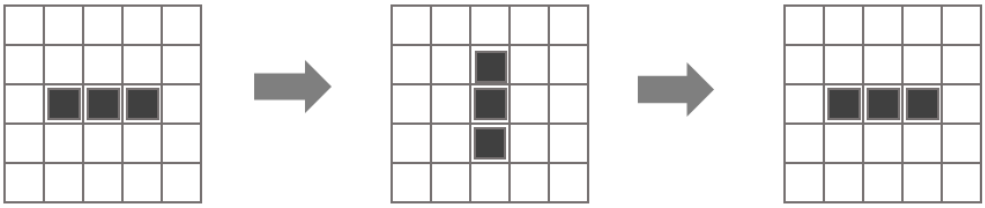
To illustrate, consider the following three examples. Each example shows three generations, beginning with the initial state, which is given. The image on the right shows how the initial state is represented in a data file which is the program's input. The file format is explained below.

Square pattern:



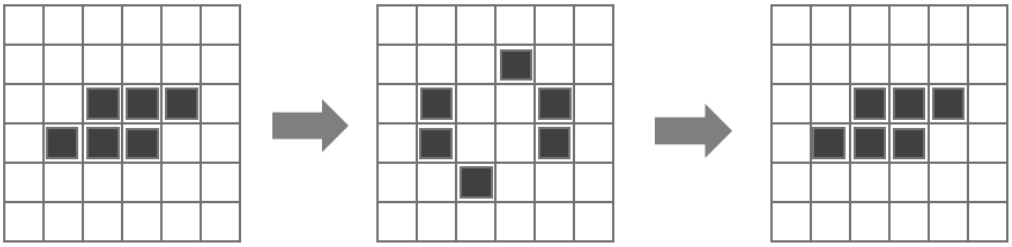
```
1: 4
2: 4
3:
4: .xx
5: .xx
6:
```

Line pattern:



```
1: 5
2: 5
3:
4:
5: .xxx
6:
7:
```

Pulsar pattern:



```
1: 6
2: 6
3:
4:
5: ..xxx
6: .xxx
7:
8:
```

The so-called "Square" pattern, shown at the top, turns out to be static: it remains unchanged from one generation to the next. The "Line" pattern switches from line to column to line to column and so on, for ever. A similar cyclic behavior also happens in the "Pulsar" pattern, which repeats itself every two generations. The pattern shown in the previous page is also a pulsar, with a cycle length of 3. There are infinite possible patterns and behaviors, of which these are just four examples.

File format

The initial state of the game is specified and stored in a text file, which serves as the program's only input. The first line in the file specifies the number of rows of the game's "board", and the second line the number of columns.

Next comes a sequence of lines, each containing a (possibly empty) string, one for each row of cells in the initial board. An empty line represents a row with dead cells only. In a non-empty line, each dead cell is represented by the character '`.`', and each live cell by the lower-case character '`x`'. In each line, all the dead cells that appear after the rightmost live cell are not represented (in other words: each line ends with the rightmost live cell, i.e. with an '`x`').

The `In` and `StdDraw` Classes

The *Game of Life* program is designed to read data from a supplied text file, and draw things on the screen. In order to perform these operations – read and draw – the `GameOfLife.java` class uses the services of two supplied I/O (input / output) libraries: `In` and `StdDraw`, respectively. Some of the `In` functions are used for reading the initial board data from the given file, and some of the `StdDraw` functions are used for drawing the board on the screen. In order to use these functions, you must compile – once only – the supplied `In.java` and `StdDraw.java` classes. This will produce the two executable files `In.class` and `StdDraw.class`. From this point onward, when your `GameOfLife.java` program will make a function call like `StdDraw.show()`, this call will be serviced by that function, which is a member of the executable `StdDraw.class` file. This is just one example of some of the `In` and `StdDraw` functions that we use in this application.

In principle, we could have given you the executable `In.class` and `StdDraw.class` files, instead of the source `.java` files. We decided to do the latter in order to maintain an open source spirit, and in order to have you compile the files yourself and "feel" how one Java class file can provide services to other Java class files. At this stage of the course, there is no need to read or try to understand the code of the `In` or `StdDraw` classes. Instead, you can focus on the In class API, and on the `StdDraw` class API.

How to read an API: There are tens of thousands of Java classes out there, and each is documented in its own API. The writers of these APIs are typically the code developers, so be properad for a mix of well-written and poorly-written APIs. Don't be alarmed by the size and complexity of any given API. When reading an API, it's a good idea to ignore all the irrelevant details and focus instead only on the small number of functions that you actually need for your application (just like reading a book of recipes, ספר בישול). Typically, someone will give you a hint what to focus on. That's what we'll do next.

Input Files

We supply five text files, named `square.dat`, `line.dat`, `glider.dat`, `pulsar.dat`, and `hypnotic.dat`. Each file stores an initial board state. These files, along with the class files `StdIn.java` and `StdDraw.java` and the skeletal `GameOfLife.java` files, must be stored in the same working folder (the folder that we gave you for this assignment).

Implementation

Your task is completing the implementation of the `GameOfLife.java` class. Start by reading the documentation of the `read` function, and then complete the function's code. Note that you have to construct a 2D array (game board) in which the actual cells data starts at row 1 (not 0), and at column 1 (not 0). This 2D array configuration will be quite helpful in the implementation of subsequent functions, as you will see later. For more information and guidelines, read the documentation of the `read` function. Tip: The only method from the `In` class that you need to use is `readLine()`. Specifically, for each line in the data file, read the line into a string, and then use the information contained in that string to initialize the respective row in the 2D array (remember that when this array was constructed, Java initialized all its elements to 0). Note: in lecture 5-1 we gave examples of how to read integers from a file, using a method named `readInt`. Reading a *line* (instead of a single *int* value) using the `readLine` method is quite similar, and both methods are described clearly in the `In` class API. (At this stage, think about *methods* as *functions*. Later in the course we'll have much to say about this distinction).

Next, implement the `print` function. Use Java's `printf` function to create a well-formatted and good looking output, as shown in the examples below.

After implementing these two functions, uncomment the `test1` function, compile, and execute. Here is the output that you should see if you run the program using the initial "Square" and "Line" patterns (in two separate runs):

```
% java GameOfLife square.dat
  0  0  0  0
  0  1  1  0
  0  1  1  0
  0  0  0  0

% java GameOfLife line.dat
  0  0  0  0  0
  0  0  0  0  0
  0  1  1  1  0
  0  0  0  0  0
  0  0  0  0  0
```

(In each run, only the initial generation is shown).

Run the program (with `test1` activated) on the two files shown above and make sure that your code reads and prints the files correctly.

Next, implement the `count` and `cellValue` functions. At this stage you should write test code of your own, designed to verify that these two functions work correctly and return the right values. Put this test code in the `test2` function, compile, and execute on the `square.dat` and `line.dat`

data files (in two separate runs). Use your judgement to decide how to implement this test, and what output the test should print.

Next, implement the `evolve` function, and test it using the `test3` function. Here is an example of the output that you should see:

```
% java GameOfLife line.dat
Generation 0:
  0  0  0  0  0
  0  0  0  0  0
  0  1  1  1  0
  0  0  0  0  0
  0  0  0  0  0
Generation 1:
  0  0  0  0  0
  0  0  1  0  0
  0  0  1  0  0
  0  0  1  0  0
  0  0  0  0  0
Generation 2:
  0  0  0  0  0
  0  0  0  0  0
  0  1  1  1  0
  0  0  0  0  0
  0  0  0  0  0
```

Finally, you are now in a position to test the `play` function. This function displays the board using the `show` function instead of the `print` function, making the output look much better.

Our `show` function, whose code is given to you, uses some of the `StdDraw` functions. Although you don't have to do it in order to complete the program, you must read the code of our `show` function and understand how it works (but wait until you see examples of using `StdDraw` in lecture 5-2). In order to do so, it is recommended to experiment with the function's code. Be sure to make a copy of this function (you can declare another function, say `show1`), before you start messing up with it. Next, start experimenting. For example, instead of using a 100 milliseconds delay factor, try using, say, 10 ms and 1000 ms, and watch the impact on the program's behavior. Then experiment with the canvas sizes, etc.

Submission

Submit only one Java class: `GameOfLife.java`. Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines.

Submission deadline: March 1st , 2024, 23:55.