CS 142, Spring 2024
Programming Project 6: Game of Life (40 points)
Due: Thursday, May 23, 2024, 11:59 PM

This project will give you practice with animation, 2D arrays and GUIs. Turn in a files `LifeModel.java`, `LifeGui.java` and `LifeMain.java`.

The **Game of Life** is a simulation by British mathematician J. H. Conway in 1970. The game models the life cycle of bacteria using a two-dimensional grid of cells. Given an initial pattern, the game simulates the birth and death of future generations of cells using a set of simple rules. In this assignment you will implement a simplified version of Conway's simulation and a basic user interface for watching the bacteria grow over time.

Your Game of Life program should begin by prompting the user for a file name and using that file's contents to set the initial state of your bacterial colony grid. Then it will allow the user to advance the colony through generations of growth. The user can type `t` to "tick" forward the bacteria simulation by one generation, or `a` to begin an animation loop that ticks forward the simulation by several generations, once every 50 milliseconds; or `q` to quit. Your menu should be case-insensitive; for example, an uppercase or lowercase `A`, `T`, or `Q` should work.

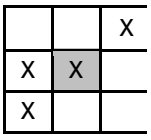
Here is an example **log of interaction** between your program and the user (with console input underlined).

```
Welcome to the CS 142 Game of Life,
a simulation of the lifecycle of a bacteria colony.
Cells (X) live and die by the following rules:
- A cell with 1 or fewer neighbors dies.
- Locations with 2 neighbors remain stable.
- Locations with 3 neighbors will create life.
- A cell with 4 or more neighbors dies.

Grid input file name? simple.txt
---------
---------
---XXX---
---------
---------
a)nimate, t)ick, q)uit? t
---------
----X----
----X----
----X----
---------
a)nimate, t)ick, q)uit? t
---------
---------
---XXX---
---------
---------
a)nimate, t)ick, q)uit? q
Have a nice Life!
```

Game of Life Simulation Rules:

Each grid location is either empty or occupied by a single living cell (X). A location's neighbors are any cells in the surrounding eight adjacent locations. In the example at right, the shaded middle location has three neighbors containing living cells. A square that is on the border of the grid has fewer than eight neighbors. For example, the top-right X square in the example at right has only three neighboring squares, and only one of them contains a living cell (the shaded square), so it has one living neighbor.



The simulation starts with an initial pattern of cells on the grid and computes successive generations of cells according to the following **rules**:

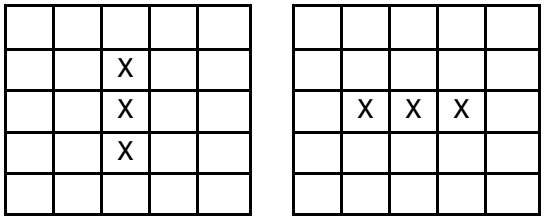
- A location that has zero or one neighbors will be empty in the next generation. If a cell was there, it dies.
- A location with two neighbors is stable. If it had a cell, it still contains a cell. If it was empty, it's still empty.
- A location with three neighbors will contain a cell in the next generation. If it was unoccupied before, a new cell

is born. If it currently contains a cell, the cell remains.

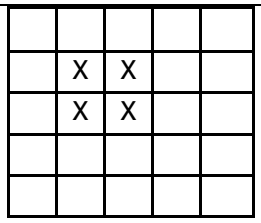
- A location with four or more neighbors will be empty in the next generation. If there was a cell in that location, it dies of overcrowding.

The births and deaths that transform one generation to the next all take effect **simultaneously**. When you are computing a new generation, new births/deaths in that generation don't impact other cells in that generation. Any changes (births or deaths) in a given generation *k* start to have effect on other neighboring cells in generation *k*+1.

Check your understanding of the game rules by looking at the following example at right. The two patterns at right should alternate forever.



Here is a second example. The pattern at right does not change on each iteration, because each cell has exactly three living neighbors. This is called a "stable" pattern or a "still life".



Input Grid Data Files:

The grid of bacteria in your program gets its initial state from one of a set of provided input text files, which follow a particular format. You can assume that the user will enter a valid grid file name and that all of the files contents are valid. You do not need to write any code to handle a misformatted file. The behavior of your program in such a case is not defined in this spec; it can crash, it can terminate, etc. You may also assume that the input file name typed by the user does not contain any spaces.

In each input file, the first two lines will contain integers *r* and *c* representing the number of rows and columns in the grid, respectively. The next lines of the file will contain the grid itself, a set of characters of size *r* x *c* with a line break (`\n`) after each row. Each grid character will be either a `'-'` (minus sign) for an empty dead cell, or an `'X'` (uppercase X) for a living cell. The input file might contain additional lines of information after the grid lines, such as comments by its author or even junk/garbage data; any such content should be **ignored** by your program.

The input files will exist in the same working directory as your program. For example, the following text might be the contents of a file `simple.txt`, a 5x9 grid with 3 initially live cells:

```
5
9
---------
---------
---XXX---
---------
---------
```

← *number of rows tall*
← *number of columns wide*

← *grid of rows and columns*
- is a dead cell, X *is a living cell*

Implementation Details:

The `LifeModel` class should represent the current state of the world. It should keep track of the state of the grid of bacterial cells in a 2-dimensional list of lists or an array of arrays. It should also contain the following public member functions and operators:

`LifeModel(`**fileName**`)`	Constructs a `LifeModel` that stores a grid of the data in the passed in file
`getRows()`	Returns the number of rows in the grid
`getCols()`	Returns the number of columns in the grid
`isAlive(`**row, col**`)`	Returns `true` if the bacteria at that row and column location is alive and `false` if it is not.

`update()`	Updates the state of the model, using the rules described above, to contain the next version of the bacteria's state.
`toString`	Returns a string representation of the grid with – characters used to represent empty cells and capital X characters used to represent live cells.

The `LifeGui` class should create and animate the simulation. It should also contain the following public member functions and operators:

`LifeGui(`**model**`)`	Constructs a `LifeGui` to display the contents of the passed in `LifeModel` class. It should ask the model to update and then redraw every half a second
`update()`	Asks the model to update its state and then redraws that updated state. Live cells should be drawn in one color and dead cells in another. Each cell should take up at least a 10 pixel by 10 pixel square. However, the exact dimensions and colors are to you. Just make sure a viewer can tell which squares represent alive cells and which don't.

The `LifeMain` class should manage the simulation and contain your `main` function. When your program starts it should print out an introduction, prompt for an input file name, read and output that file's contents and prompt the user with the following options:

```
Welcome to the CS 132 Game of Life,
a simulation of the lifecycle of a bacteria colony.
Cells (X) live and die by the following rules:
- A cell with 1 or fewer neighbors dies.
- Locations with 2 neighbors remain stable.
- Locations with 3 neighbors will create life.
- A cell with 4 or more neighbors dies.

Grid input file name? simple.txt
---------
---------
---XXX---
---------
---------

a)nimate, t)ick, q)uit? a
```

If the user types in **a**, as above, your program should open the GUI and run your animation until the user closes it. When the user closes the animation window the program **should not close**. Instead, it should re-prompt the user with the same options. If the user instead types **t**, it should advance the state of the model by one step, output it as text to the console and then re-prompt with the same options. You can see an example of this and of the quit message in the log of execution shown on the first page of this document.

Style Guidelines:

Compiling: Your code should compile without any errors or warnings and should work with any input files in the format described above.

Style: Half of your grade will be based on the style of your code. Follow the same good style practices that we have followed all quarter in the rest of our projects. Use whitespace and indentation properly. Use descriptive variable and function names. Limit lines to 100 characters. Give meaningful names to functions/variables and follow Java's naming standards. Localize variables. Include only the fields you absolutely need.

As always, reduce redundancy whenever possible. Use loops, methods, parameters, returns and calling methods of your objects to achieve this. You are welcome to add additional methods. However, if you add them to your classes make sure that they are **private**. All fields must also be private.

Your code should have adequate **commenting**. The top of all of your files should have a descriptive comment header with your name, the assignment name, date, course and a description of what your code does. Each method should have a comment header describing that method's behavior, any **parameters** it accepts, any values it **returns**, and any assumptions the method makes about how it will be used. For larger methods, you should also place a brief inline comment on any complex sections of code to explain what the code is doing.