# CS 111, Spring 2020

# Programming Assignment #1

# Due 2/6/2020, 7 pm

**TOPICS: 2 dimensional arrays, classes, constructors, mutators, accessors, iteration. Moderately Difficult**

This assignment is based on problem P7.10 in edition 8/e of your text. This problem is also in earlier editions. The problem involves the “Game of Life”. To facilitate viewing each generation of the game you will be given two methods, 1 that sets the canvas size based on the size of the “board” and one that draws the board to the screen using StdDraw. These method may need to be modified based on the implementation of YOUR class.

To implement this project you will create a class **Life**. The class contains a 2-D array for the game board. The data type of the array can be either character or integer (Your choice). Each element in the array represents one square in the game board that represents a “cell” that is in one of 4 states: occupied, empty/free, being born (in the next generation), or dying (in the next generation). The board evolves through a series of generations with generation 0 (zero) being the initial board with some cells occupied and the rest free.

To determine the next generation you must traverse the entire array setting the state of individual cells subject to the following rules:

1. A new cell will be born if it has EXACTLY 3 occupied neighboring cells
2. A cell will die of loneliness if has one or fewer neighbors.
3. A cell will also die of overcrowding if it has 4 or more neighbors.
4. All other cells remain unchanged.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

The neighbors of a given cell are in pink in the diagrams below:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

In the current generation cells that are dying are still considered occupied, and can thus affect the state of their neighbors. Cells that are being born have no effect on surrounding cells.

The class has the following members:

* Data members (**ALL SHOULD BE PRIVATE**)
  + A 2 dimensional array of either characters or integers. Initially it is declared without a size, but will be instantiated inside the constructors. IE: int[][] theBoard
  + integer variables for the number of rows and columns. These values will be given by the user when an instance of the class is created.
  + 4 private data members (either integer or character) representing the 4 states of a **cell**, **beingBorn**, **dying**, **free**, or **occupied**. These will be initialized in your constructor to a value of YOUR CHOICE.
* Constructors:
  + A default constructor with no parameters: **public Life()** this method creates a 10x10 board, setting the rows, columns, and state data members appropriately.
  + A non-default constructor that gives the size of the board:

**public Life( int row, int col)**

This will instantiate the board using the given number of rows and columns, and then will initialize the other data members appropriately.

* Accessors:
  + **public boolean isFree(int r, int c):** This method returns true if the state of the cell at row ‘r’ and column ‘c’ is **“free”.**
  + **public boolean isOccupied(int r, int c):** This method returns true if the state of the cell at row ‘r’ and column ‘c’ is **“occupied”.**
  + **public boolean isDying(int r, int c):** This method returns true if the state of the cell at row ‘r’ and column ‘c’ is **“dying”.**
  + **public boolean isBorn(int r, int c):** This method returns true if the state of the cell at row ‘r’ and column ‘c’ is **“beingBorn”.**
  + **private int countNeighbors(int r, int c):** For the cell at row r and column c this method counts and returns the number of neighbors that are either “occupied” or “dying” in the current generation. Any cell in the board potentially has 8 neighbors. To determine the indexes of the neighbors, you will need to determine how the values of r and c need to change to identify a neighboring cell. Before accessing a cell you must determine if it is in bounds, otherwise you will get an **ArrayIndexOutOfBounds** exception! **This is a private method that is used only inside the class to count the neighbors of a cell**
* Mutators:
  + **public void setBorn(int r, int c):** sets the cell indicated by r(row) and c (col) to “**beingBorn**”.
  + **public void setOccupied(int r, int c):** sets the cell indicated by r(row) and c (col) to “**occupied**”.
  + **public void setDying(int r, int c):** sets the cell indicated by r(row) and c (col) to “**dying**”.
  + **public void setFree(int r, int c):** sets the cell indicated by r(row) and c (col) to “**free**”.
  + **public void clearBoard():** this method sets all of the cells in the current board to “**free**”
  + **public void fillBoard(Scanner inputFile):** this method reads the contents of the data file, marking some cells in theBoard as occupied. Until the file is empty, it reads values 2 at a time, the two values represent a row/column pair which names a cell who’s state should be set to “**occupied**”.
    - This final thing this method should do is call the “setCanvas” method that sets the canvas size for StdDraw, and then calls the drawBoard method that draws the initial board.
  + **public void playGame():** This method will make traversals through the board as long as the board changes. If a generation produces no changes, then the method should exit. In each traversal,
    - this method visits EVERY cell in the board, and calls the countNeighbors method to calculate the number of neighbors the cell has, then:
      * if the cell has 3 neighbors, and is NOT currently “**occupied**”, it sets the state of the cell to “**beingBorn**”
      * If the cell is “**occupied**”, and has 4 or more neighbors, it sets the state of the cell to “**dying**”
      * if the cell is “**occupied**” and has one or fewer neighbors it sets the state of the cell to “**dying**”
    - At the end of the traversal of the entire board, it calls the “nextGeneration” method to calculate the next generation of the game.
    - Finally it will redisplay the board by calling the drawBoard method.
    - To watch the board change gracefully, after drawing the board it is best to ask the program to “sleep” a few seconds, OR with current CPU speeds the game moves too fast to enjoy changes in each generation. To pause the game, after calling drawBoard add the command: **Thread.sleep(7000);**

The value 7000 will cause the game to sleep for 7 seconds, 7000 is 7 miliseconds. To access this command we must deal with an exception: **InterruptedException**

Add **throws InterruptedException** to both the heading of playGame method and your main method.

* + **private void nextGeneration():** This method traverses the entire board, if a cell’s state is “**beingBorn**” it set’s it’s state to “**occupied**”. If the state is “**dying**” it sets it to “**free**”. All other cells remain unchanged. **This should be a private method that is used only inside the class to calculate the next generation of the game**
* PROVIDED METHODS: These are utility methods that can be used and modified to reflect your implementation of your Life class. The keyword “this” is optional but refers to the current instance of the Life class that is being played.
* **setCanvas is called after the board is filled, and initializes the drawing canvas for StdDraw based on the size of the board**

**private void setCanvas(){  
 StdDraw.setXscale(0.0, 10\*this.rows);  
 StdDraw.setYscale(0.0, 10\*this.cols);  
 }**

* **drawBoard draws the board to the screen, if the cell is occupied it is drawn in blue, free cells are white.**

private void drawBoard(){

// this assumes “cell[0][0] if the array is the upper

//left corner”  
 int x, y;  
   
 for(int i=0; i< this.rows; i++){  
 for(int j=0; j< this.cols; j++){  
 //use the isOccupied method to see if a cell

// is occupied  
 if(this.isOccupied(i, j))   
 StdDraw.setPenColor( StdDraw.BLUE);  
 else  
 StdDraw.setPenColor( StdDraw.WHITE);  
   
 x = j\*10+5;  
 y = (10\*(this.rows-i)) - 5;  
   
 StdDraw.filledSquare( x, y, 5);  
 }//inner  
 }  
 }// end drawBoard

# Your main program:

1. Your main program will prompt the user for the name of a data file containing information about the current game to be played.
2. You must use loops and try catch blocks to insure that a valid file name is given.
3. The first line of the data file contains 2 integers, that gives the dimensions of the board.
4. Once these are read in call the constructor for your Life class to create an instance of the game supplying the number of rows and columns the board will contain
5. Call fillBoard for the instance created in step 4, with the name of the Scanner attached to the input file. fillBoard will read series of values from the file indicating cells to set to occupied. A sample data file that constructs the “glider” game illustrated in your text in problem 7.10 is :

6 7

1 2

2 3

3 1

3 2

3 3

* The final thing your main program will do is call playGame on the instance created in step 4