HW05 Code

GameOfLife:

```
 *  Game of Life.
 *  Usage: "java GameOfLife fileName"
 *  The file represents the initial board.
 *  The file format is described in the homework document.
 */

public class GameOfLife {

    public static void main(String[] args) {
        //String fileName = args[0];
        //// Uncomment the test that you want to execute, and re-compile.
        //// (Run one test at a time).
        //// test1("glider.dat");
        //// test2("line.dat");
        //// test3("pulsar.dat", 3);
        //// play("hypnotic.dat");
    }

    // Reads the data file and prints the initial board.
    private static void test1(String fileName) {
        int[][] board = read(fileName);
        show(board);
    }

    // Reads the data file, and runs a test that checks
    // the count and cellValue functions.
    private static void test2(String fileName) {
        int[][] board = read(fileName);
        show(board);
        board = evolve(board);
        show(board);
    }

    // Reads the data file, plays the game for Ngen generations,
    // and prints the board at the beginning of each generation.
    private static void test3(String fileName, int Ngen) {
        int[][] board = read(fileName);
        for (int gen = 0; gen < Ngen; gen++) {
            System.out.println("Generation " + gen + ":");
            print(board);
            board = evolve(board);
        }
    }

    // Reads the data file and plays the game, for ever.
    public static void play(String fileName) {
        int[][] board = read(fileName);
        while (true) {
            show(board);
            board = evolve(board);
        }
    }

    // Reads the initial board configuration from the file whose name is
fileName, uses the data
    // to construct and populate a 2D array that represents the game
board, and returns this array.
    // Live and dead cells are represented by 1 and 0, respectively. The
constructed board has 2 extra
    // rows and 2 extra columns, containing zeros. These are the top and
the bottom row, and the leftmost
    // and the rightmost columns. Thus the actual board is surrounded by a
"frame" of zeros. You can think
    // of this frame as representing the infinite number of dead cells
that exist in every direction.
    // This function assumes that the input file contains valid data, and
does no input testing.
    public static int[][] read(String fileName) {
        In in = new In(fileName); // Constructs an In object for reading
the input file
        int rows = Integer.parseInt(in.readLine());
        int cols = Integer.parseInt(in.readLine());
        int[][] board = new int[rows + 2][cols + 2];
        for (int i = 0; i < rows+2; i++) {
            for (int j = 0; j < cols+2; j++) {
```

```java
                board[i][j] = 0;
            }
        }
        for(int i = 1; i <= rows; i++){
            String line = in.readLine();
            if (line == null || line == "") {
                continue;
            }
            char[] cells = line.toCharArray();
            for (int j = 0; j < cells.length; j++) {
                if (cells[j] == 'x') {
                    board[i][j+1] = 1;
                }
            }

        }
        return board;
    }

    // Creates a new board from the given board, using the rules of the
game.
    // Uses the cellValue(board,i,j) function to compute the value of each
    // cell in the new board. Returns the new board.
    public static int[][] evolve(int[][] board) {
        int rows = board.length;
        int columns = board[0].length;
        int[][] newBoard = new int[rows][columns];
        for (int i = 1; i < rows-1; i++) {
            for (int j = 1; j < columns-1; j++) {
                newBoard[i][j] = cellValue(board, i, j);
            }
        }
        return newBoard;
    }

    // Returns the value that cell (i,j) should have in the next
generation.
    // If the cell is alive (equals 1) and has fewer than two live
neighbors, it dies (becomes 0).
```

```java
    // If the cell is alive and has two or three live neighbors, it
remains alive.
    // If the cell is alive and has more than three live neighbors, it
dies.
    // If the cell is dead and and has three live neighbors, it becomes
alive.
    // Otherwise the cell does not change.
    // Assumes that i is at least 1 and at most the number of rows in the
board - 1.
    // Assumes that j is at least 1 and at most the number of columns in
the board - 1.
    // Uses the count(board,i,j) function to count the number of alive
neighbors.
    public static int cellValue(int[][] board, int i, int j) {
        int value = 0;
        if (board[i][j] == 1) {
            if ((count(board, i, j) == 2) || (count(board, i, j) == 3)) {
                value = 1;
            }
        }else{
            if (count(board, i, j) == 3) {
                value = 1;
            }
        }

        return value;
    }

    // Counts and returns the number of living neighbors of the given cell
    // (The cell itself is not counted).
    // Assumes that i is at least 1 and at most the number of rows in the
board - 1.
    // Assumes that j is at least 1 and at most the number of columns in
the board - 1.
    public static int count(int[][] board, int i, int j) {
        int aliveCells = 0;
        if (board[i-1][j-1] == 1) {
            aliveCells++;
        }
        if (board[i-1][j] == 1) {
            aliveCells++;
        }
        if (board[i-1][j+1] == 1) {
            aliveCells++;
        }
        if (board[i][j-1] == 1) {
            aliveCells++;
        }
        if (board[i][j+1] == 1) {
            aliveCells++;
        }
        if (board[i+1][j-1] == 1) {
            aliveCells++;
        }
        if (board[i+1][j] == 1) {
            aliveCells++;
        }
        if (board[i+1][j+1] == 1) {
            aliveCells++;
        }

        return aliveCells;
    }

    // Prints the board. Alive and dead cells are printed as 1 and 0,
respectively.
    public static void print(int[][] arr) {
        for (int i = 1; i < arr.length-1; i++) {
            for (int j = 1; j < arr[0].length-1; j++) {
                if ((i == 1) && (j == 1)) {
                    System.out.print(arr[i][j]);
                } else {
                    System.out.print("  "+arr[i][j]);
                }
            }
            System.out.println("");
        }
    }
    // Displays the board. Living and dead cells are represented by black
and white squares, respectively.
    // We use a fixed-size canvas of 900 pixels by 900 pixels for
displaying game boards of different sizes.
    // In order to handle any given board size, we scale the X and Y
dimensions according to the board size.
    // This results in the following visual effect: The smaller the board,
the larger the squares
    // representing cells.
    public static void show(int[][] board) {
        StdDraw.setCanvasSize(900, 900);
        int rows = board.length;
        int cols = board[0].length;
        StdDraw.setXscale(0, cols);
        StdDraw.setYscale(0, rows);

        // Enables drawing graphics in memory and showing it on the screen
only when
        // the StdDraw.show function is called.
        StdDraw.enableDoubleBuffering();

        // For each cell (i,j), draws a filled square of size 1 by 1
(remember that the canvas was
        // already scaled to the dimensions rows by cols, which were read
from the data file).
        // Uses i and j to calculate the (x,y) location of the square's
center, i.e. where it
        // will be drawn in the overall canvas. If the cell contains 1,
sets the square's color
        // to black; otherwise, sets it to white. In the RGB
(Red-Green-Blue) color scheme used by
        // StdDraw, the RGB codes of black and white are, respetively,
(0,0,0) and (255,255,255).
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int color = 255 * (1 - board[i][j]);
                StdDraw.setPenColor(color, color, color);
                StdDraw.filledRectangle(j + 0.5, rows - i - 0.5, 0.5,
0.5);
            }
        }
        StdDraw.show();
        StdDraw.pause(100);
    }
}
```