Game of Life :

```
/**
 *  Game of Life.
 *  Usage: "GameOfLife fileName"
 *  The file represents the initial board.
 *  The file format is described in the HW05 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(fileName);
        //// test2(fileName);
        //// test3(fileName, 3);
        play(fileName);
    }

    // Reads the data file and prints the initial board.
    private static void test1(String fileName) {
        int[][] board = read(fileName);
        print(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);
        for(int i=1; i<board.length-1; i++)
        {
            for(int j=1; j<board[0].length-1;j++)
            {
                System.out.println("for cell " + i + " " + j + " the
sum is: " +count(board,i,j ));
                System.out.println("the new cell value for this cell is
" + cellValue(board, i, j) );

            }
        }
        //// Write here code that tests that that the count and
cellValue functions
        //// are working properly, and returning the correct values.
```

```
    }

    // 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.
    private static void play(String fileName) {
        int[][] board = read(fileName);
        while (true) {
            show(board);
            board = evolve(board);
        }
    }

    // Reads the data from the given fileName, uses the data to
construct the initial board,
    // and returns the initial board. Live and dead cells are
represented by 1 and 0, respectively.
    // To avoid dealing with extreme cases, constructs a board that has
2 extra rows and 2
    // extra columns, containing zeros. These will be the top and the
bottom row, and the
    // 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.
    private static int[][] read(String fileName) {
        StdIn.setInput(fileName);
        int rows = Integer.parseInt(StdIn.readLine());
        int cols = Integer.parseInt(StdIn.readLine());
        int [][] board = new int [rows][cols];

        for (int i=0; i<board.length; i++)
        {
            for (int j=0; j<board[0].length; j++)
                board[i][j]=0;
        }

        for (int i=0; i < board.length; i++)
```

```
        {
            String line=StdIn.readLine();
            for (int j=0; j < line.length(); j++)
            {
                if(line.charAt(j)=='x')
                    board[i][j]=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.
    private static int[][] evolve(int[][] board)
    {
            int [][] newboard = new int
[board.length][board[0].length];
            for(int i=1;i<board.length-1; i++)
            {
                for(int j=1; j<board[0].length-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).
    // 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.
    private static int cellValue(int[][] board, int i, int j)
    {
        if(board[i][j]==1)
        {
            if(count(board,i,j)<2)
                return 0;
            if(count(board,i,j)==2||count(board,i,j)==3)
                return 1;
            if(count(board,i,j)>3)
                return 0;
        }
        if(board[i][j]==0)
        {
            if(count(board,i,j)==3)
                return 1;
            else
                return 0;
        }
        return 0;
    }

    // Counts and returns the number of living neighbors of the given
cell.
    // 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.
    private static int count(int[][] board, int i, int j)
     {
        int sum=0;
        if(board[i][j-1]==1)
            sum++;
        if(board[i-1][j]==1)
            sum++;
        if(board[i][j+1]==1)
            sum++;
        if(board[i+1][j]==1)
            sum++;
        if(board[i-1][j-1]==1)
            sum++;
        if(board[i-1][j+1]==1)
            sum++;
        if(board[i+1][j+1]==1)
            sum++;
        if(board[i+1][j-1]==1)
            sum++;
```

```
        return sum;
    }

    // Prints the board. Alive and dead cells are printed as 1 and 0,
respectively.
    private static void print(int[][] arr)
    {
        for (int i=0; i < arr.length; i++)
        {
            for (int j=0; j< arr[0].length; j++)
                System.out.print(arr[i][j]+ " ");
            System.out.println("");

        }
    }

    // Displays the board. Living and dead cells are represented by
black and white squares, respectively.
    // In this app we use the same canvas (graphical window) size for
displaying boards of different sizes.
    // In order to handle any board size, we scale the X and Y
dimensions according to the board size.
    // This causes the following effect: The smaller the board, the
larger the squares.
    private 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);
        StdDraw.show(100); // delay the next display 100 miliseconds
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                int grey = 255 * (1 - board[i][j]);
                StdDraw.setPenColor(grey, grey, grey);
                StdDraw.filledRectangle(j + 0.5, rows - i - 0.5, 0.5,
0.5);
            }
        }
        StdDraw.show();
    }
}
```