AP Computer Science — Programming Project—Game of Life

ASSIGNMENT OVERVIEW
In this assignment you'll be creating a program called `Life.java`, which will allow the user to run a text-based or graphics-based simulation of John Conways game of *Life* on the computer.

This assignment is worth 50 points and is due on the *crashwhite.polytechnic.org* server at 23:59:59 on the date given in class.

BACKGROUND
John Conway's game of *Life* is a type of *cellular automata* that demonstrates how simple rules can lead to complex patterns or behaviors. Wikipedia has an excellent introduction to the topic, and Project 7.2 in our textbook describes the game as well.

In *Life*, the "world" is simulated on a large, two-dimensional grid, with each cell in the grid either empty or occupied by an "organism." In the course of a turn, the contents of each cell are determined by looking at the eight cells surrounding it. In the basic game, an organism in one cell will be born, or live, or die, according to the following rules:

1. A live cell with < 2 neighbors will die (due to under population).
2. A live cell 2-3 neighbors lives on to the next generation.
3. A live cell with > 3 neighbors dies (due to overpopulation).
4. An empty cell with 3 neighbors becomes a live cell (from reproduction).

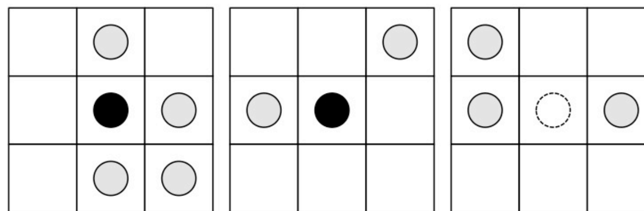

Figure 1. Figure 2. Figure 3.

In Figure 1, the cell whose neighbors are being counted is at the center and colored black. (In the program's output, however, all "live" cells will have the same color, shape, or designation—see the *Sample Interactions* at the end of this document.) The neighbors in the eight surrounding squares are colored gray, and in this first case, the black colored cell will "die" during this generation due to overpopulation in this area of the board—it has four or more occupied cells around him.

In Figure 2 the live cell at the center will survive into the next generation because it has 2-3 neighbors around it.

In Figure 3, the empty cell in the center will be the site of a live cell in the next generation (indicated here by an uncolored dashed circle) , thanks to the existence of the three neighbors surrounding it.

See *https://en.wikipedia.org/wiki/Conway's_Game_of_Life* for further information on this game.

```
..00.000.0.......000...
....00..0..............
......00...............
......0.......00.......
00....0.......000......
00..00........0..000...
................000.00.
.......0........0...0..
```

The *Model* for this game might take a number of forms depending on how you choose to code it: a `boolean` grid with live cells `true` and empty cells `false`; a `char` or `String` array with "`.`" and "`0`"; an `int` array with `0`s and `1`s. This Model of the game is internal, and isn't something that the user will have any direct interaction with.

The *View* for the game that the user sees is also up to you: a text-based version might simply display the strings in your array, or display characters based on the true-false values in a boolean array; a graphics-version will take the cells and display them as pixels, or perhaps as squares in a grid.

Running a simulation consists of establishing a "board" for the grid of cells, populating the cells with an initial "seed" generation, and then watching the population on the board evolve over time as successive generations are produced according to the population rules above.

PROGRAM SPECIFICATION

Create a Java program, with associated classes, that:

a. includes appropriate APIs for all classes and methods
b. creates a 2-dimensional Array on which the simulation will run
c. populates the array, either randomly and/or by user selection of cells
d. displays the initial state of the array on screen
e. uses the basic rules of Conway's *Life* to calculate a new generation
f. displays the new state of the array on screen
g. repeats this process until the user breaks out of the program

DELIVERABLES

Life.zip

This single file will be a zipped directory (folder) of your BlueJ project. It will include as a minimum your `Life.java` main program and a `Board.java` class, any other classes you create during the development of your program), and a `package.BlueJ` file.

To submit your assignment for grading, copy your file to your directory in `/home/studentID/forInstructor/` at *crashwhite.polytechnic.org* before the deadline.

ASSIGNMENT NOTES

- This program will probably consist of 2-3 classes. At the least it will require a `Board.java` class that manipulates the 2-d grid, and a `Life.java` main program that runs the simulation.

- Because this simulation is observed via a display, you'll need to decide whether you want to run a text-based simulation or a graphics-based simulation. The text-based simulation is easier to program but doesn't look as good, while the graphics-based version allows for greater detail (the cells are smaller), at the cost of increased programming complexity. It is strongly recommended that you do the text-based simulation, at least to begin.

- If using the text-based display, you'll want to clear the screen between the display of successive generations. You can use this code for that purpose:

```
private static void clearConsole()
{
    System.out.print("\033[H\033[2J");
    System.out.flush();
}
```

- To create the initial seed generation of cells, you can either have the user identify cells that he/she wants populated, and/or have the program randomly populate the board. User input will require using the `Scanner` class, while randomly populating the board will require the `Random` class.

- You'll actually need *two* 2-d arrays when running this simulation: one for the current population state, and one that you'll create for the next population when the rules of *Life* are applied. Once the rules have been applied to completely create the entire next population, the contents of that array will be transferred into the original array, and the process repeats from there.

- When counting neighbors, the eight squares surrounding a given cell should be evaluated using a nested `for` loop. If a given cell is populated, a sum counter is incremented, and the final result of that sum counter is used to determine the fate of that cell in the next generation. In the example below, `board[1][4]` has eight cells around it that should be evaluated. Note that gray cells in the example are not part of the board, and should *not* be evaluated. Attempts to identify a cell that is outside the bounds of the board–when looking at `board[0][0]`, for example–will cause an error.

columns

rows

	0, 0	0, 1	0, 2	0, 3	0, 4	0, 5
	1, 0	1, 1	1,2	1, 3	1, 4	1, 5
	2, 0	2, 1	2, 2	2, 3	2, 4	2, 5

- You will probably find that the simulation runs faster than your computer display can refresh, producing some odd flickering effects. Most programs will require a brief delay between screen refreshes. You can use this code to introduce a pause for some number of milliseconds:

```
// Sleep for some amount of time to slow display down
try
{
    Thread.sleep(TIME_DELAY);
```

```
        // TIME_DELAY is an integer in milliseconds
    }
    catch(InterruptedException ex)
    {
        Thread.currentThread().interrupt();
    }
```

GETTING STARTED

1. With paper and pencil, and perhaps in collaboration with a partner, run a small, simple Life simulation on paper to make sure you understand the rules.

2. Identify what the main components are that you'll need to include in your program.

3. Sketch out the basic flow of your program using a flowchart, and write some pseudocode that you can use to begin implementing those main components.

4. Create a new project in BlueJ that will allow you to manage this assignment.

5. Create a `Board.java` class that you will use to construct and manipulate the board. This class will probably include methods `.toString()`, `.set()`, and `.get()`. Test this class before moving on to the main program.

6. Much of your program will consist of working through the contents of the board. It is common to use nested loops for this process:

   ```
   for (int row = 0; row < Board.ROWS; row++)
   {
       for (int col = 0; col < Board.COLS; col++)
       {
           // do stuff with board[row][col]
       }
   }
   ```

   Note that Java arrays list the *row* first, which runs vertically, and then the *column*, which runs horizontally.

7. *Test each bit of code as you go*, making sure that one piece works before you proceed on to the next section. You'll repeatedly run through this edit-compile-test, edit-compile-test process to progressively find bugs and fix them *while* you're writing your program, not afterwards.

8. Save your program from time to time, and once a day or so, archive/zip your BlueJ *Life* folder and save a backup copy of it on another device or machine: a flash drive, your home folder on the *crashwhite.polytechnic.org* server, etc.

9. When your program is completed (but before the deadline), copy a final archived package to the server as indicated above.

QUESTIONS FOR YOU TO CONSIDER (NOT HAND IN)

1. Which strategy did you use for copying your data from one array to another? `Clone`? `System.arraycopy`? Nested loops that copy one element at a time?

2. Which runs your program faster: the BlueJ software package, or a Console/Terminal running on your computer?

3. What is a *cellular automata*?

4. Names have been given to many of the forms that arise in a typical game of Life. A *block, beehive,* or *loaf*, is not uncommon, and you're sure to see a *blinker*. If you're lucky, you might see a *glider* skittering across the screen.

5. Conway's original game of Life is a 2-dimensional cellular automata, with a set of rules that govern the calculation of future states. What would a 1-dimensional cellular automata look like? What rules would allow one to calculate future states? See *https://en.wikipedia.org/wiki/Elementary_cellular_automaton* for an interesting discussion.

6. Would you get a *glider* tattoo? Would you buy a 1-d cellular automata scarf?

SAMPLE INTERACTIONS

```
John Conway's Game of Life
..........00..............00........0...
........0...0....0........0.0......00...
.........0...0..000.........0..0........
...00.........0.0..0...0.0.0.0..0....0.0
00.0..0...0...0..000..0....00..00.......
...0...0..0.......0....0..0...0.000000.0
....00..0...............00.....0....000.
......00......................0.........
......0........0...............0........
00.....0......00...............0.....0..
00...0...........0..0............0....0.
....000........0000.0.......0..........0
...0.0000...........0....0...0.......0.0
..................00.........0.........0
............................0...........
......00.................0..............
.........0.............00...............
......0...0.............................
.......00.0.............................
.........0..............................

Generation: 10


John Conway's Game of Life
...........0..............00.......00...
.........0000...000.......0.0......00...
.............0.00.0.......0.00......0...
..000........00.0..0......00.00.0.......
...0...........0.0.0..000..0000...00.0..
..00.000.0.......000...00000..0..000....
....00..0...............00....00000...0.
......00......................00.....0..
......0.......00..............00........
00....0.......000...............0.......
00..00........0..000..................0.
................000.00.................0
.......0........0...0.......00.........0
......00...........0........00........0.
........................................
........................0...............
......00................0...............
.......00.0.............................
.......00.0.............................
........00..............................

Generation: 11


John Conway's Game of Life
...........00....0........00.......00...
..........000..00.0......00.00.......0..
...0......00.0....00.....00...0....00...
..000........0.....0...0.00....0...00...
.....00.......00.0.00.0............00...
..00.0000.......00.0..0....0........0...
...00...0.........0....0.....0.....0....
......00.....................0...0......
.....00.......0.0.............0.0.......
00....0......0..000............0........
00...0........0....00...................
...............00...00................00
......00........0.0.00......00........00
......00....................00..........
........................................
........................................
......000...............................
........................................
..........0.............................
.......000..............................

Generation: 12
```