# Homework 08 - A Game Of Life

**Authors: Hussain Miyaziwala, Jack Kelly, Vince Li, Andrew Chafos, Davis Williams, Shishir Bhat**

## Problem Description

You were uploading some changes to the Matrix (fixing some bugs regarding fidget spinners) when finally instead of clicking "submit" you accidentally clicked "delete everything". You had mentioned in a team meeting that it really doesn't make sense to have a "delete everything" button, but your boss Carl has a sense of flare and insisted on keeping it for dramatic effect. And so, just like that, the Matrix disapeared with billions of humans now without a simulation to live in. To prevent the humans from waking up and revolting, program a simple simulation to keep things under control.

## Solution Description

Your task will be to simulate a population of cells. Your simulation will consist of a 2D array that we will call our universe. This 2D array will contain all of the `Cell`s that make up our basic universe. For this assignment you have been provided `Cell.java`, a simple class that will represent a cell. Like all cells, a `Cell` is either dead or alive.

You will also be provided part of the `Board.java` class - to complete this assignment you should correctly fill in the body of each method as specified. You may create more methods and fields if you like but you may not modify the existing method headers.

The following fields are included in the provided code and should remain as is:

```
private Cell[][] universe;
private int dimensionLength = 0;
private int generationCount = 0;
```

**The following are methods for you to complete:**

- **`public Board(int dimensionLength, int[][] seed)`**

This constructor should do the following:

1. Assign instance variable `dimensionLength` the value of the respective parameter.
2. Instantiate `universe` to be a 2D `Cell` array of size `dimensionLength` x `dimensionLength`
3. Loop through each `Cell` in `universe` and instantiate them with the default constructor
4. Call method `applySeed` with the `seed` array (more on this later - see the `applySeed` section)
    - For this homework you may assume dimensionLength will be at least 4, and `seed` will contain at least one value.

- **`private void applySeed(int[][] seed)`**

This method determines what cells will start off alive in `universe`. Loop through `universe` and make the `Cells` "alive" where the corresponding entry in the `seed` array is not 0. You may assume that `seed` has the same dimensions of `universe`.

Ex) if `seed[3][2] == 3`, then `universe[3][2].setAlive(true)`

- **`private int getCellStatus(Cell[][] arr, int x, int y)`**

Return *0* if `arr[x][y]` is out of bounds or the `Cell` at that location is dead; otherwise return 1. This method is used by the provided `getAliveNeighbors` method.

- **`public void tick()`**

This method should progress the universe one step forward and increment `generationCount` by one accordingly. During this method you should iterate through `universe` and for each `Cell` decide if it should be dead or alive based on the following criteria. Note: You should consider these factors only for the current generation of cells. The updated values of cells should not be taken into account when computing the next generation. (Hint: it might be helpful to store a temporary universe array to temporarily store the updated cells)

*Be sure to use the provided `getAliveNeighbors` method to help you implement this method.*

1. **Underpopulation**: Any living cell with less than two neighbors dies
2. **Overpopulation**: Any living cell with more than three neighbors dies
3. **Continuation**: Any living cell with either two or three neighbors continues to live
4. **Reproduction**: Any dead cell with exactly three neigbors becomes alive

**The following methods have been provided to you:**

`public String toString()`

Will return a simple visualization of `universe` and `generationCount`

`private int getAliveNeighbors(Cell[][] arr, int x, int y)`

This method depends on your correct implementation of the above `getCellStatus` method. It will return the number of living neighbors for any given cell. *You need not do anything for this method, however understanding it is crucial for this assignment.*

## Testing

We encourage you to write your own tests for this assignment. Create a `Test.java` file, instantiate a `Board` and try out different seeds. You can use something like the following:

```
public class Test {
    public static void main(String[] args) {
        int[][] seed = {
            {0, 1, 0, 0, 1},
            {0, 0, 1, 1, 1},
            {0, 0, 0, 0, 0},
            {0, 0, 0, 0, 1},
            {1, 1, 0, 0, 0}
        };
        Board b = new Board(5, seed);
        System.out.println(b);
        for (int i = 0; i < 10; i++) {
            b.tick();
            System.out.println(b);
        }
    }
}
```

## Inspiration

This problem is an implementation of *Conway's Game Of Life*. It's just a peak into the world of cellular automata!

## Rubric

- [22] **constructor**
  - [4] Assign distanceLength resepective value and make variable private
  - [4] Instantiate `universe` correctly and make variable private
  - [11] Universe has no null values
  - [3] `applySeed` method is called
- [15] `applySeed`
  - [15] Correctly sets initial state
- [10] `getCellStatus`
  - [4] Returns 0 if out of bounds
  - [3] Returns 0 if dead
  - [3] Returns 1 if alive
- [55] `tick`
  - [3] Increments `generationCount`
  - [13] Underpopulation
  - [13] Overpopulation
  - [13] Continuation
  - [13] Reproduction

## Import Restrictions

You may not import anything for this homework assignment.

## Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- var (the reserved keyword)
- System.exit
- try/catch blocks

## Javadocs

For this assignment, you will be commenting your code with Javadocs. Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online overview for them is extremely detailed and helpful.

You can generate the javadocs for your code using the command below, which will put all the files into a folder called javadoc:

```
$ javadoc *.java -d javadoc
```

The relevant tags that you need to include are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 1.0
```

```
 */
public class Dog {

    /**
     * Creates an awesome dog (NOT a dawg!)
     */
    public Dog() {
    ...
    }

    /**
     * This method takes in two ints and returns their sum
     * @param a first number
     * @param b second number
     * @return sum of a and b
     */
    public int add(int a, int b) {
    ...
    }
}
```

A more thorough tutorial for Javadocs can be found here. Take note of a few things:

1. Javadocs are begun with `/**` and ended with `*/`.
2. Every class you write must be Javadoc'd and the `@author` and `@verion` tag included. The comments for a class should start with a brief description of the role of the class in your program.
3. Every non-private method you write must be Javadoc'd and the `@param` tag included for every method parameter. The format for an `@param` tag is `@param <name of parameter as written in method header> <description of parameter>`. If the method has a non-void return type, include the `@return` tag which should have a simple description of what the method returns, semantically.

Checkstyle can check for Javadocs using the -a flag, as described in the next section.

### Checkstyle

For this assignment, we will be enforcing style guidelines with Checkstyle, a program we use to check Java style. Checkstyle can be downloaded here. To run Checkstyle, put the `jar` file in the same folder as your homework files and run

```
java -jar checkstyle-6.2.2.jar -a *.java
```

The Checkstyle cap for this assignment is **65 points**. This means that up to 65 points can be lost from Checkstyle errors.

## Collaboration

### Collaboration Statement

To ensure that you acknowledge collaboration and give credit where credit is due, **we require that you place a collaboration statement as a comment at the top of at least one java file that you submit**. That collaboration statement should say either:

*I worked on the homework assignment alone, using only course materials.*

or

*In order to help learn course concepts, I worked on the homework with [give the names of the people you worked with], discussed homework topics and issues with [provide names of people], and/or consulted related material that can be found at [cite any other materials not provided as course materials for CS 1331 that assisted your learning].*

Recall that comments are special lines in Java that begin with `//`.

## Turn-In Procedure

### Submission

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- Board.java

Make sure you see the message stating "HW08 submitted successfully". From this point, Gradescope will run a basic autograder on your submission as discussed in the next section.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your last submission: be sure to **submit every file each time you resubmit**.

### Gradescope Autograder

For each submission, you will be able to see the results of a few basic test cases on your code. Each test typically corresponds to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

1. Prevent upload mistakes (e.g. forgetting checkstyle, non-compiling code)
2. Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

### Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items
- Do not submit `.class` files.
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read the "Allowed Imports" and "Restricted Features" to avoid losing points
- Check on Piazza for all official clarifications