Project 1: Game of Life

DUE: Tuesday Feb 11 at 11:59pm

Extra Credit Available for Early Submissions!
Extra Credit Available for special feature addition (see last page)!

Project Requirements and Constraints

You must:

- Pass the provided style checker from P0 on all your Java files.
- Comment your code well AND pass the provided JavaDoc checker from P0 on all your Java files for this project (no need to overdo it, just do it well).
- Have code that compiles with the command: **`javac *.java`** in your code directory without errors or warnings.
- For methods that come with a big-O requirement (check the provided template Java files for details), make sure your implementation meets the requirement.
- Have code that runs with the command: **`java GameOfLife`**

You may:

- Add additional methods and class/instance variables, however they **must be private or protected**.

You may NOT:

- Make your program part of a package.
- Add additional public methods or public class/instance variables, remember that local variables are not the same as class/instance variables!
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, HashSet, etc.).
- Alter any method signatures defined in this document or the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).

You should NOT:

- Submit any code containing System.exit() calls. This will interfere with automatic grading and negatively impact your project score.
- Submit any code with System.out.println() or System.err.println() calls (outside of main() method). We understand that printing to console is a valid way to debug your program, but extraneous printing may also interfere with grading. Please remove or comment out any debugging print statements that you add before submitting.

Setup

- Create a folder for **`p1`**. This is your "project directory".
- Download the **`p1.zip`** and unzip it in your project directory. This will create a folder **`yourCodeHere`**. This is your "code directory".

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Submission Instructions

- Go to GradeScope and submit *just the Java files from your code directory* to the Project 1 submission link.
- You can resubmit as many times as you'd like, only the last submission will be graded.
- *The GradeScope autograder will give you some immediate feedback* to ensure you have submitted everything correctly. READ THIS FEEDBACK. Common error messages when submitting to the autograder:

  - Cannot Find Java Files

    Something is wrong with your submission, and you should re-check what you've uploaded. Common mistakes: uploading a folder rather than just the Java files, uploading a zip, rar, or other compressed file, uploading .class files.

  - Fail "Compile Alone"

    You are probably missing a file, make sure you submit ***all*** your Java files.

  - Fail "Compile With Tests (Preliminary Check)"

    Somehow you are not compiling with the JUnit tests. Check your own JUnit tests to see why this might be.

  - Fail "Checkstyle Code" or failed "Checkstyle Comments"

    You aren't passing the style checker or the Javadoc checker. Check on your own computer!

Overview

The Game of Life Project is an interactive and dynamic simulation of Conway's Game of Life, a popular example of cellular automata that models the evolution of a grid of cells based on simple rules. In this project, you will implement and extend the classic Game of Life in Java. You will also develop your own `DynamicArray` class as a custom implementation to replace the `ArrayList` class, gaining critical experience with the nuts and bolts of the data structure. The final application will allow users to customize the grid size, load patterns in RLE format, and control the simulation through interface controls. By integrating additional features such as statistics tracking, preset grid sizes, and pattern loading, this project serves as a practical exercise in software design, algorithm development, and file compression.

How the Game of Life Works

The **Game of Life**, devised by British mathematician **John Horton Conway** in 1970, is a cellular automaton that simulates the evolution of a grid of cells based on a set of simple rules. Each cell in the grid can be either **alive** or **dead** and interacts with its neighboring cells to determine its state in the next generation. The game progresses in discrete time steps, known as **generations**, with the state of the grid evolving based on predefined rules.

Rules of the Game of Life

1. **Underpopulation**:
   - A live cell with **fewer than 2 live neighbors** dies, as if by underpopulation.
2. **Survival**:
   - A live cell with **2 or 3 live neighbors** survives to the next generation.
3. **Overpopulation**:
   - A live cell with **more than 3 live neighbors** dies, as if by overpopulation.
4. **Reproduction**:
   - A dead cell with exactly **3 live neighbors** becomes a live cell, as if by reproduction.

Neighbor Definition

Neighbors include the eight surrounding cells:

- Directly adjacent cells (north, south, east, west).
- Diagonal cells (northeast, northwest, southeast, southwest).

Key Characteristics

- **Initial Configuration**:
  - The simulation begins with an initial configuration of alive and dead cells, often referred to as the "seed."
- **Emergent Behavior**:
  - Despite the simplicity of the rules, the Game of Life exhibits complex and often unpredictable patterns, including:
    - **Static Patterns**: Patterns that remain unchanged over time.
    - **Oscillators**: Patterns that repeat after a fixed number of generations.
    - **Spaceships**: Patterns that move across the grid over time.
- **Infinite Grid Potential**:
  - Although typically implemented with a fixed-size grid, the theoretical Game of Life operates on an infinite grid.

Significance
The Game of Life is a powerful demonstration of how complexity and emergent behavior can arise from simple rules. It has applications in mathematics, computer science, and systems modeling, providing insights into cellular automata, algorithms, and even biological systems.

How RLE Files Work

Run-Length Encoded (RLE) files are a compact and widely-used format for describing patterns in Conway's Game of Life. They encode the grid in a way that reduces redundancy, representing consecutive cells of the same state (alive or dead) with a single count and symbol. Each RLE file begins with a **header** specifying the grid's dimensions (e.g., `x = 20, y = 10`) and may include optional comments starting with `#`. The main body of the file contains the encoded pattern, using `o` for live cells, `b` for dead cells, and `$` to indicate the end of a row. A number preceding these symbols specifies the count (e.g., `3o` for three consecutive live cells). The file ends with an exclamation mark (`!`) to signify the end of the pattern. For example, an RLE file might contain `x = 3, y = 2\n3o$b2o!`, which represents a grid with a row of three live cells, followed by a new row with a dead cell and two live cells. This format is efficient for describing sparse or repetitive patterns and is compatible with many Game of Life implementations.

Note that .rle files are simply text files. But they probably won't default to being opened by a text editor on your system. You can modify the settings in your OS to select your favorite text editor as the default opener for this file extension.

More info on rle files here: [Run Length Encoded - LifeWiki]

new 1 | gosperglidergun.rle

```
1  #N Gosper glider gun
2  #O Bill Gosper
3  #C A true period 30 glider gun.
4  #C The first known gun and the first known finite pattern with unbounded growth.
5  #C www.conwaylife.com/wiki/index.php?title=Gosper_glider_gun
6  x = 36, y = 9, rule = B3/S23
7  24bo11b$22bobo11b$12b2o6b2o12b2o$11bo3bo4b2o12b2o$2o8bo5bo3b2o14b$2o8b
8  o3bob2o4bobo11b$10bo5bo7bo11b$11bo3bo20b$12b2o!
```

Fig 1. rle file opened in Notepad++

Understanding File Compression Through RLE
*Working with **Run-Length Encoding (RLE)** files in this project gives you an opportunity to explore the basics of file compression. RLE simplifies repetitive data by encoding consecutive identical values with a count and a value, such as 3o to represent three consecutive live cells. As you parse and use RLE patterns in the Game of Life, you'll see how this approach reduces the size of data while preserving its meaning. This hands-on experience helps you understand how compression works in real-world scenarios, making data storage and transmission more efficient, and gives you a foundation for exploring more advanced compression techniques in the future.*

Requirements

An overview of the requirements is listed below, please see the grading rubric for more details.

- Implementing the classes - You will need to implement required classes and fill the provided template files.
  - GameOfLife is already complete.
  - You have to complete DynamicArray, Cell, and Simulation.

- JavaDocs - You are required to write JavaDoc comments for all the required classes and methods.
  - Some javadocs are already written. You have to write any additional required. The checkstyle will tell you what is needed.

- Big-O - Template files provided to you contain instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.
  - If no big-O requirement is listed for a method, then the requirement does not exist for that method.

- Code Style - You are required to provide appropriate in-line comments to your code and comply with the cs310code stylechecker.

Overview of the Classes

1. GameOfLife

- **Purpose**: Acts as the **view** and **controller** of the application. Handles user interface elements, including rendering the grid and managing user interactions, but does not perform any simulation or evolving.
- **Key Responsibilities**:
  - Render the grid on the screen using the data provided by the Simulation.
  - Manage GUI components like buttons, sliders, and preset size controls.
  - Forward user actions (e.g., toggle cells, load patterns) to the Simulation.
- **Key Methods**:
  - **Rendering**:
    - drawGrid(Graphics g): Draws the grid and cells based on the Simulation data.
  - **Interaction Management**:
    - initializeControls(): Sets up GUI elements such as buttons and sliders.
    - toggleCell(MouseEvent evt): Converts mouse events into grid coordinates and forwards them to Simulation.
  - **Statistics Display**:
    - Updates labels and displays with information fetched from the Simulation.

---

2. Simulation

- **Purpose**: Handles all logic and state management for Conway's Game of Life. Includes the grid, cell evolution, RLE parsing, and rule enforcement.
- **Key Responsibilities**:
  - Maintain and manage the grid of cells.
  - Implement the rules for evolving the grid.
  - Parse and apply RLE patterns to the grid.
  - Provide statistics like alive cells, average age, and max age to the GUI.
- **Key Methods**:
  - **Grid Management**:
    - toggleCell(int row, int col): Toggles the state of a specific cell.
    - reset(): Resets all cells to dead.
  - **Simulation**:
    - evolve(): Advances the grid by one generation based on Conway's rules.
    - applyPatternToGrid(boolean[][] pattern): Applies a predefined or RLE-parsed pattern to the grid.
  - **RLE Parsing**:
    - parseRle(DynamicArray<String> lines): Parses RLE input and converts it into a 2D boolean array.
  - **Statistics**:
    - getAliveCells(): Returns the total number of alive cells.
    - getAverageAge(): Calculates and returns the average age of alive cells.
    - getMaxAge(): Finds and returns the maximum age among all alive cells.
    - getGenerations(): Returns the number of generations elapsed.

---

3. Cell

- **Purpose**: Represents an individual cell in the grid, encapsulating its state (alive or dead) and its age.
- **Key Responsibilities**:
  - Track whether the cell is alive or dead.
  - Maintain the age of the cell when alive.
- **Key Methods**:

- isAlive(): Checks if the cell is alive.
- setAlive(): Sets the cell state and resets its age if becoming alive.
- getAge(): Returns the current age of the cell.
- reset(): Resets the cell to a dead state and age to zero.

---

4. DynamicArray

- **Purpose**: A custom implementation of a dynamic array to replace Java's ArrayList. Manages resizable storage for rows and cells in the simulation grid. Implements the Iterable interface.
- **Key Responsibilities**:
  - Provide functionality to dynamically resize an internal array.
  - Serve as the storage container for grid rows and cells.
- **Key Methods**:
  - add(T element): Adds an element to the array.
  - get(int index): Retrieves an element at a specific index.
  - remove(int index): Removes an element at a specific index.
  - size(): Returns the current number of elements.

How To Handle a Multi-Week Project

While this project is given to you to work on over about two weeks, you are unlikely to be able to complete this is one sitting. We recommend the following order of attack:

Step 1	○ Read and familiarize yourself with the project, both specification and source code. ○ Practice running the javadoc and codestyle checker. You will get a sense of what you will need to fix. It can be tedious and perhaps overwhelming to do this at the end, especially if you are time-crunched.

Step 2	○ Complete implementation of DynamicArray.java
	  ○ It is a complete code skeleton as-is, so it will compile, but it does not work correctly.
	  ○ This class is the main supporting data structure for the simulation, so it is critical to have it correctly working in order to proceed to the next part of the project.
	  ○ Use the main() method of DynamicArray to test. Some tests are already provided, but you should add more! There is more detail on this in the next section, Testing.
	  ○ Also note that some functionality of DynamicArray, such as expanding the size of the array, is not used in the GameOfLife simulation, so you should test this independently.

Step 3	○ Complete implementation of Cell.java
	  ○ This is a simple class; mainly a container object for cell state and age.
	  ○ You will need to write complete javadocs for this class.
	○ *At this point, you should be able to compile and run the GUI and have it appear normal (but non-functional).*

Step 4	○ Start implementation of Simulation.java
	  ○ Once you have completed the constructor and toggleCell, you should be able to see the grid in the GUI and click on cells to activate (alive) them.
	  ○ evolve and countLiveNeighbors are the key methods that will enable the simulation to run. Once these are complete, you should be able to see some action!
	○ Complete other methods in Simulation like reset, and various statistic methods.
	○ *At this point, you should be able to run the entire simulation with the exception of loadRLE button.*

Step 5	○ Finish the RLE file related methods in Simulation.java ○ Add more main() method testing to Simulation.java if needed. ○ You are free to use the loadRleFile method in Simulation.java in your main() testing.

Testing

Running the simulator is one way to test, but we recommend testing smaller units of code as you go and not just the final project after weeks of work. To help you with this, the DynamicArray class has a main method provided in the template files contain useful example code to test your project as you work. This main method works like any other main method and you can use the command "**`java DynamicArray`**" to run the testing defined in **`main()`**

Below is sample output from running the main method in DynamicArray.java:

```
> javac DynamicArray.java
> java DynamicArray
Yay 1
Yay 2
Yay 3
Yay 4
Printing values: 5 10
```

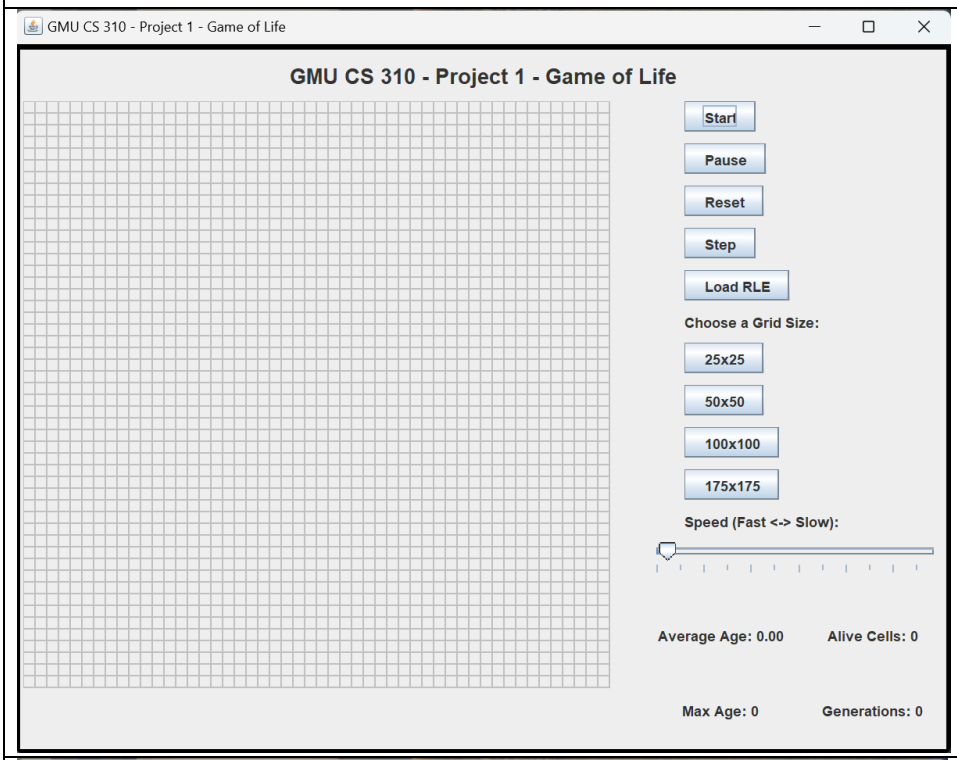
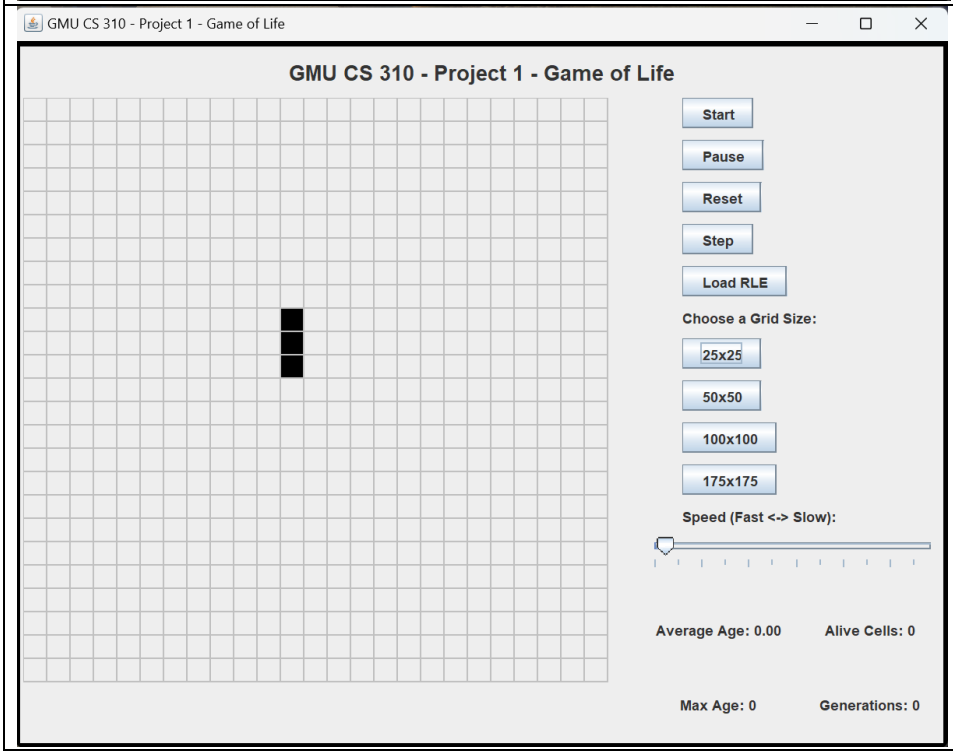
Testing code in main() method of Simulation.java is also provided. You're encouraged to add some more!

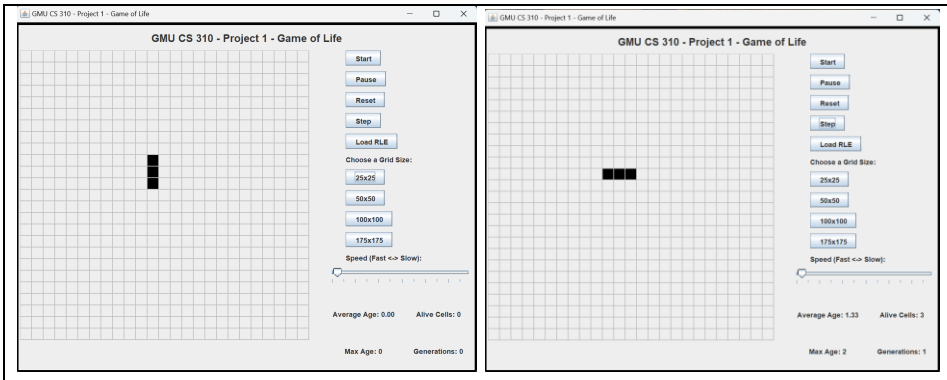
Note: passing all yays != 100% on the project! Those are only examples for you to start testing and they only cover limited cases. Make sure you add many more tests in your development. You can edit **`main()`** to perform additional testing. And you can also add a main (and main method testing) to any other files if you want (except **`GameOfLife`** which already has a main method).

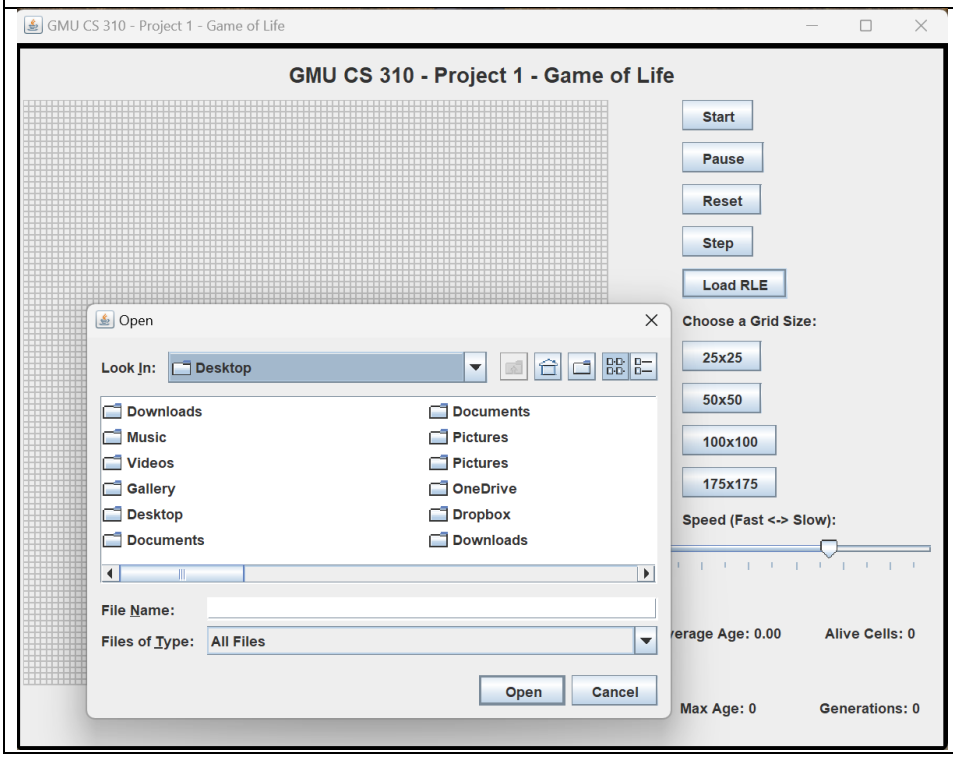
Keep in mind that the Game of Life simulation does not utilize all the methods in the DynamicArray class. For example, the simulation does not remove any elements from the DynamicArrays. But all methods will be graded, so make sure you test on your own!

JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A large part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

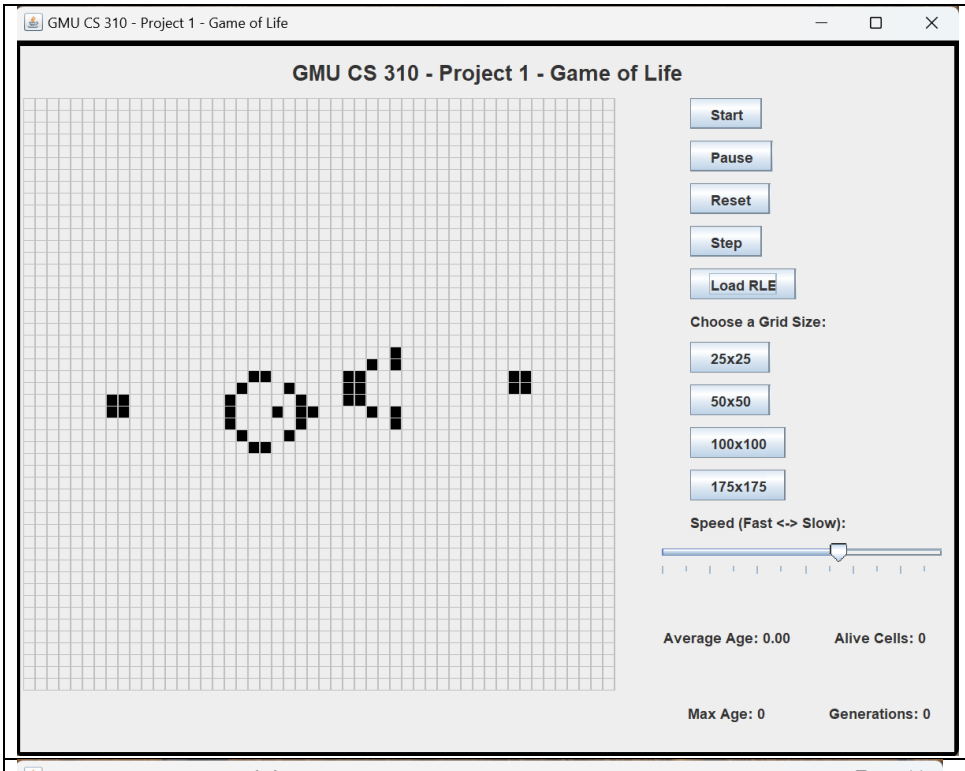
Sample GameOfLife Runs (With Comments)

Simulator	Comments
	This is how the simulation should look at start-up if everything is working correctly. Note the statistics at bottom right show appropriate zero values. Grid will default to 50x50 size. Buttons are generally self-explanatory, but note that Step is provided to help with debugging. It will advance the simulation by ONE generation only.
	Here is a 25x25 grid with three cells manually drawn in, simulation NOT started yet. This is an oscillating pattern. It will alternate between vertical and horizontal each generation. (shown below). You can verify this by hand yourself by considering the alive/dead rules.



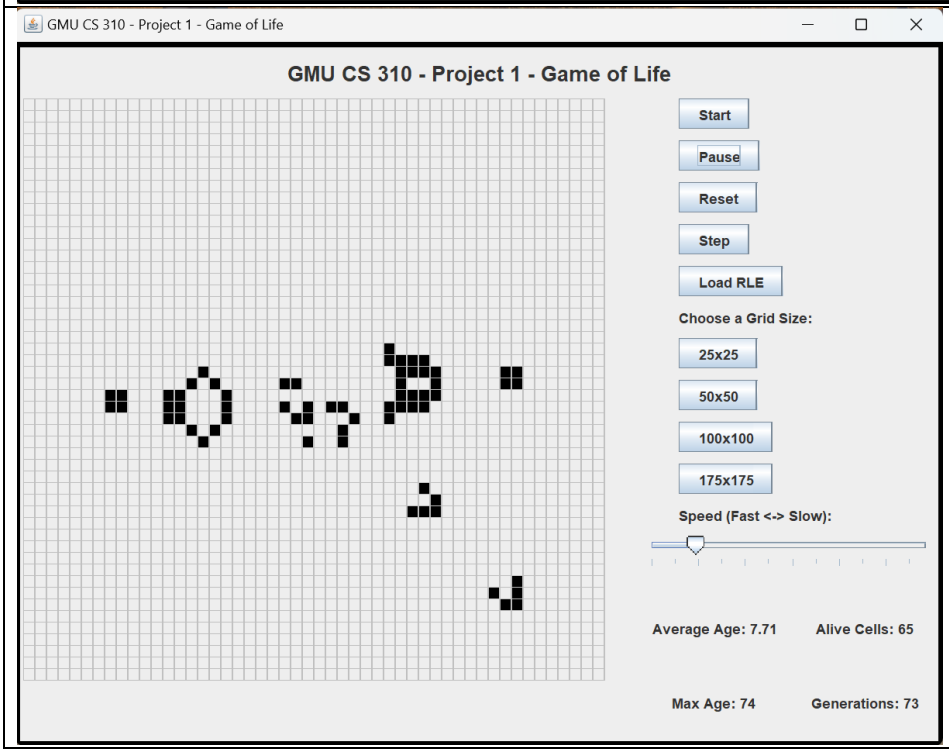


This is what you will see if you click Load RLE button. You can navigate to the sample-rle-files directory that was provided in the project package to load a complex pattern to the grid.



Here is an example of gosperglidergun.rle file loaded to 50x50 grid.

Note that your parseRle and applyPatternToGrid methods needs to be implemented before this will work correctly.



This is what the Gosper Glider Gun will look like after 73 generations. You can see the gliders moving toward the bottom right of the grid.

Testing complex patterns like this is probably best done by loading a pattern with a known behavior and seeing if your implementation replicates the behavior.

To test the correctness of your evolution rules, simple hand-drawn patterns are probably fine.

Here is a good resource for patterns and visualizations: LifeWiki

FAQ

- Do I have to write javadocs for *[insert any field/class/method]*?
  - You have to do whatever you need to do to pass the checkstyle! This involves writing javadocs for *every* class, method, and field.

- What do I do if my javadoc checkstyle doesn't pass?
  - Reference the line number of the error and FIX IT! That may involve deleting weird characters or fixing some spacing issues. If all else fails, delete the whole javadoc comment and re-type it.
  - You are allowed to edit any javadoc that is already provided in the event that it is erroring.

- What do I do if my checkstyle cs310code does not pass?
  - Reference the line number of the error and FIX IT! That may involve changing the name of a variable to meet the naming convention.

- My GUI is not working; I think something is wrong with the provided code?
  - We're 99% sure that is not the case. Double check your supporting classes like DynamicArray and Simulation to ensure all data structures are valid and non-null.
  - You may make a Piazza post if you need additional help.

- Which files do I submit?
  - Please submit : Simulation.java, Cell.java, DynamicArray.java
  - If you submit GameOfLife.java also, IT IS OK. The autograder will be setup to ignore it, and it is not graded in any way.
  - Do not submit any .class files, .jar files, or .xml files.

- What should I do if a method parameter is null?
  - Unless it is specifically called out in the code, don't worry about it.
  - DynamicArray CAN HANDLE null inputs and this is noted where relevant in the source code.

- Should I throw an exception for *[insert any situation]*?
  - It will be clearly stated in the code (mainly DynamicArray) where exceptions are expected to be thrown. Do not do any extra, as that could negatively affect our ability to grade your project.
  - Make sure you understand what it means to throw an exception. DO NOT CATCH IT.

- Should I delete my main() methods before I submit my code?
  - It does not matter.

- Help! My GUI runs too slow!
  - Check to make sure you're not printing to System.out or System.err excessively.
  - Alternatively, buy a better laptop 😉

EXTRA CREDIT opportunity

You may earn 1 (ONE) additional point by doing the following:

Add some cool color to the game! We'll let you be creative in doing this. You may consider to change the color of a cell based on how old it is, or have recently deceased cells decay in color. It's up to you! We would like you to really explore what new enhancement you can add to the game.

Since we want you to have complete creative freedom, we will provide a separate Gradescope assignment for you to submit your extra-credit attempt. The main project submission should be the standard, basic, no-frills version. This is so you won't have to worry about following all the project constraints when you're implementing your creative vision.

When you're ready to share your creation (same due date applies, no late options), follow the submission instructions in the Gradescope assignment marked as P1 Extra Credit. (primarily uploading a demonstration video of your feature along with some code snippets detailing how you implemented it)

Depending on how many submissions we have, the UTAs will be picking the top X submissions to show off to the class. If your code is chosen, you'll get +1pt extra credit.