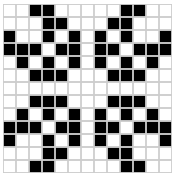
Game of Life



The Game of Life is not actually a game; it is an implementation of a "cellular automata" that John H. Conway chose to call The Game of Life. Once the original organisms are placed on a grid, simple rules determine everything that will happen in future generations. Sometimes the outcome is that all organisms die, other times they end up in a static or oscillating world. Other times, they may form complex organisms that generate new life forms that glide across the screen.

The original article describing the game can be found in the Apr. 1970 issue of Scientific American on p. 120. The game is played on a field of cells, each having eight neighbors (adjacent cells). A cell is either occupied (by an organism) or not. The rules for deriving a (next) generation from the previous one are as follows:

Death If an occupied cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, the organism dies (0 or 1 neighbor(s), it dies of loneliness; 4 to 8 of overcrowding).

Survival If an occupied cell has two or three neighbors, the organism survives to the next generation.

Birth If an unoccupied cell has three occupied neighbors, it becomes occupied.

You will use these rules to create your own life simulation. Wikipedia has a decent description of some of the various lifeforms that are possible (and more) here.

There are much larger, more interesting organisms (patterns) on the Internet. Some require a world that has several hundred elements in each direction. For this assignment, your world is (initially) constrained to a 60 x 60 grid. If you have computing power for larger grids, the changes are easy to make.

The Assignment

You are given the following files, which follow the Model-View-Controller (MVC) design pattern:

Life.java	Controller (runner), contains a `main` method
LifeCell.java	Individual location (cell) in the grid (simulation) Has two states, `aliveNow` (what you see) and `aliveNext` (what you'll see next)
LifeModel.java	The logic that is running the View
LifeView.java	Contains a graphical view (the GUI); displays the simulation

Of these, you need only (initially) modify the model component. To begin, it is suggested that you inspect the LifeCell class to see what instance variables it contains and how it behaves.

In the model class' code, complete the `private void oneGeneration()` method to create a "new generation" from the "current generation". Your `oneGeneration` method must implement the rules of Conway's Game of Life as described previously.

The method `oneGeneration` will be called automagically over and over once the Run button (in the GUI) has been clicked to advance from generation to generation. It can also be paused or resumed under the user's control.

The Life class contains a `main` method to test the simulation. You can start with a randomized population or test your program with the included "*.lif" files: "blinker.lif", "glgun13.lif", "penta.lif", and "tumbler.lif". **Note:** You may need to add the constructors required to accomplish this. The "*.lif" files use the following format, using **"blinker.lif"** as an example:

```
7
11  11
12  11
11  12
12  12
13  12
11  13
12  13
```

This data file represents 7 cells initially alive at positions <row=11, col=11>, <row=12, col=12>, etc., and so forth. This particular initial pattern creates a "blinker" organism after the rules of life have been applied for a few generations.

When finished, create a runnable JAR file (an archive (bundle) of all the files necessary to make your project run) along with your source code (inside the same project folder is fine). Here is how:

- **BlueJ**: *Project -> Create jar file…*, then choose the class with the `main` method and the destination.
- **Eclipse**: *File -> Export*, then *Java -> Runnable JAR file*. Choose the "launch configuration" (the class with the `main` method in your project), then the destination.
    - If you don't see your class with the `main` method as an option, make sure you've run the program at least once; Eclipse will make a launch configuration for you when it runs.

(Advanced) Added functionality

Finally, add the following buttons (and supporting code) to the simulation's GUI:

- ***Reset*** – resets the state of the simulation (re-randomizes the board* or runs another simulation loaded from text file), used rather than quitting and re-running the program

    ** To a new random state, you don't need to store the initial random configuration*

- ***Randomize color*** – toggles the option for living cells to be shown in a randomly generated color (or back to the default blue).
    - Add a drop-down menu or color chooser window to allow the user to pick a color