Chapter 6 Labs

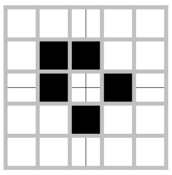
Game of Life

Introduction

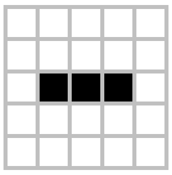
The Game of Life belongs to an interesting class of programs called cellular automatons. Similar to other automatons, the Game of Life consists of a rectangular grid of cells. Each cell can be in one of two states: "alive" (black) or "dead" (white). In each step of the "game", the following rules are applied:

1. Any live cell with fewer than two live neighbors dies
2. Any live cell with two or three live neighbors lives on to the next generation
3. Any live cell with more than three live neighbors dies
4. Any dead cell with exactly three live neighbors becomes a live cell

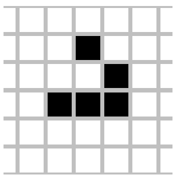
Although these rules are rather simple, surprisingly complex behavior arises from them. For example, stable configurations can occur like this boat:



Although this pattern is static, there are some stable configurations that oscillate like this blinker:



Finally, some patterns called spaceships move across the grid like this glider:



Your Task

In this exercise, all of the graphical work has been done already. Your task is to fill in the `Board` class which includes the actual logic for transitioning from one state to the next (according to the rules above). Some of the class has already been filled in. In the code, the grid is represented as a 2D array of booleans (`true` for alive, `false` for dead). Note that there are actually two arrays, `state` for the current state and `nextState` for the next state (so that the original array isn't modified during the transition).

The first step is to fill in the following helper methods. These will help with the transition logic.

- `public boolean isInside(int row, int col)`—returns `true` if the location is inside of the board and `false` otherwise.

- `public static int[][] getNeighbors(int row, int col)`—returns an 8x2 array containing the locations of the eight neighbors as (row, col) pairs

Now we're ready to fill in the `update()` method. Where applicable, please make use of the helper methods in the class including the ones above. To help guide you, here's a procedural outline:

- Loop through every cell in `state`.
  - Get the neighbors of the cell.
  - Ignoring the invalid neighbors, compute the number of alive neighbors.
  - Use this to compute the state of the corresponding cell in `nextState` according to the rules at the beginning.
- Update `state` using the `shiftState()` method.

When you're finished, run the `GameWindow.java` program. When running, the board will update 10 times per second. To pause/resume it, press the space bar. To run a single update, press the right arrow. To modify individual cells, just click on them (probably while it's paused) to toggle their state. For more fun, use the items in the Load menu to change the seed state.