CSC 202 Lab 3

Goals

- Gain experience using 2D array of objects and nested loops
- Gain experience writing and using classes

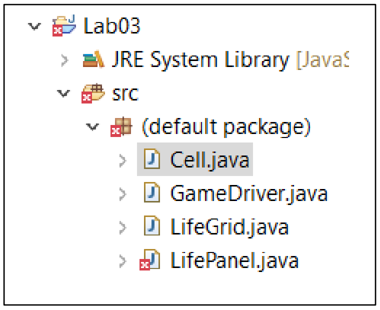
Getting Started

- Open your Eclipse workspace.

- Create a new Java Project **Lab03** (remember select **Don't Create** when asked about creating a module).

- Download the start files from Moodle and drag them into the src folder. Your project should look like the one shown to the right..LifePanel will have syntax errors until you complete some code.

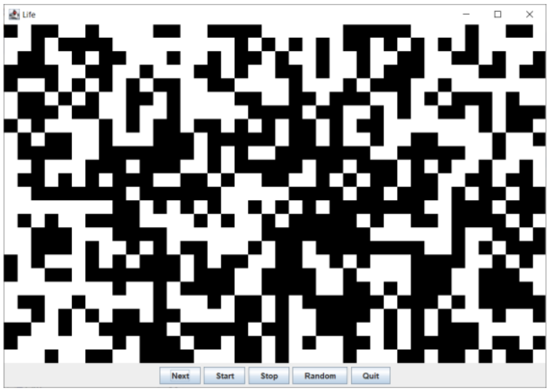
- Add your name(s) and date in the header comment to the Cell class and the LifeGrid class.

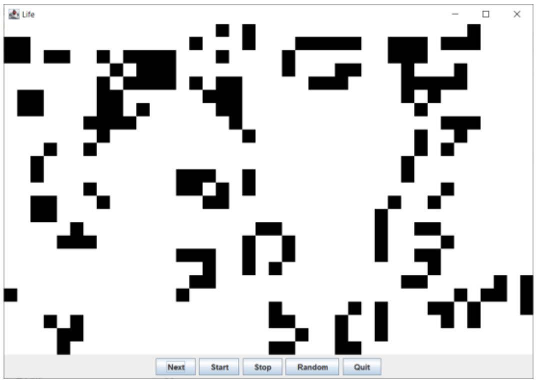


Lab03
JRE System Library [JavaS
src
(default package)
Cell.java
GameDriver.java
LifeGrid.java
LifePanel.java

Overview of Conway's "Game" of Life and Our Implementation

In this lab, you will explore Conway's Game of Life, which isn't actually a game (well, it's a zero-person game). The Game of Life is played on a grid of cells, each of which may be "alive" or "dead". Depending on the state of the game, each cell may either die or come to life at each pass, or *generation*. In our simulation, the black cells are alive and the white cells are dead.





You will be implementing the Cell class and the LifeGrid class. The LifePanel class and GameDriver classes have been completed for you.

Cell class

The game is played on a grid of cells. Each cell can either be alive or dead. Implement the Cell class as described here.

The Cell class has one private data field named `status`, a `boolean`, to represent whether the status of the cell is dead(false) or alive(true).

The Cell class has two constructors.

The first constructor has one parameter named `status`, a `boolean`, which is used to initialize the data field.
The second constructor randomly sets the cell's status to dead or alive (false or true) with a 50-50 chance. To do so, store a value into the data field status generated using the `nextBoolean`() method of the Random class:

```
status = randGenerator.nextBoolean()
```

The Cell class has one accessor method, `isAlive`, with no parameters and returns a boolean, the status of the cell.

- After implementing the Cell class, uncomment the test code at the end of the class. Save and run the test code.

Begin the LifeGrid class

The LifeGrid class represents the grid of cells. Each generation, each cell's status is updated depending on a set of rules.

The LifeGrid class has one private data field, board, which is a 2D array of Cell objects.

The LifeGrid class has one constructor with two int parameters, the number of rows and the number of columns of the board in that order. Instantiate the data field board using these dimensions. Recall that, when creating 2D arrays, the first dimension refers to the *number of rows* and the second dimension refers to the *number of columns*. After instantiating the 2D array, then fill the board with Cells. Each Cell is randomly dead or alive.

The LifeGrid class has the following instance methods:

fillRandom with no parameters that fills the board with Cells. Each Cell is randomly dead or alive. (Since these loops are also in the constructor, replace the redundant code in the constructor with a call to the fillRandom method.)

isCellAlive with two parameters, rowIndex and colIndex, returns whether the Cell at that location is alive or dead. Assume rowIndex and colIndex are valid.

- Though your code is not complete, you should begin to test your implementation of the LifeGrid class. Open and execute GameDriver. Each time you click on the Random button, a new random grid of cells should be displayed.

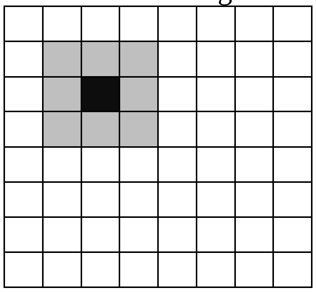
Counting Live Neighbors

You will now implement the instance methods in the LifeGrid class that determine if a Cell should live or die in the next generation. To do this, the number of living neighbors for any given Cell at rowIndex, colIndex needs to be counted.

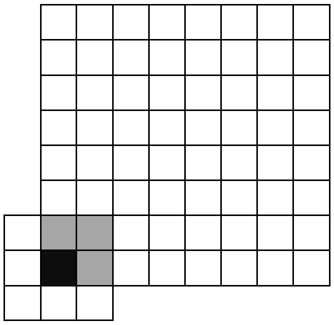
countLivingNeighbors accepts two parameters, rowIndex and colIndex, and returns an int, the number of living neighbors of the cell at rowIndex, colIndex. A cell "in the middle" of the board will have 8 neighbors, the 8 cells "touching" that cell. When a cell is on the border of the game board, however, that cell will not have 8 neighbors. If a cell doesn't have 8 neighbors, then count the number of living neighbors from the neighbors that the cell has. This method is a "helper" method that will only be used within the LifeGrid class and therefore should be declared **private.**

The countLivingNeighbors method will take some thought so that the code doesn't throw an ArrayIndexOutOfBoundsException. In my implementation, I wrote an additional private helper method named isCellInBoundsAndAlive which accepted the rowIndex and colIndex of a cell and returned true if the index values received were both within the bounds of the gameboard and the cell at the valid index values was alive. This method helped me "pretend" that each cell had 8 neighbors because it would return false if that "neighbor" didn't actually have valid index values.

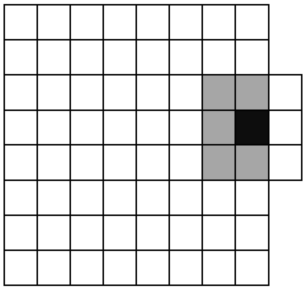
This cell has 8 neighbors.



This cell has only 3 neighbors
within the bounds of the gameboard.



This cell has only 5 neighbors
within the bounds of the gameboard.



- After implementing the countLivingNeighbors method, uncomment the main method at the bottom of the LifeGrid class. Execute the code. Since the cells in the grid are randomly dead or alive, you will need to look at the true/false values printed for the cells and the corresponding counts that your countLivingNeighbors method produces to decide whether your implementation is correct. Since you have not yet implemented the nextGeneration method, the output following the call to nextGeneration in the test code still have the original true/false values.

Updating Life or Death

The last step is to implement the nextGeneration method that updates the state of the board. Since you need to access the current state of every cell of the board to create its updated state, **you'll need a temporary array with the same dimensions as the current board to store the update**.

Follow these steps to implement the nextGeneration method which accepts no parameters and returns no value.

1. Declare and instantiate a 2D array named nextBoard with the same dimensions as your data field board.

2. Iterating over all the cells in board, for each cell determine whether the cell should be living or dead in the update array. These are the rules you must follow:

    a. Any living cell with fewer than two living neighbors dies (due to underpopulation or loneliness).

    b. Any living cell with more than three living neighbors dies (due to overcrowding).

    c. By inference, any living cell with exactly two or three living neighbors stays alive.

    d. Any dead cell with exactly three living neighbors becomes alive (slightly awkward reproduction)! Otherwise, the dead cell remains dead.

    Use your countLivingNeighbors method to get the number of living neighbors for the purpose creating the correct cell in nextBoard array.

3. After every cell of the updated array has been instantiated, then the data field board needs to be replaced with the new, updated board which is accomplished with the assignment statement

    ```
    board = nextBoard
    ```

- Test your implementation of nextGeneration using the main method in the LifeGrid class. Again, since the cells are randomly being set to true/false, you'll need to look at the output to determine if your nextGeneration method is correct.

- For the final test, execute GameDriver. At this point, you should be able to click the "Next" button to see a single generation. Every time you click "Next", your nextGeneration method is called. Every time you click "Random", your fillRandom method is called. Use the "Start" and "Stop" buttons to run through continuous generations and see your board evolve!

  Although it's random, if your board always stops changing after only 4-5 generations, something is probably off in your countLivingNeighbors method. My results consistently either never converged to a steady state or took over 20 generations to converge.

Finishing Up

Your program is expected to use variable names descriptive of the value stored and use the named constants given at the top of the file. You should have good indenting (Source|Format in Eclipse will fix that up), consistent use of curly braces, and judicious use of blank lines within your code to make it easy to follow (and score!). **The variable names in the parameter list of each method should match the variable names in the Javadoc comment.**

Scoring

Style Expectations	5 pts
Cell class	4 pts
LifeGrid class	21 pts
countLivingNeighbors	8 pts
nextGeneration	8 pts
other methods collectively	5 pts

When you are satisfied that you have met the requirements of the lab or have run out of time, upload `LifeGrid.java` and `Cell.java` from project Lab03 to Moodle. **Even if not completed, your code should have no syntax errors**. If you have syntax errors, comment out code so that the code you upload has no syntax errors. Only code that executes will be scored.

If you worked as a pair, make sure that both names are in the top comments. Only one partner needs to upload the completed files.