**Project summary**

Project 2 helps me explore 2D Arrays by implementing cellular automata to to simulate of Conway’s Game of Life. I created 2D arrays to represent the landscape grid of Cell object references and implemented Cell methods to manage the states of Cells.

Game of Cell presents the evolution of Cells, which is dependent on the states (alive/dead) of its neighboring Cells and rules of underproduction/reproduction. If the density of living cells is initialized randomly in the grid as I did in the project, the number of living cells will first decrease, and eventually living cells will exist in groups, either as still lifes (still patterns that don’t change along with generation update), or as periodic oscillators, or as shapes that keeps moving towards a direction in the grid.

**Task Solutions**

1. Cell class: it serves as an object for the grid in Landscape class and stroes the state of the Cell, whether it is alive or dead, represented by a boolean. It has several methods to reset its state to dead (false), set its state and get its state.
2. Landscape class: I first created a field of 2D array for the Landscape class, which will be used to hold Cell object references. The class has three parameters for its constructor, allowing the user to customize the number of rows and columns of its array field, which are also int fields of the Landscape class. similar to the Cell class, it includes several methods that return information of the class, such as getting the number of rows and columns, and getting the Cell located at a specific position. The getNeighbors method returns an arrayList of references of the 8 neighbors of a specific Cell. I first checked the boundaries of the landscape grid before indexing using nested for loops, as shown below.
3. Landscape Visualization: to visualize my Landscape, I first added a draw() method in both Landscape and Cell class, which will fill the cell to orange if it is alive, and black if it is dead.
4. Updating Cell States: I created another method in Cell class called updateState(), which takes in an ArrayList of Cells, which can be generated by calling the getNeighbor() method in the Landscape class. The updateState() method considers the states of neighboring Cells and decide whether a Cell should be alive or dead in the next generation. I first loop through the arraylist of its neighbors to collect the number of living cells and implement the rules.
5. Advancing the Game: to make the Cells move to next generation, I created the advance() method in Landscape class. The evolution of Cells should take place simultaneously, because the evolution is a mutual process; if the Cells are updated one at a time, the Cell that is updated earlier will affect the state of cells that gets updated afterwards. Therefore, I created a temporary memory of the Cell grid, copied over the Cells from the original Grid, and called updateState() on each Cell in the temporary memory, and shift the reference of the grid field (this.land) to the temporary memory, as shown below. Thus, the changes of all Cells will take place at the same time.
6. Life Simulation: I created the LifeSimulation class based on the model of LandscapeDisplay.main(). It contains a field of a Landscape object, and its constructor initializes the 2D grid in the Landscape object using values in the parameters as its number of rows and columns. Then it randomly activates Cells in the grid based on the density value passed in from the parameter. The main function of LifeSimulation first asks for command line argument as the number of rows and columns of the grid, and then uses the input to create a LifeSimulation object. To keep the evolution taking place, I created a for loop that iterates for 100 time maximum and repeatedly calls the Landscape().advance() and repaint() meothod of Landscape Display, and terminate the program by 250ms by calling Thread.sleep(250). Thus, we can observe the evolution that takes place in the grid. The gif below shows the Game.

**Extensions**

1. First, in order to test out the rules that I wrote for the evolution in the updateState() method of the Cell class, I created some special patterns in the cell grid that either oscillates throughout the evolution, or moving towards a direction. To do this effectively, I created another constructor of the LifeSimulation class that asks for String parameter that either calls to build a “blinker”, or a “spaceship”, each of which calls the respective build method that initializes that special shapes onto the grid.

Below is a snippet of the second constructor.

Below are gifs of the blinker and the spaceship throughout cell evolution.

1. Then I edited the main function of LifeSimultion to allow users customize more variables in the evolution. I created a Scanner object to ask for active user inputs of the height width, scale of the grid and the density of active cells with random initialization. Below is a snippet of the terminal with user input.

I tried the program several times with different initial density of active Cells, and found that a higher density of active cell doesn’t necessary prolong the evolution time until all Cells are in stable state (either in oscillation or still groups). A higher initial density may result in quicker decline of active Cell numbers due to overpopulation.

1. Then, I created a different type of cell, which is also a Cell class object, but is represented with a different color (gray). The int type field differentiates the two cell types – the orange (default cell) has 0 as the value for type, and the gray cell has 1. I set the default type field to be 0 (as shown below) to ensure that the programs that I wrote in the core project will still execute without the need to making a lot of changes, meaning that if I only want one type of cell in the grid, we can just initialize the program exactly like I did before. The way to change the type of cell is through the changeType(int t) method. I also updated the draw() method to draw the two type of cells differently.
2. Next, I updated the rule for evolution in the updateState () method. I first made sure that all the rules in the core project still stand, but only on the same color of cells. For example, a dead orange can only be activated if there are three alive orange cells around it, not the gray ones, and vice versa. The method first checks both numbers of living orange cells and grey cells in the neighbor list. In the rule-implementation section, I also first checked the type of the cell before implementing the same rules of evolution. I also added a new rule where if an active orange cell has more than 2 active gray cells around it, the orange cell will die. Below is the snippet of this new rule.
3. To test the new rule, I created the NewSimulation class, which is a child class of LifeSimulation so that it gets all its methods. I first modify the constructor of the child class, making it initialize both the total density of living cells (done by calling the parent constructor) and the density of the orange cells. Below the snippet of the child constructor. In NewSimulation, I also want to test java swing of JOptionPane, which can provide the users with a pop-up window for them to enter their desired values of grid width, height, scale, density of total alive cells, and the density of each type of cells, by using the JOptionPane.showInputDialog method.

This is a JOptionPane that asks for the density of alive cells.

Below is the gif showing the result of initializing the grid with 50% alive cell and 60% orange cell.

Observation: I notice that the number of gray cells will first increase because of the new rule, but then decrease to approximately the same as the orange cell, due to overpopulation.

1. Identify “lost” dynamically allocated memory: whenever the reference of an object changes its pointer, it causes a lost of the original data that the reference points to, because the reference is updated to point to a new data. Below is an example in the advanve() method of the Landscape class, where the field this.land points to new data newland, therefore loses its old memory.

**Conclusion**

In this project, I learnt how to create and mange 2D arraylist, and how to manage dynamic memory and use it to advance the evolution of cells. I also learnt how to use java Scanner and JOptionPane to better user experience.

Credits:

Caleb, Michael (TA), Professor Layton