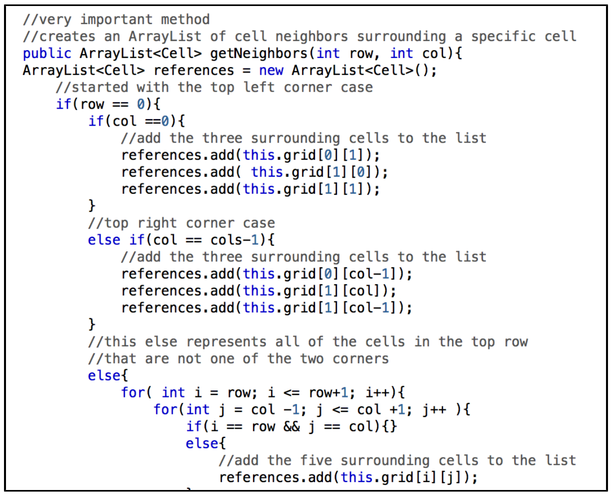
Project 2

<u>PROJECT 2</u>

In this project, we begin work on a multi-week project using 2D grids. We are asked to create a life simulation that has a series of cells in a grid landscape. These cells were each given a boolean status of being alive or dead. We also had the cells interact with each other to create multiple life simulations. Depending on the life status of the cell and its surrounding neighbors, the cell could change in the next simulation. This causes different generations of life. To do this, we had to create three class. Overall, this project tested and improved my knowledge of Java arrays, ArrayLists, for loops, double-nested for loops, conditionals, and constructors.

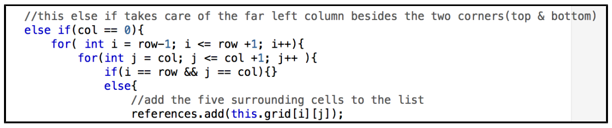
Task 1. In this task, we are asked to create a Cell class. The cell object represents a location on a grid and should store the data of its row location, its column location and its boolean status of whether the cell is alive or not. To do this, we had to implement a number of methods. Some of these methods included setAlive() which sets the boolean life status, setPosition() which sets the x and y location of the cell in the grid, and isAlive() which simply returns the boolean life status of the cell--whether the cell is dead or alive.

Task 2. In this task, we were asked to create a Landscape class that holds Cell objects in a 2D array grid. To do this, in my public Landscape class, I initialize a new grid of type Cells with a given amount of rows and columns. Then, I loop through all the rows and columns of the grid and add a new Cell object to each spot. I automatically set the Cells life status to dead. Some important methods in this task include: getCell() which simply returns a reference to a specific cell, toString() which creates a visual representation of the grid landscape, and finally ArrayList getNeighbors(). This method is very important. The goal of this last method is to take the given cell and create an ArrayList of all its surrounding neighbors and their boolean life status values. To do this, I broke the grid into specific sections. Here are a couples snippets of my code:



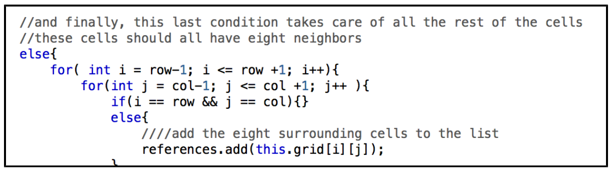
```java
//very important method
//creates an ArrayList of cell neighbors surrounding a specific cell
public ArrayList<Cell> getNeighbors(int row, int col){
ArrayList<Cell> references = new ArrayList<Cell>();
    //started with the top left corner case
    if(row == 0){
        if(col ==0){
            //add the three surrounding cells to the list
            references.add(this.grid[0][1]);
            references.add( this.grid[1][0]);
            references.add(this.grid[1][1]);
        }
        //top right corner case
        else if(col == cols-1){
            //add the three surrounding cells to the list
            references.add(this.grid[0][col-1]);
            references.add(this.grid[1][col]);
            references.add(this.grid[1][col-1]);
        }
        //this else represents all of the cells in the top row
        //that are not one of the two corners
        else{
            for( int i = row; i <= row+1; i++){
                for(int j = col -1; j <= col +1; j++ ){
                    if(i == row && j == col){}
                    else{
                        //add the five surrounding cells to the list
                        references.add(this.grid[i][j]);
```

Here, I am focusing on only the top row of the grid. The top right and left corners should only have three neighbors, and every other cell on the top row of the grid should only have 5 neighbors. I used conditionals to identify each cell and then add the correct neighbors to the ArrayList.



```java
//this else if takes care of the far left column besides the two corners(top & bottom)
else if(col == 0){
    for( int i = row-1; i <= row +1; i++){
        for(int j = col; j <= col +1; j++ ){
            if(i == row && j == col){}
            else{
                //add the five surrounding cells to the list
                references.add(this.grid[i][j]);
```

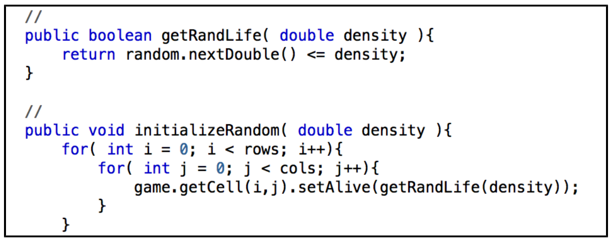
Here, I am focusing on the first column, but excluding the top and bottom corners because I accounted for those early in the previous code. I loop through all the cells in the first column and add the correct 5 neighbors to the ArrayList.



```java
//and finally, this last condition takes care of all the rest of the cells
//these cells should all have eight neighbors
else{
    for( int i = row-1; i <= row +1; i++){
        for(int j = col-1; j <= col +1; j++ ){
            if(i == row && j == col){}
            else{
                ////add the eight surrounding cells to the list
                references.add(this.grid[i][j]);
```

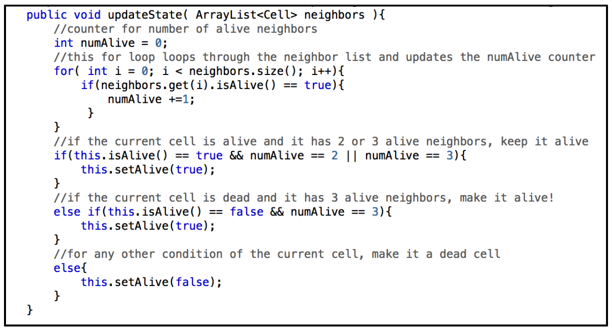
Finally, here I am focusing on everything inside the middle of the grid. These cells should have 8 surrounding neighbors.

Task 3. In this task, we had to create a LifeSimulation class. This class will run a Game of Life simulation on a grid Landscape. To do this, I had to initialize a Landscape of a given size and then use a random method to determine what proportion of the cells would be set to alive. Thus, I created the initializeRandom(double density) method. First, I created a method called getRandLife() that simply took a density value and returned true if it was less than a random double. Then, I looped through every cell in my initializeRandom() method and used that getRandLife() method to randomly determine if the cell should be alive or dead. All the while, the proportion of alive to dead stayed at the value of my density parameter! Here is a snippet of my code:



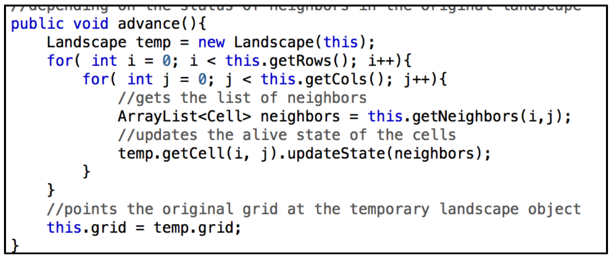
```
 //
 public boolean getRandLife( double density ){
     return random.nextDouble() <= density;
 }

 //
 public void initializeRandom( double density ){
     for( int i = 0; i < rows; i++){
         for( int j = 0; j < cols; j++){
             game.getCell(i,j).setAlive(getRandLife(density));
         }
     }
```

Task 4. This task asked me to add a new method to my Cell class that updated the life status of the current cell depending on its current and status and the status of its surrounding neighbors. First off, I created a counter for the number of alive cells in the ArrayList of neighbors for the current cell. Then, I added three rules with conditionals. If a living cell has two or three living neighbors, it will stay alive. If a dead cell has three living neighbors it will become alive, and any other cell will become dead. Here is my code for this method:



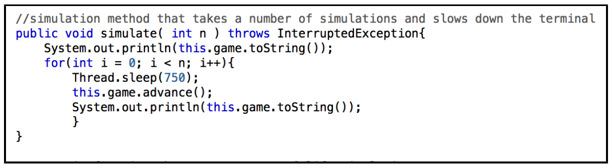
```
 public void updateState( ArrayList<Cell> neighbors ){
     //counter for number of alive neighbors
     int numAlive = 0;
     //this for loop loops through the neighbor list and updates the numAlive counter
     for( int i = 0; i < neighbors.size(); i++){
         if(neighbors.get(i).isAlive() == true){
             numAlive +=1;
          }
     }
     //if the current cell is alive and it has 2 or 3 alive neighbors, keep it alive
     if(this.isAlive() == true && numAlive == 2 || numAlive == 3){
         this.setAlive(true);
     }
     //if the current cell is dead and it has 3 alive neighbors, make it alive!
     else if(this.isAlive() == false && numAlive == 3){
         this.setAlive(true);
     }
     //for any other condition of the current cell, make it a dead cell
     else{
         this.setAlive(false);
     }
 }
```

Task 5. This task then asked me to make the Game of Life advance after the simulation has ended and updated the state of each cell. To do this, I created a temporary Landscape that was a true copy of the original Landscape. Then, I looped through the cells of the original grid, got their information and then updated the states of the cells in the temp grid. At the end, I then created a pointer to the temporary grid thus making it the new and original grid. This is how you can advance on and run more than one simulation. Here is my code for this advance() method:



```
//depending on the status of neighbors in the original landscape
public void advance(){
    Landscape temp = new Landscape(this);
    for( int i = 0; i < this.getRows(); i++){
        for( int j = 0; j < this.getCols(); j++){
            //gets the list of neighbors
            ArrayList<Cell> neighbors = this.getNeighbors(i,j);
            //updates the alive state of the cells
            temp.getCell(i, j).updateState(neighbors);
        }
    }
    //points the original grid at the temporary landscape object
    this.grid = temp.grid;
}
```

Task 6. This task asked me to make a simulate method that advances the grid through a given number of simulations. Also, we used a Thread.sleep() function to slow down the terminal as it printed the toString() representation of each simulations grid. Here is my code for this method:

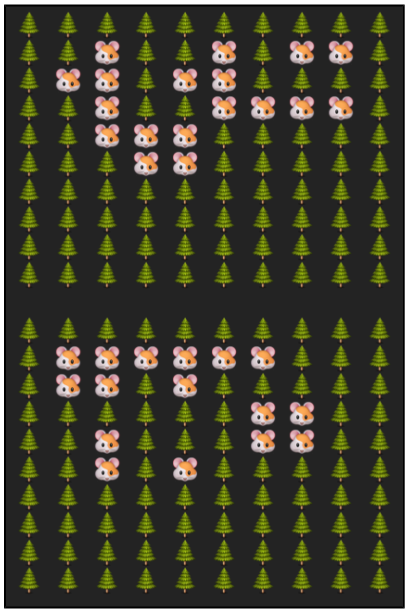


```
//simulation method that takes a number of simulations and slows down the terminal
public void simulate( int n ) throws InterruptedException{
    System.out.println(this.game.toString());
    for(int i = 0; i < n; i++){
        Thread.sleep(750);
        this.game.advance();
        System.out.println(this.game.toString());
        }
}
```

Task 7. Finally, this task created a main method that was to run the simulation a number of times and of a specific size. Here is my main method:

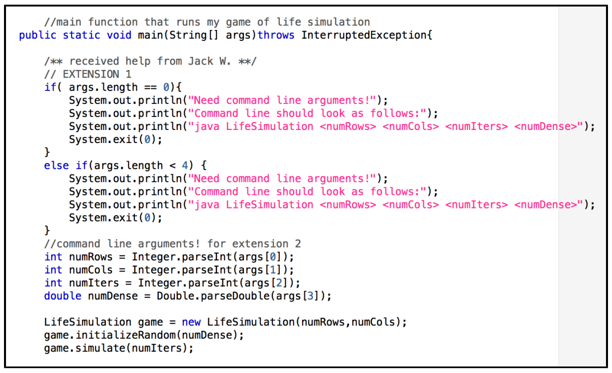
LifeSimulation game = new LifeSimulation(numRows,numCols); game.initializeRandom(numDense); game.simulate(numIters);
LifeSimulation game = new LifeSimulation(numRows,numCols);
game.initializeRandom(numDense);
game.simulate(numIters);

The parameters are command line arguments. I did this for my first extension. Here is a snippet of what my simulation generated. The emojis have to do with my second extension:



```
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🐹 🌲 🌲 🐹 🌲 🐹 🐹 🌲
🌲 🐹 🐹 🌲 🐹 🐹 🌲 🌲 🌲 🌲
🌲 🌲 🐹 🌲 🌲 🐹 🐹 🐹 🐹 🌲
🌲 🌲 🐹 🐹 🐹 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🐹 🐹 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲

🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🐹 🐹 🐹 🐹 🐹 🐹 🌲 🌲 🌲
🌲 🐹 🐹 🌲 🐹 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🐹 🐹 🌲 🌲
🌲 🌲 🐹 🌲 🌲 🌲 🐹 🐹 🌲 🌲
🌲 🌲 🐹 🌲 🐹 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲 🌲
```

Extension 1. For my first extension, I implemented command line arguments. I created command line arguments for the number of rows, the number of columns, the number of simulations, and the density value. To create the command line arg for the density value I had to use Double.parseDouble(). Here is a snippet of my code for my extension:



```java
    //main function that runs my game of life simulation
public static void main(String[] args)throws InterruptedException{

    /** received help from Jack W. **/
    // EXTENSION 1
    if( args.length == 0){
        System.out.println("Need command line arguments!");
        System.out.println("Command line should look as follows:");
        System.out.println("java LifeSimulation <numRows> <numCols> <numIters> <numDense>");
        System.exit(0);
    }
    else if(args.length < 4) {
        System.out.println("Need command line arguments!");
        System.out.println("Command line should look as follows:");
        System.out.println("java LifeSimulation <numRows> <numCols> <numIters> <numDense>");
        System.exit(0);
    }
    //command line arguments! for extension 2
    int numRows = Integer.parseInt(args[0]);
    int numCols = Integer.parseInt(args[1]);
    int numIters = Integer.parseInt(args[2]);
    double numDense = Double.parseDouble(args[3]);

    LifeSimulation game = new LifeSimulation(numRows,numCols);
    game.initializeRandom(numDense);
    game.simulate(numIters);
```

Now, if i write the command line in my terminal of this "java LifeSimulation 10 10 10 0.4 > Life.txt" and export it into a .txt file. I will get a 10 simulations of a 10x10 grid with 0.4 living density. Here is the .txt file:

Life.txt

Extension 2. For this extension, I used emojis to diversify the visual representation of my simulations. First off, I did part of this extension in my project by using the Hamster and the Trees. I simply just replaced the X and "<space> " for the alive and dead representations. The first part of extension 2 is in my life.txt file above. This is a bunch of hamsters getting lost in the forrest. Another part of this extension is in this file:

Extension2a.txt

This one represents the population of alligators on a topical island. Lastly, one final extension represents the life or death of a fresh egg. It either ends up as a chick(alive) or a fried egg(dead). Here is that simulation text:

Extension2b.txt

Conclusion. Overall, this project taught me how to use arrays to make 2D grids. I became familiar with interactions between parts of 2D grids and how to create Conway's Way of Life simulations. I also gained more experience with my ArrayList skills. In the end, I really furthered my Java knowledge. It is a new language to me and the projects are helping it become clearer and clearer. Such a useful language.

Who helped me? I recieved help from Dale during his Monday office hours. Also, I received help from both CP Majgaard and Jack Walpuck. I worked alongside and collaborated with fellow CS231 students Julia Saul and Steven Parrot throughout the project.