Conway's Game of Life:Adithya Shastry

Introduction:

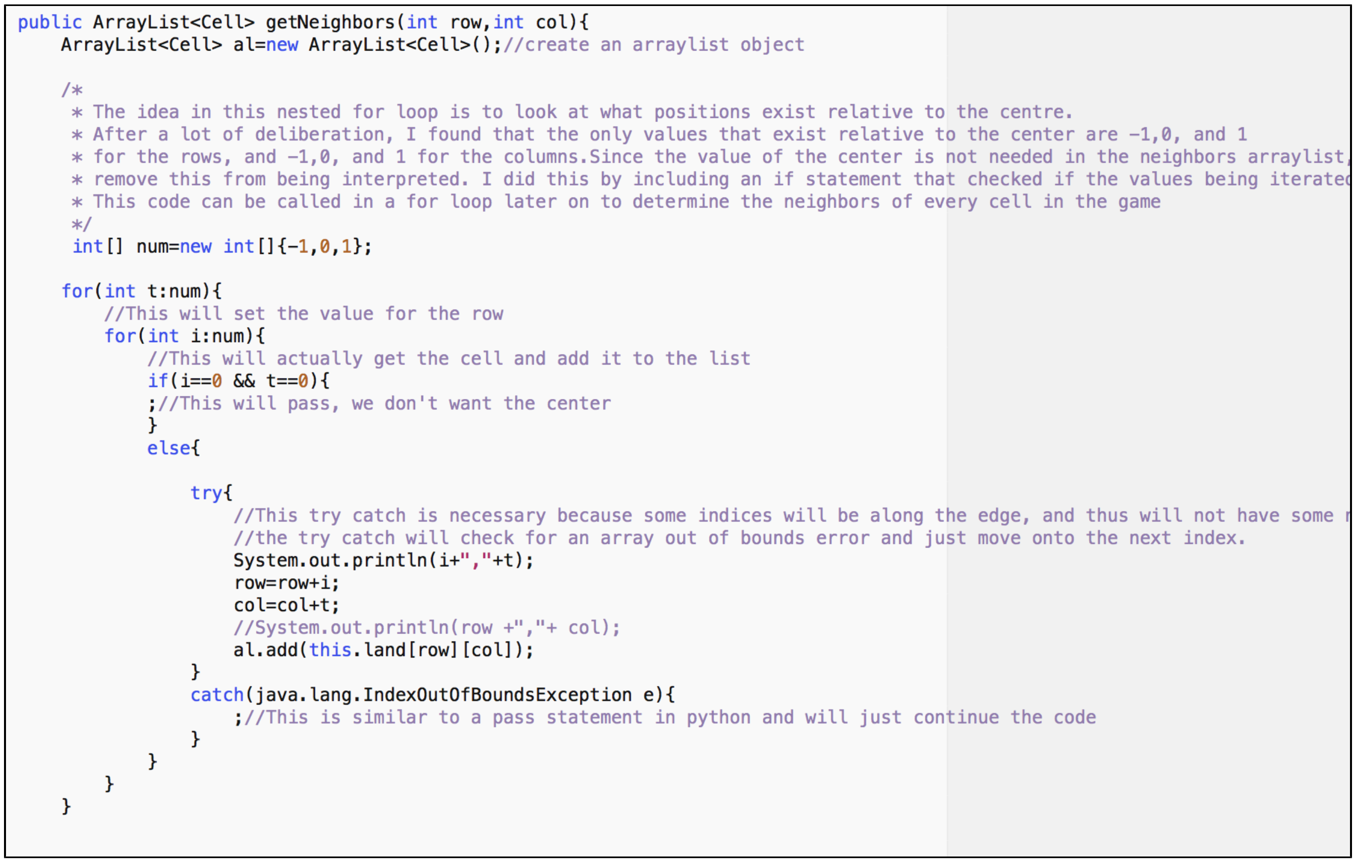
This project creates a simulation of life known as Conway's game of life. This game of life has very simple rules, but is very interesting because of the very similar structures that it produces no matter the conditions. This project allowed me to understand very interesting topics about the way java handles data and develop my debugging skills because of the logical issues that came up throughout my working on this project.

Cell Class:

The Cell class was very easy to construct because this class would hold a Boolean value of true or false, or alive or dead respectively.

Landscape Class:

The landscape class created a 2D array that held all the cells in the game. Things were done to the Cells based on the circumstances that each Cell is under. This class caused a lot of issue for me because of the getNeighbors() method that the class had me implement. This method caused some trouble because I wanted to create an efficient method to get the neighbors of the class. In the end, I created a list that held three values: -1,0,and 1. This was all that was needed, and would cover all of the neighbors that a Cell would have. The interesting part,however,was the issue of how to deal with indices that were outside of the array. Since not every position in the array will have 8 neighbors, it is important to incorporate this into the program. I could have done this with if else statements, but decided not to because that would be rather inefficient from my perspective. I instead used a try-catch block that would skip any positions in the array that returned an out of bounds error. The code can be seen below:

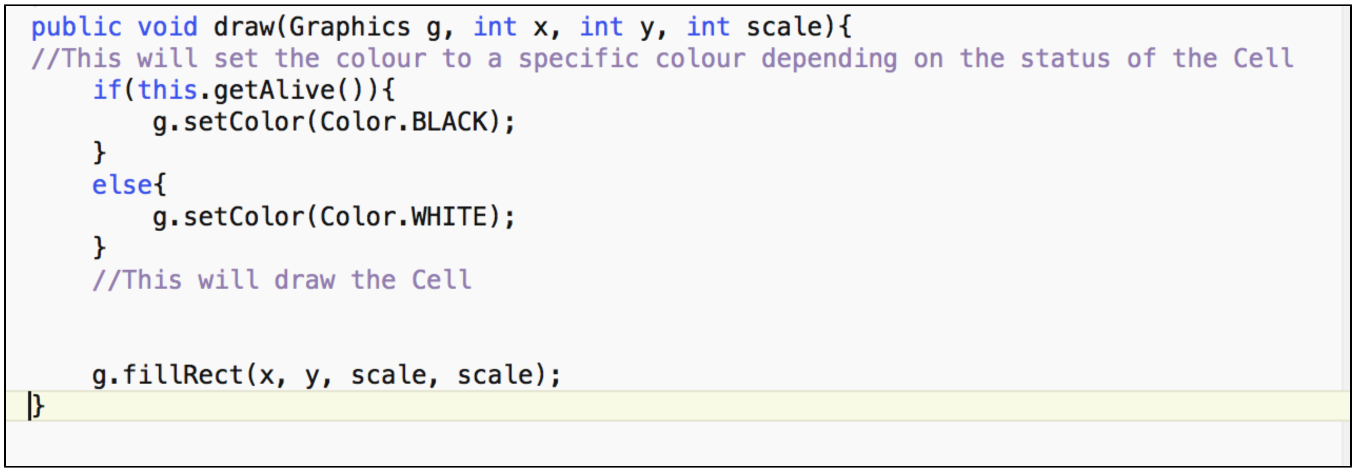


```java
public ArrayList<Cell> getNeighbors(int row,int col){
    ArrayList<Cell> al=new ArrayList<Cell>();//create an arraylist object

    /*
     * The idea in this nested for loop is to look at what positions exist relative to the centre.
     * After a lot of deliberation, I found that the only values that exist relative to the center are -1,0, and 1
     * for the rows, and -1,0, and 1 for the columns.Since the value of the center is not needed in the neighbors arraylist,
     * remove this from being interpreted. I did this by including an if statement that checked if the values being iterated
     * This code can be called in a for loop later on to determine the neighbors of every cell in the game
     */
     int[] num=new int[]{-1,0,1};

    for(int t:num){
        //This will set the value for the row
        for(int i:num){
            //This will actually get the cell and add it to the list
            if(i==0 && t==0){
            ;//This will pass, we don't want the center
            }
            else{

                try{
                    //This try catch is necessary because some indices will be along the edge, and thus will not have some n
                    //the try catch will check for an array out of bounds error and just move onto the next index.
                    System.out.println(i+","+t);
                    row=row+i;
                    col=col+t;
                    //System.out.println(row +","+ col);
                    al.add(this.land[row][col]);
                }
                catch(java.lang.IndexOutOfBoundsException e){
                    ;//This is similar to a pass statement in python and will just continue the code
                }
            }
        }
    }
```

The rest of the landscape class was relatively easy to program since not too much logic was necessary. There was however a lot of logic when it came to getting the neighbors and advancing the game forward to the next generation. This will be talked about below.

Landscape Display:

The landscape display class was provided to us therefore we did not have to code anything ourselves, except for the draw method in the Cell class. the code for this can be seen below:



```
public void draw(Graphics g, int x, int y, int scale){
//This will set the colour to a specific colour depending on the status of the Cell
    if(this.getAlive()){
        g.setColor(Color.BLACK);
    }
    else{
        g.setColor(Color.WHITE);
    }
    //This will draw the Cell


    g.fillRect(x, y, scale, scale);
}
```

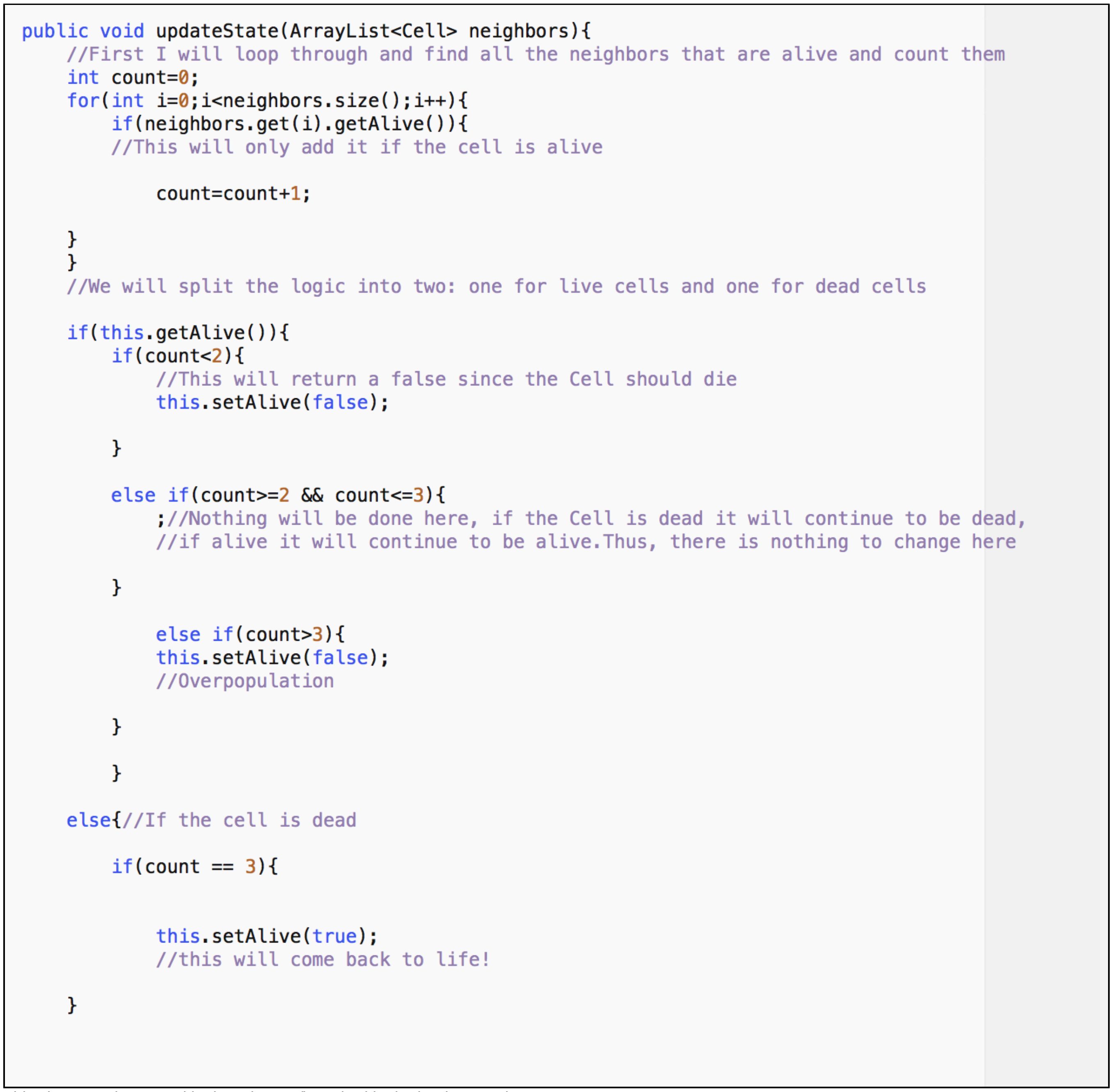
One thing that is very interesting about the setColor() method in the graphics class is that the method will change the color being used to draw on the canvas. Since this is the case, the color of the pen should be set first then the shape should actually be drawn. This is what the if else statement looks at and changes the color of the pen accordingly.

Updating Cell Status and Advancing the Game:

The updating Cell status included the getState() method in the Cell class and the advance() method in the landscape class. Both of these methods required lots of logic and thus will be discussed below.

getState() method:

The getState() method in the Cell class took the neighbors arraylist as an input and used it to compute the outcome of the cell in the next generation. The idea here was to create a count of the number of neighbors that were alive. This number was then used to complete the logic based on the four rules that the game uses to simulate life. Before this however, I split the logic into two sections depending on if the main cell,the one the getState() is running on, was alive or dead. This was done because one of the rules was only to be run if the cell was dead, specifically the reproduction rule. This rule made the Cell come become alive if the cell had exactly three neighbors that were alive. This rule was just easy to run by splitting the decision making into two types. Once this was done, everything is pretty straightforward because I just used if then logic statements to closely follow the rules. This method can be seen below:

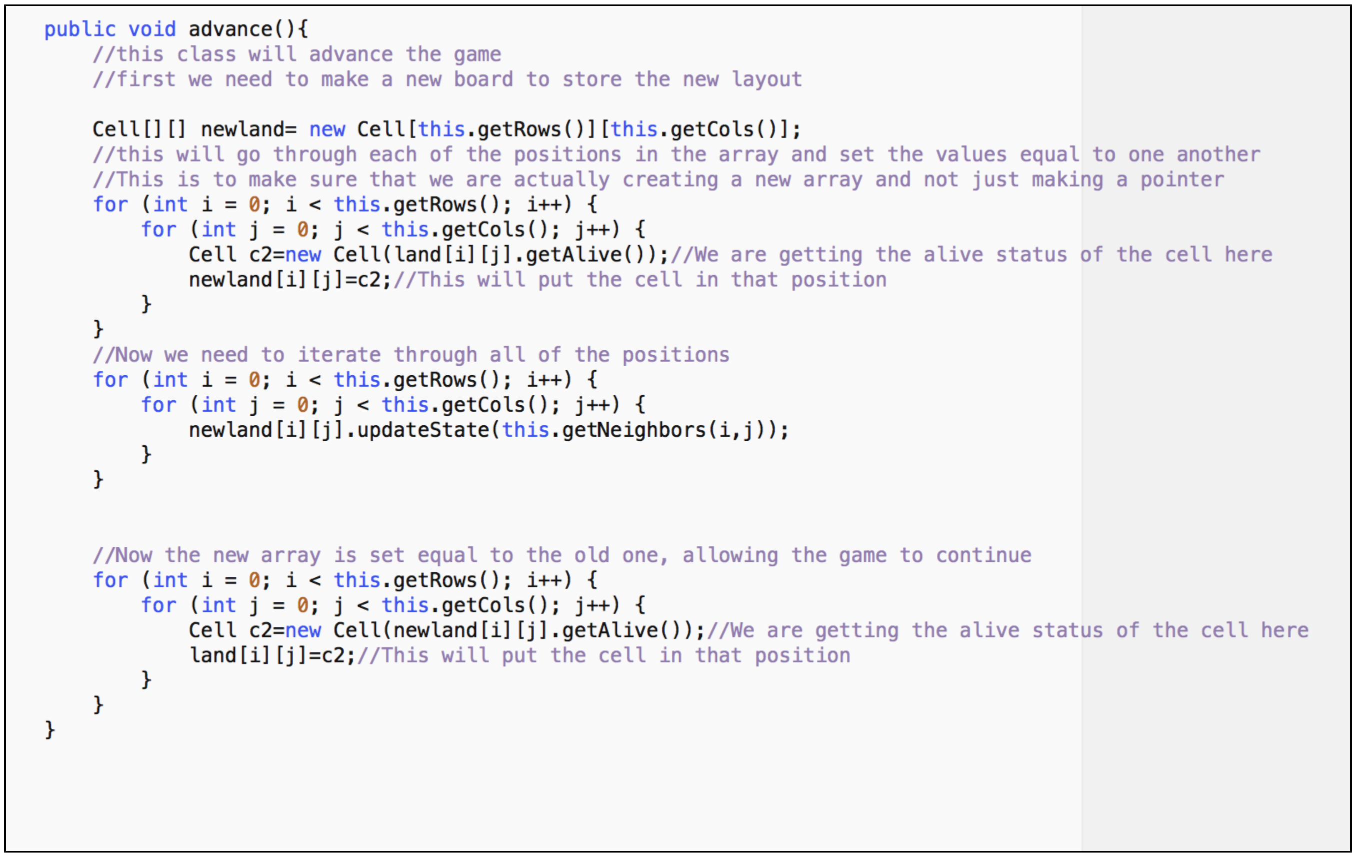


```java
public void updateState(ArrayList<Cell> neighbors){
  //First I will loop through and find all the neighbors that are alive and count them
  int count=0;
  for(int i=0;i<neighbors.size();i++){
    if(neighbors.get(i).getAlive()){
    //This will only add it if the cell is alive

      count=count+1;

  }
  }
  //We will split the logic into two: one for live cells and one for dead cells

  if(this.getAlive()){
    if(count<2){
      //This will return a false since the Cell should die
      this.setAlive(false);

    }

    else if(count>=2 && count<=3){
      ;//Nothing will be done here, if the Cell is dead it will continue to be dead,
      //if alive it will continue to be alive.Thus, there is nothing to change here

    }

      else if(count>3){
      this.setAlive(false);
      //Overpopulation

    }

    }

  else{//If the cell is dead

    if(count == 3){


      this.setAlive(true);
      //this will come back to life!

  }
```

This class was later used in the advance() method in the landscape class.

advance() method:

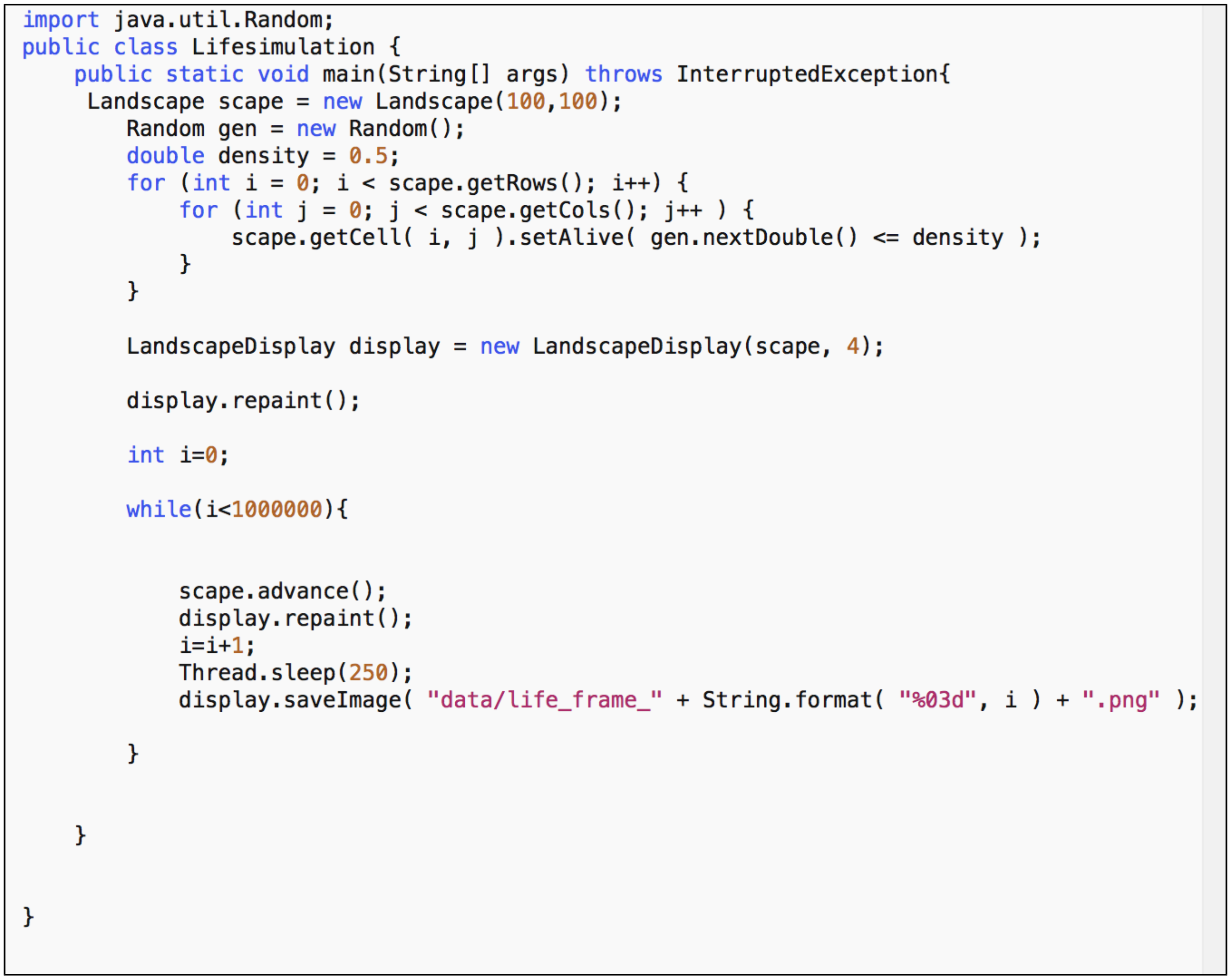
The advance method will run through every cell in the landscape and update the Cell's status depending on the output. This was a very interesting endevour because it required that all the Cells be updated at the same time. I accomplished this by creating a new array which was a copy of the actual one,called land. I quickly found out that just equating these two arrays would just create a pointer instead of actually allocating a new space for another array. This isn't what I wanted so I had to create a nested for loop that looped through the entire array and set the alive status of each cell equal to the corresponding one in the original array,land. Once this was done, I needed to actually update the status of each cell. This was done by looping through the Cell objects in the new array, newland, and updating the states depending on the inputs from the original array,land. This made sure that the Cells were not affected by the status change of the cells done before them and were instead done based on the input of the previous generation. Then the land was set equal to newland since land is the one that actually exists within the class. This was done with another nested for loop, which is overkill because a pointer would have worked perfectly, I just wanted to save some memory which this method does since it overwrites an array that already exists instead of just having arrays that aren't being used stored every time the game advances. The code for this method can be seen below:



```java
public void advance(){
    //this class will advance the game
    //first we need to make a new board to store the new layout

    Cell[][] newland= new Cell[this.getRows()][this.getCols()];
    //this will go through each of the positions in the array and set the values equal to one another
    //This is to make sure that we are actually creating a new array and not just making a pointer
    for (int i = 0; i < this.getRows(); i++) {
        for (int j = 0; j < this.getCols(); j++) {
            Cell c2=new Cell(land[i][j].getAlive());//We are getting the alive status of the cell here
            newland[i][j]=c2;//This will put the cell in that position
        }
    }
    //Now we need to iterate through all of the positions
    for (int i = 0; i < this.getRows(); i++) {
        for (int j = 0; j < this.getCols(); j++) {
            newland[i][j].updateState(this.getNeighbors(i,j));
        }
    }


    //Now the new array is set equal to the old one, allowing the game to continue
    for (int i = 0; i < this.getRows(); i++) {
        for (int j = 0; j < this.getCols(); j++) {
            Cell c2=new Cell(newland[i][j].getAlive());//We are getting the alive status of the cell here
            land[i][j]=c2;//This will put the cell in that position
        }
    }
}
```

Life Simulation Class:

This class was relatively straightforward because most of the code was provided for us, with the exception of the throws verb that had to be used as the Thread.sleep() method threw it. A TA helped me realize this issue and explained to me what it meant. This is a very interesting topic because it goes down to how the java compiler was actually created. In essence the throws verb tells programmers that a certain exception might be thrown because of how the method was created. This is very interesting because it means that someone actually programmed this into the java compiler at a very low level, making it easy for us programmers understand what is actually happening in our code and what we need to fix. The code can be seen below:

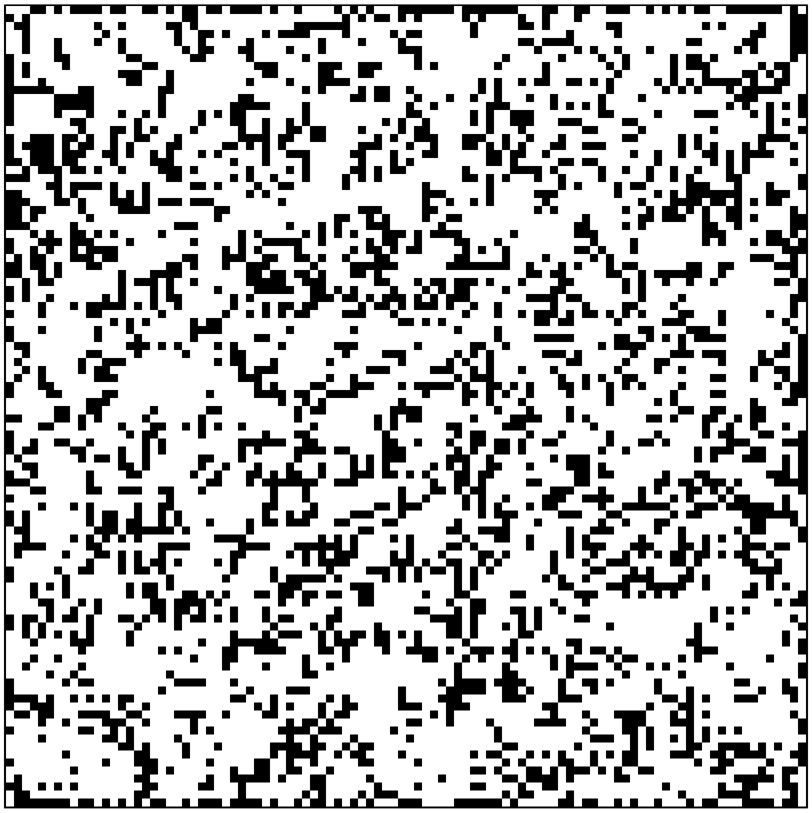


```java
import java.util.Random;
public class Lifesimulation {
    public static void main(String[] args) throws InterruptedException{
     Landscape scape = new Landscape(100,100);
        Random gen = new Random();
        double density = 0.5;
        for (int i = 0; i < scape.getRows(); i++) {
            for (int j = 0; j < scape.getCols(); j++ ) {
                scape.getCell( i, j ).setAlive( gen.nextDouble() <= density );
            }
        }

        LandscapeDisplay display = new LandscapeDisplay(scape, 4);

        display.repaint();

        int i=0;

        while(i<1000000){


            scape.advance();
            display.repaint();
            i=i+1;
            Thread.sleep(250);
            display.saveImage( "data/life_frame_" + String.format( "%03d", i ) + ".png" );

        }


    }


}
```

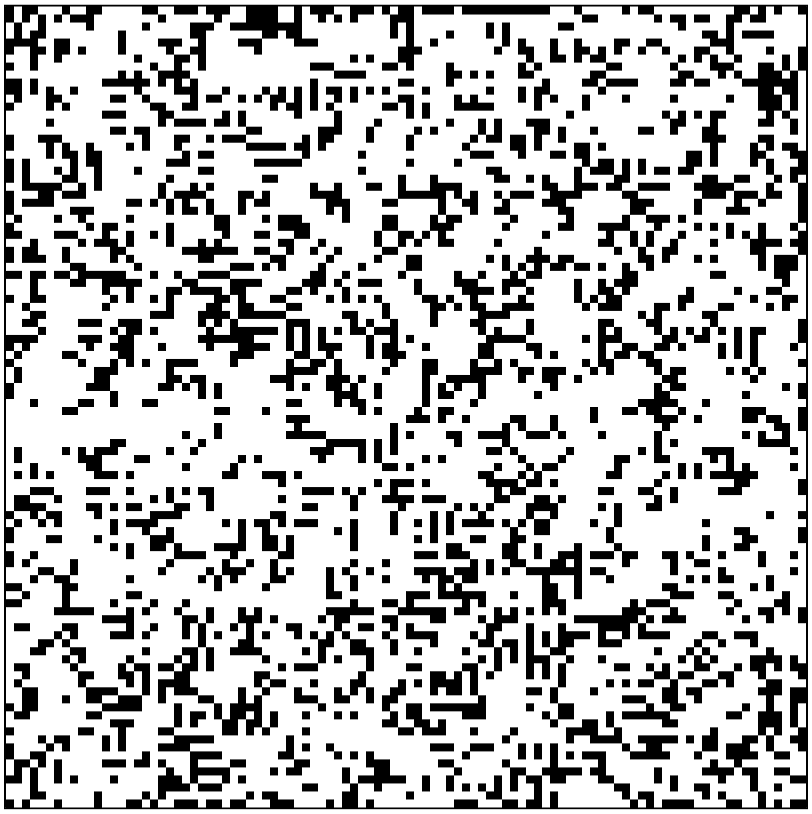
Final Output of the Simulation:

These gifs show the final output of the simulation. The only way to check to see if the output is correct is to see that the pattern types, which are found in the wikipedia page for the game, work in a similar way.

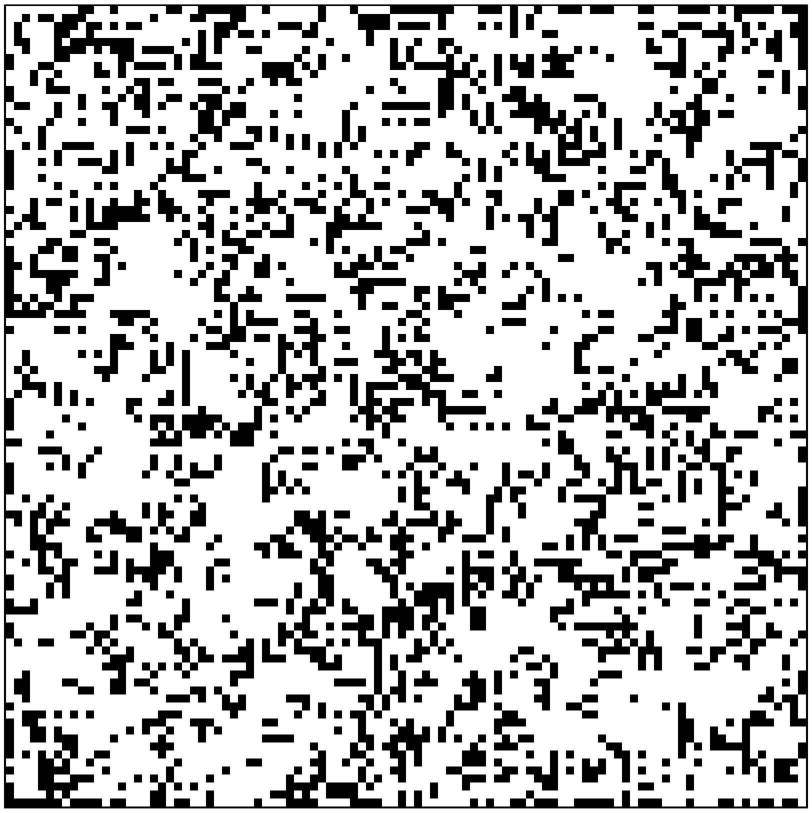
Output 1:



Output 2:



Output 3:



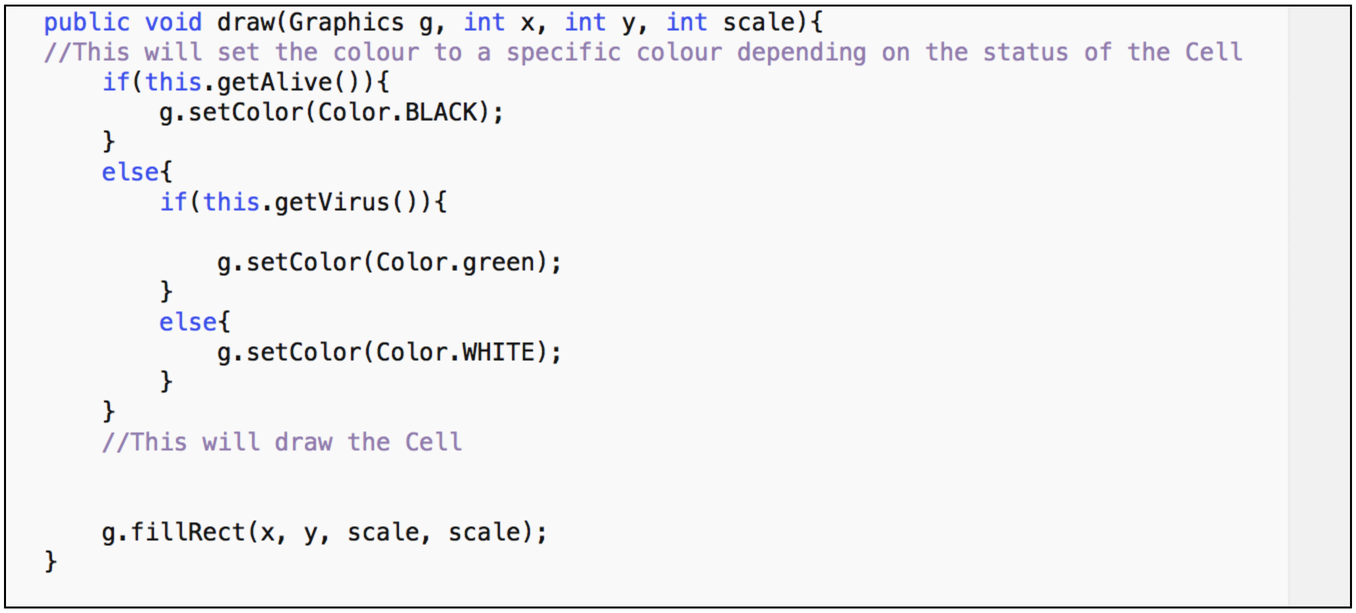
Extensions:

Creating a Virus Cell:

This Virus cell will be added as a Boolean in the Cell class and will turn any normal cells into virus cells if it is a neighboring cell. The code can be seen below:

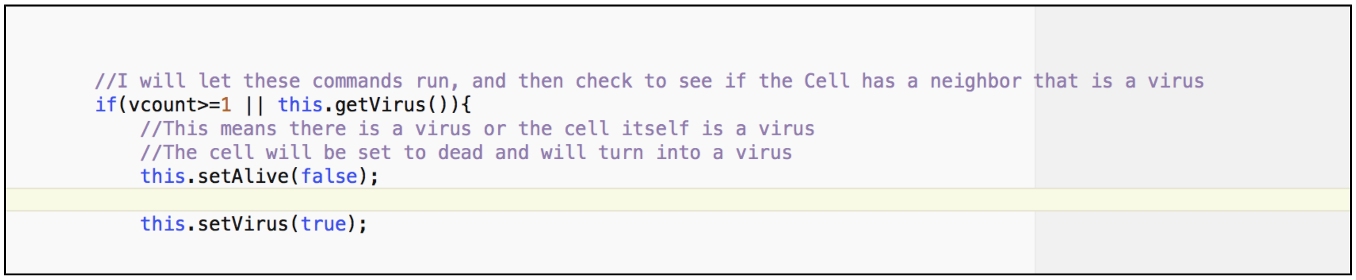
```
boolean virus;//This virus boolean will check to see if the cell is a virus


public boolean getVirus(){
//this method will check to see if the cell is a virus or not
    return this.virus

}

public void setVirus(boolean virus){
    this.virus=virus;
    if(virus){
        //If this is a virus then the cell is really dead
        this.alive=false;

    }

}
```

The next step will be to color the Cell differently so it is easy to visualize in the output.



```java
public void draw(Graphics g, int x, int y, int scale){
//This will set the colour to a specific colour depending on the status of the Cell
    if(this.getAlive()){
        g.setColor(Color.BLACK);
    }
    else{
        if(this.getVirus()){

            g.setColor(Color.green);
        }
        else{
            g.setColor(Color.WHITE);
        }
    }
    //This will draw the Cell


    g.fillRect(x, y, scale, scale);
}
```

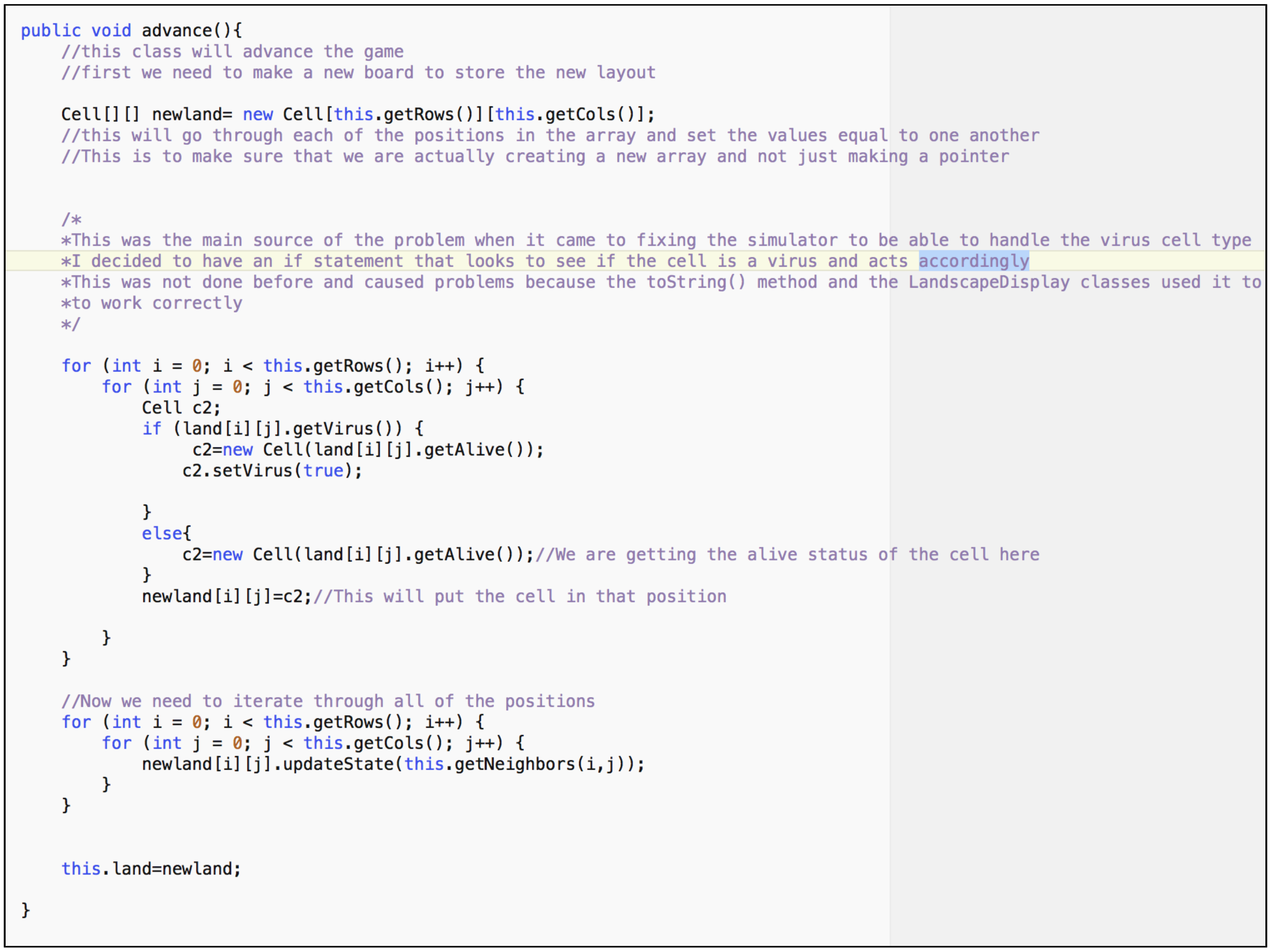
Next, I will change the getState() method to account for the virus. This can be seen below:



```java
    //I will let these commands run, and then check to see if the Cell has a neighbor that is a virus
    if(vcount>=1 || this.getVirus()){
        //This means there is a virus or the cell itself is a virus
        //The cell will be set to dead and will turn into a virus
        this.setAlive(false);

        this.setVirus(true);
```

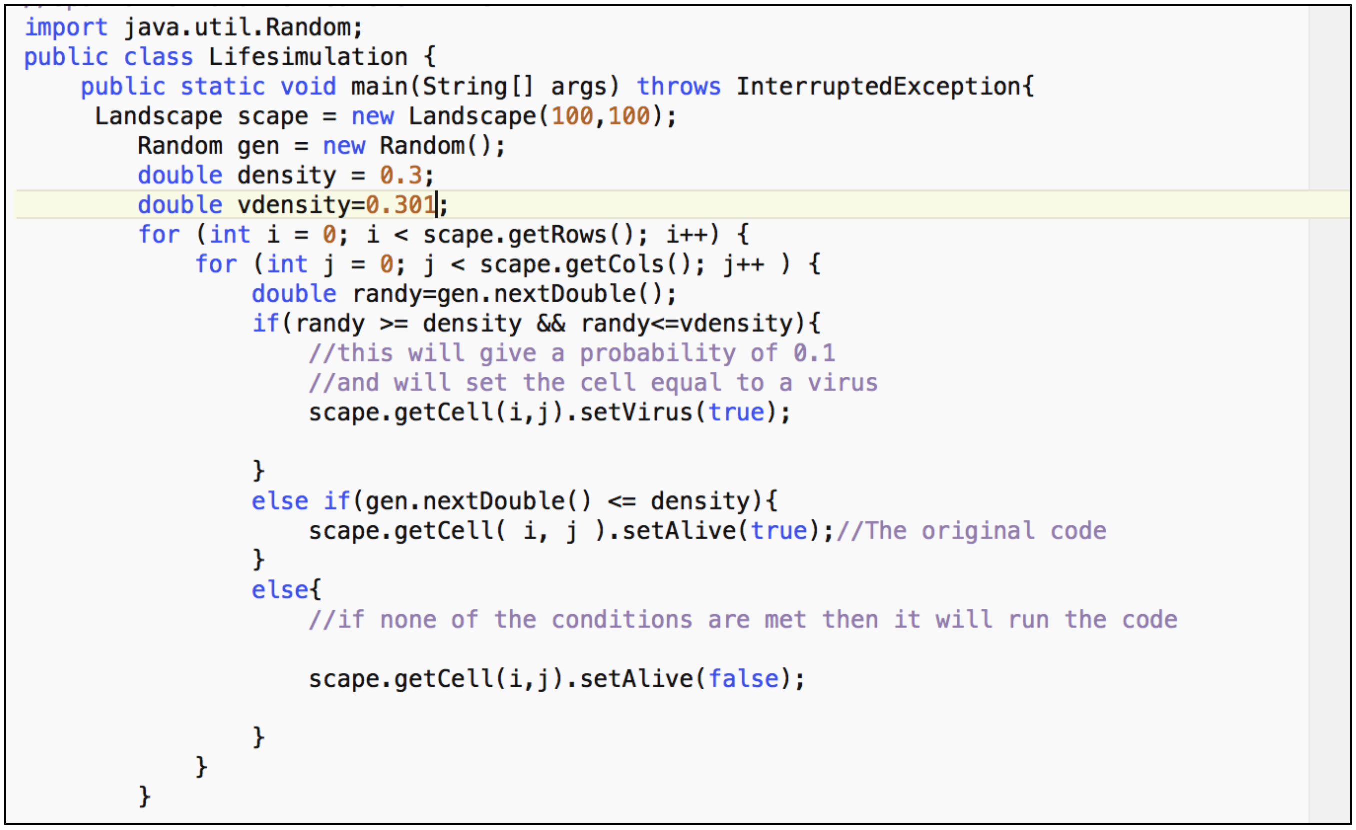
This will not affect the actual logic of the getState() method, because it is run after the initial logic of the original game. After this is done, the logic of the virus will take effect:the virus will kill any neighboring cells and make them into viruses.

After up to 2 hours of debugging, with a TA, we figured out that the advance method also needed to change because it was not taking into account the virus cell type. This was a very interesting endevour that I believe has improved my debugging abilities, a skill that I need to improve on greatly.The code can be seen below:



```
public void advance(){
    //this class will advance the game
    //first we need to make a new board to store the new layout

    Cell[][] newland= new Cell[this.getRows()][this.getCols()];
    //this will go through each of the positions in the array and set the values equal to one another
    //This is to make sure that we are actually creating a new array and not just making a pointer


    /*
    *This was the main source of the problem when it came to fixing the simulator to be able to handle the virus cell type
    *I decided to have an if statement that looks to see if the cell is a virus and acts accordingly
    *This was not done before and caused problems because the toString() method and the LandscapeDisplay classes used it to
    *to work correctly
    */

    for (int i = 0; i < this.getRows(); i++) {
        for (int j = 0; j < this.getCols(); j++) {
            Cell c2;
            if (land[i][j].getVirus()) {
                 c2=new Cell(land[i][j].getAlive());
                c2.setVirus(true);

            }
            else{
                c2=new Cell(land[i][j].getAlive());//We are getting the alive status of the cell here
            }
            newland[i][j]=c2;//This will put the cell in that position

        }
    }

    //Now we need to iterate through all of the positions
    for (int i = 0; i < this.getRows(); i++) {
        for (int j = 0; j < this.getCols(); j++) {
            newland[i][j].updateState(this.getNeighbors(i,j));
        }
    }


    this.land=newland;

}
```

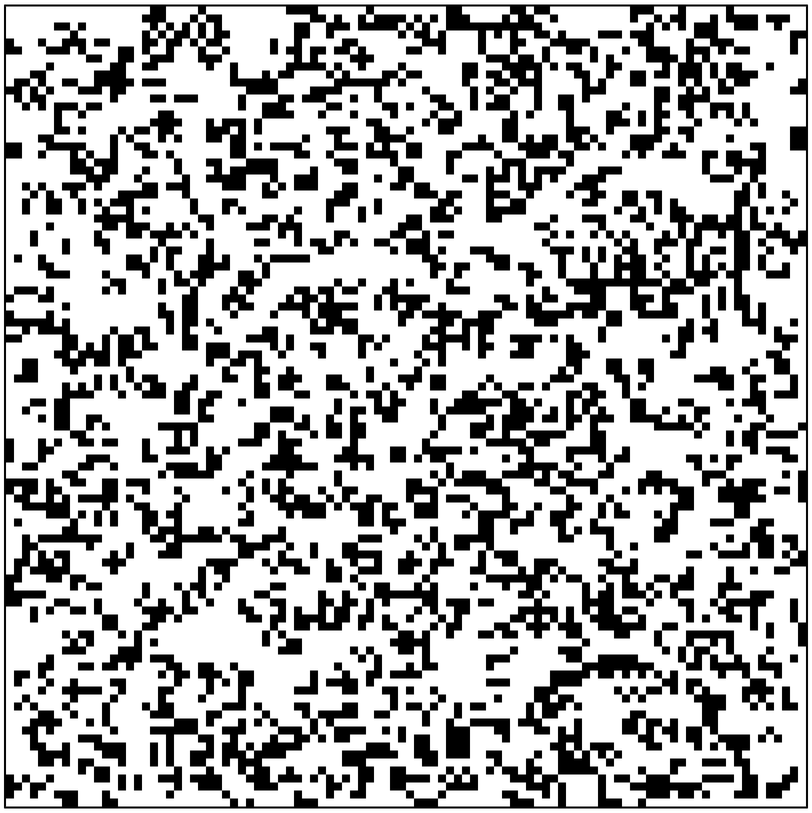
The final aspect to change is to actually use virus cells in the Lifesimulation class.this will be done below:



```
import java.util.Random;
public class Lifesimulation {
    public static void main(String[] args) throws InterruptedException{
     Landscape scape = new Landscape(100,100);
        Random gen = new Random();
        double density = 0.3;
        double vdensity=0.301;
        for (int i = 0; i < scape.getRows(); i++) {
            for (int j = 0; j < scape.getCols(); j++ ) {
                double randy=gen.nextDouble();
                if(randy >= density && randy<=vdensity){
                    //this will give a probability of 0.1
                    //and will set the cell equal to a virus
                    scape.getCell(i,j).setVirus(true);

                }
                else if(gen.nextDouble() <= density){
                    scape.getCell( i, j ).setAlive(true);//The original code
                }
                else{
                    //if none of the conditions are met then it will run the code

                    scape.getCell(i,j).setAlive(false);

                }
            }
        }
```

This makes the density of virus cells equal to 0.01.

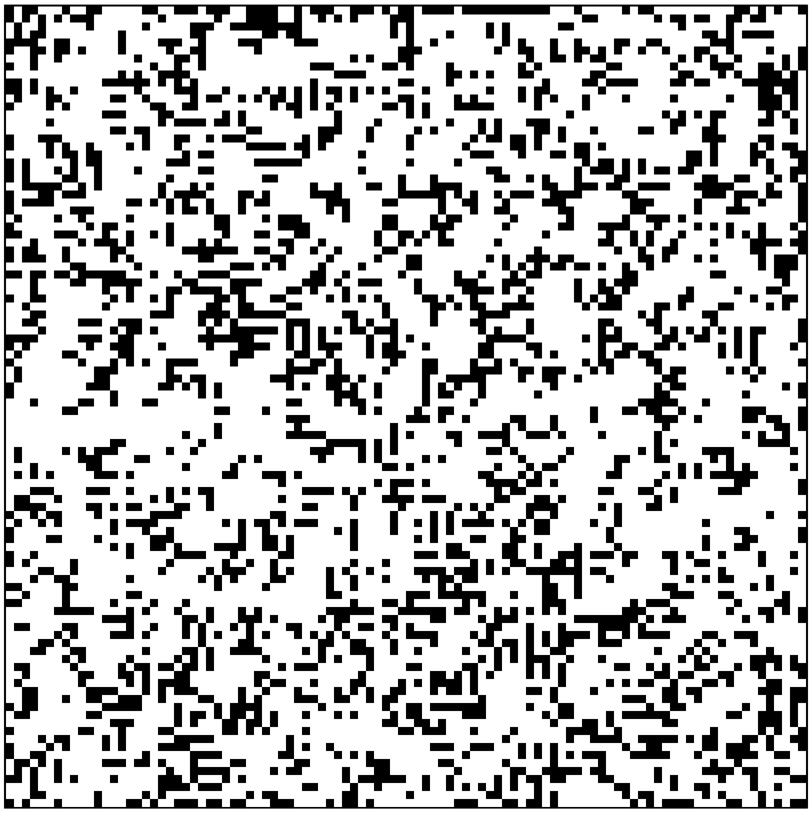
Final output:



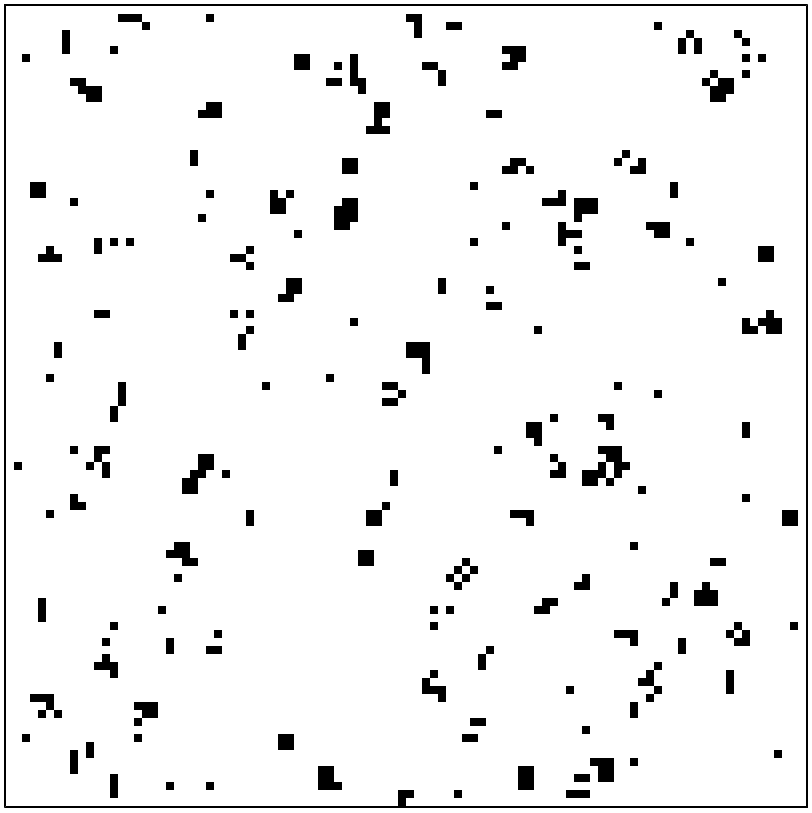
Checking different densities:

I wanted to test to see if different densities would affect the outcome of the density of the Cells on the board. This could go either way because there are rules for overpopulation, that kills the Cell, as well as under population killing the Cell. The goal of this would be to find what the optimal density is so that the most Cells end up in one of the oscillating or stationary patterns. I will test three densities and go from there based on which one is better, this will be a qualitative analysis of the board.

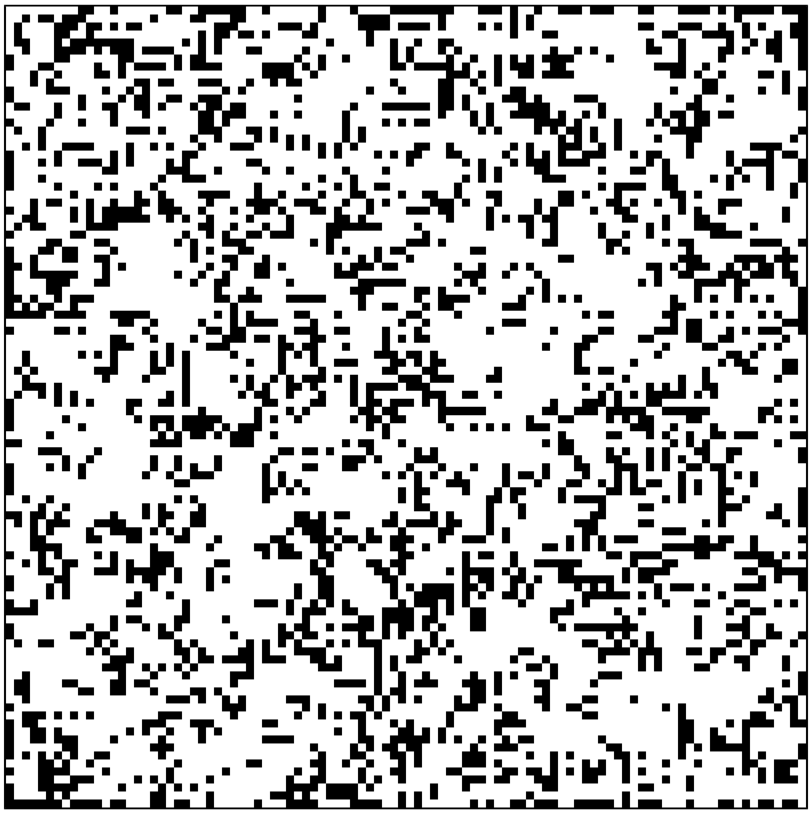
Density 0.3:



Density 0.5:



Density 0.9:



From the evidence above, it shows that the density of the map in terms of the Cells in it is actually not related to the density after some generations have passed.

Conclusion:

This project taught me many things about how to logically solve issues based on a given set of rules. It taught me various things about the quirks of the Java language, as well as a deeper understanding of how the Java compiler was actually created. Through this project I have also improved my ability to debug code, which is something I need to improve on.

Acknowledgements:

I would like to thank all the TAs,Dr.Skrien,Dr.Codabux, and Dr.Maxwell for helping me debug my code when my efforts fell short!