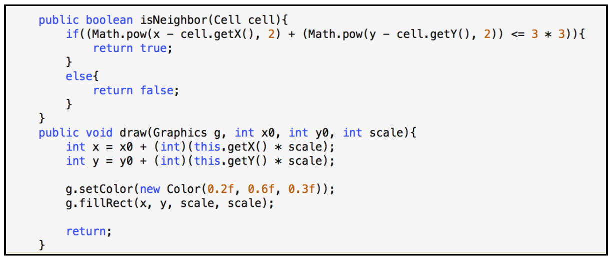
Project 4

In this weeks project, we continue to work with our LinkedLists, Landscape, and Cell classes. This week, we add a visual component to our simulations. In addition, we make use of a something new called an abstract class to use inheritance and abstract method. Also, I learned about the Java Graphics package. We copy over and update our old Cell, Landscape, and LinkedList classes and add new Simulation, ClumpingCell, PreferenceCell, and LifeCell classes. Overall, the goal of this project was to add more advance visual components to our simulations and continue our work with LinkedLists.

Task 1. In this task, we copied over our old Class.java file and made a few minor changes by making it an abstract class adding three abstract methods: isNeighbor, updateState, and draw. Also, we imported the java.awt.Graphics package.

Task 2. In this task, we created a new class called ClumpingCell that extends Cell and has a similar structure Cell class from Project 3. We made a an isNeighbor method that returns true if the argument cell is within some radius of the cell and a draw method that draws a colored rectangle representation of the cell. Also, we used an updateState that was very similar to the one from Project3 to mimic a clumping behavior. Here is a snippet of some of the code:

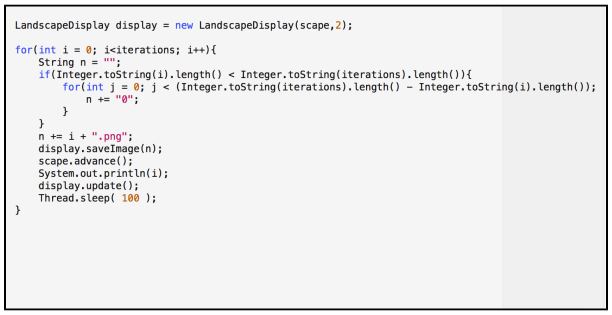


```java
    public boolean isNeighbor(Cell cell){
        if((Math.pow(x - cell.getX(), 2) + (Math.pow(y - cell.getY(), 2)) <= 3 * 3)){
            return true;
        }
        else{
            return false;
        }
    }
    public void draw(Graphics g, int x0, int y0, int scale){
        int x = x0 + (int)(this.getX() * scale);
        int y = y0 + (int)(this.getY() * scale);

        g.setColor(new Color(0.2f, 0.6f, 0.3f));
        g.fillRect(x, y, scale, scale);

        return;
    }
```

Task 3. In this task, we copied over our old Landscape.java class and updated it. We updated a few minor changes with width, height, getCols, getRows and making some values integers and some values doubles. Next, we updated our getNeighbors method which returns a list of neighbors around a given cell. Also, we update the main function to make sure it adds ClumpingCells rather than Cells to the landscape.

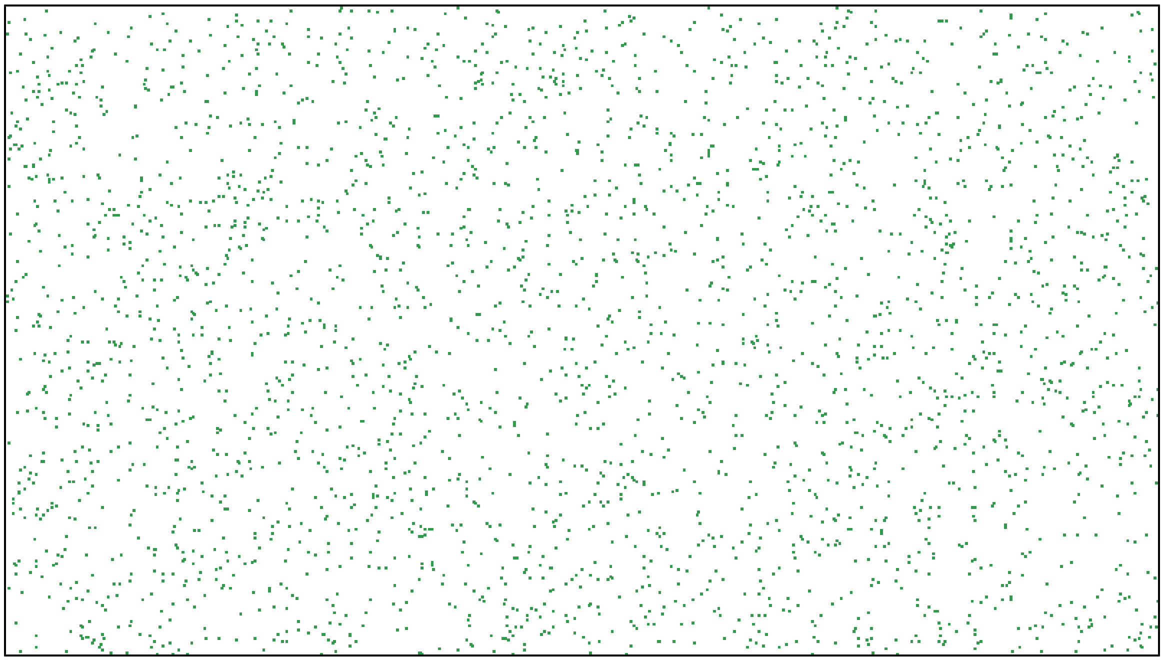
Task 4. For this task, we simply compiled and run the LandscapeDisplay.java class Dale gave to us and compare its behavior to our Landscape class.

Task 5. Next, for this task, we were asked to create a new Simulation.java class. This class follows the main function of LandscapeDisplay and uses its save method to save a series of .png images. In this task, I had to tackle to problem of making sure the files organized like 01.png, 02.png, etc RATHER THAN, 01.png, 11.png, etc. To do this, I received some help from CP Majgaard. Here is a snippet of our code:



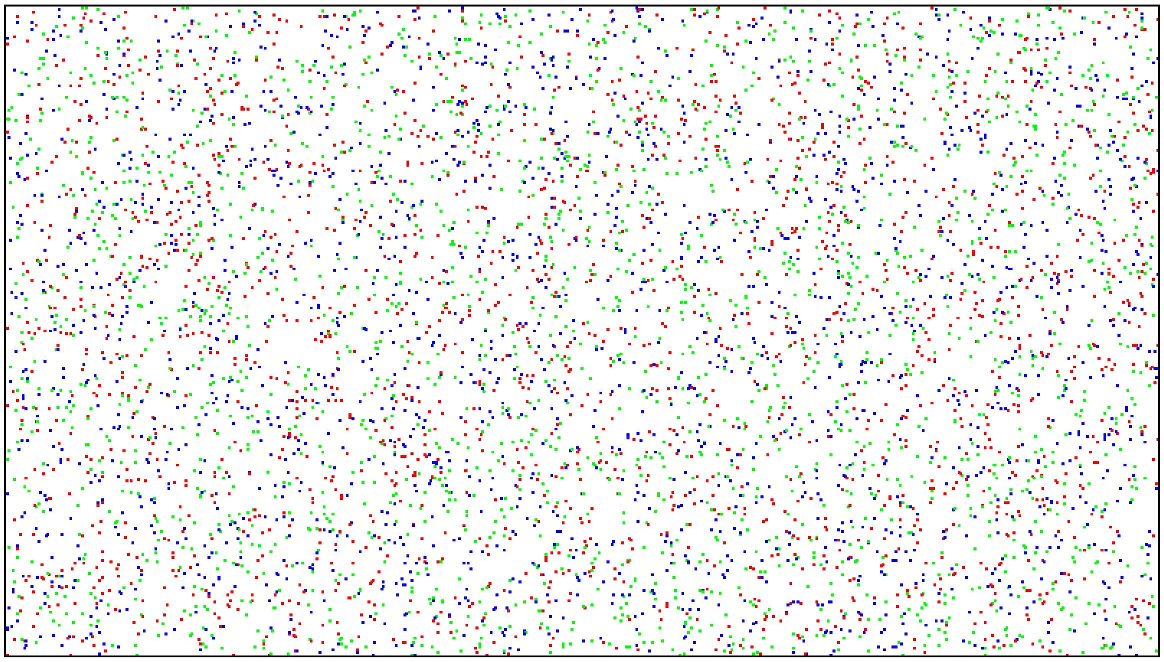
```java
LandscapeDisplay display = new LandscapeDisplay(scape,2);

for(int i = 0; i<iterations; i++){
    String n = "";
    if(Integer.toString(i).length() < Integer.toString(iterations).length()){
        for(int j = 0; j < (Integer.toString(iterations).length() - Integer.toString(i).length());
            n += "0";
        }
    }
    n += i + ".png";
    display.saveImage(n);
    scape.advance();
    System.out.println(i);
    display.update();
    Thread.sleep( 100 );
}
```

Next, once I had compiled an organized list of .png files I was able to create a .gif. So, when I run the Simulation.java class with ClumpingCells for 50 iterations, I get 50.png and use "convert -delay 60 *.png mysim.gif" and create a Gif movie. Here is the ClumpingCell GIF:



Task 6. Next, in task 6, we are asked to make a new child class of Cell called PreferenceCell. It implements the same simulation as CategorizedCell from Project 3, but using three different color categories.

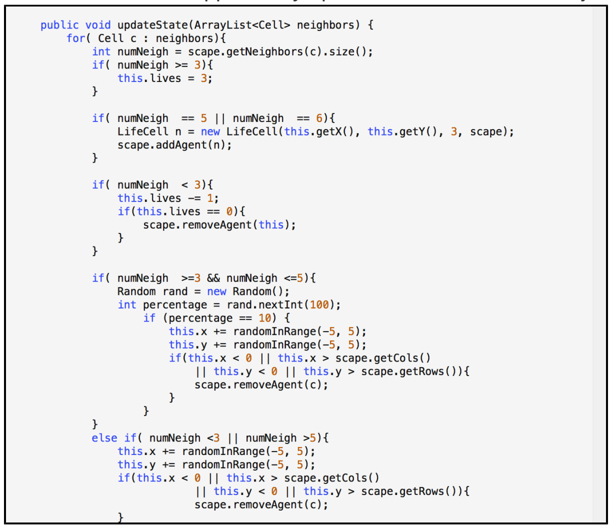
Task 7. In this task, we updated our Simulation class to control the type of Cell added to the Landscape and simulated from the command line. We know have two different possible child cells of Cell -- either ClumpingCell or PreferenceCell. I ran 50 iterations of PreferenceCells and then made the .pngs into a GIF. Here is wha the Preference GIF looked like:



Task 8. This task we created a third child class of Cell called LifeCell. It had to implement the following rules:

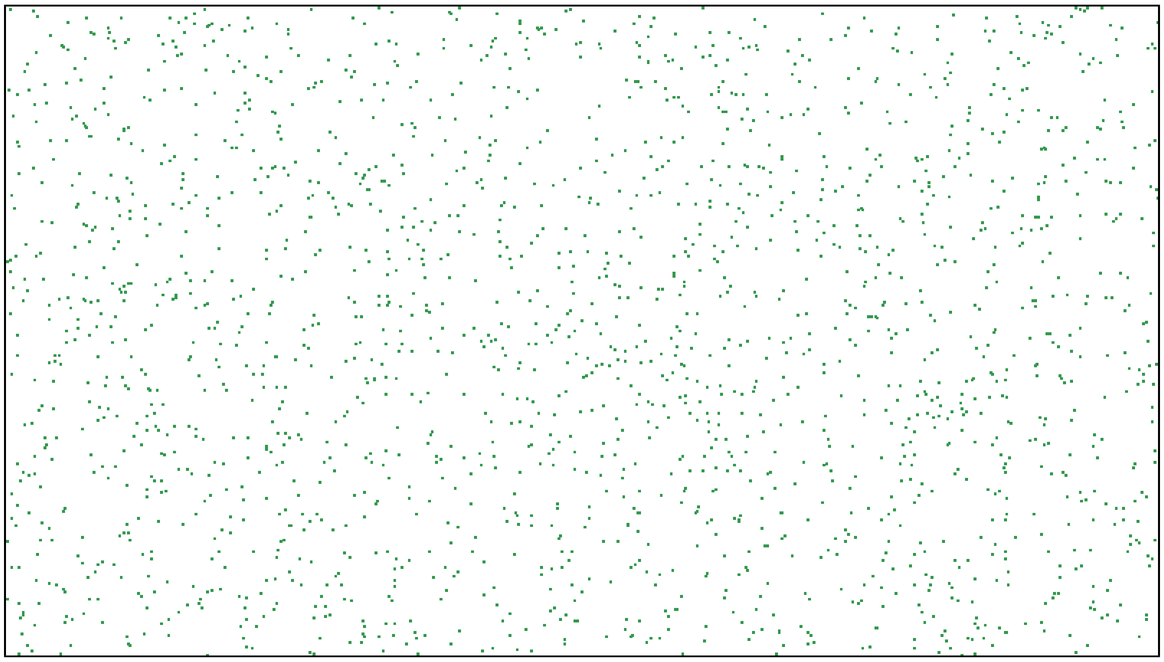
- If a LifeCell has at least 3 neighbors, it is alive and refreshed, which means it has exactly three lives.
- If a LifeCell has 5 or 6 neighbors, it can create a new LifeCell nearby, which should be added to the list of agents.
- If the LifeCell has fewer than 3 neighbors, then it loses one of its lives. A LifeCell with zero lives should be removed from the list of agents.
- A LifeCell that has between 3, 4, or 5 neighbors has a 1% chance of moving each round. A LifeCell with fewer than 3 or more than 5 will always move.To do this, I created a bunch of conditionals! Then, we had to add a remove method to our LinkedList class and

then subsequently a removeAgent method to the our Landscape class. This can be used in the rules and to remove cells that go out of bounds. Here is a snippet of my updateState code in LifeCell.java:

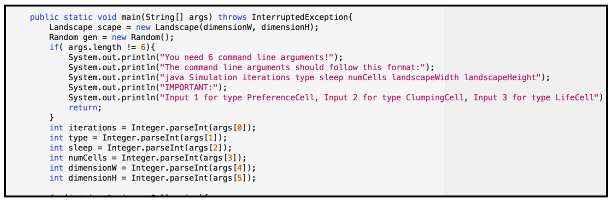


```
public void updateState(ArrayList<Cell> neighbors) {
    for( Cell c : neighbors){
        int numNeigh = scape.getNeighbors(c).size();
        if( numNeigh >= 3){
            this.lives = 3;
        }

        if( numNeigh  == 5 || numNeigh  == 6){
            LifeCell n = new LifeCell(this.getX(), this.getY(), 3, scape);
            scape.addAgent(n);
        }

        if( numNeigh  < 3){
            this.lives -= 1;
            if(this.lives == 0){
                scape.removeAgent(this);
            }
        }

        if( numNeigh  >=3 && numNeigh <=5){
            Random rand = new Random();
            int percentage = rand.nextInt(100);
                if (percentage == 10) {
                    this.x += randomInRange(-5, 5);
                    this.y += randomInRange(-5, 5);
                    if(this.x < 0 || this.x > scape.getCols()
                        || this.y < 0 || this.y > scape.getRows()){
                        scape.removeAgent(c);
                    }
                }
        }
        else if( numNeigh <3 || numNeigh >5){
            this.x += randomInRange(-5, 5);
            this.y += randomInRange(-5, 5);
            if(this.x < 0 || this.x > scape.getCols()
                        || this.y < 0 || this.y > scape.getRows()){
                        scape.removeAgent(c);
            }
```

Task 9. Finally, we updated our Simulation to have a third option of LifeCell filled Landscape. Then, I ran 10 iterations and used the .pngs to make a GIF. The LifeCell GIF looked like this:

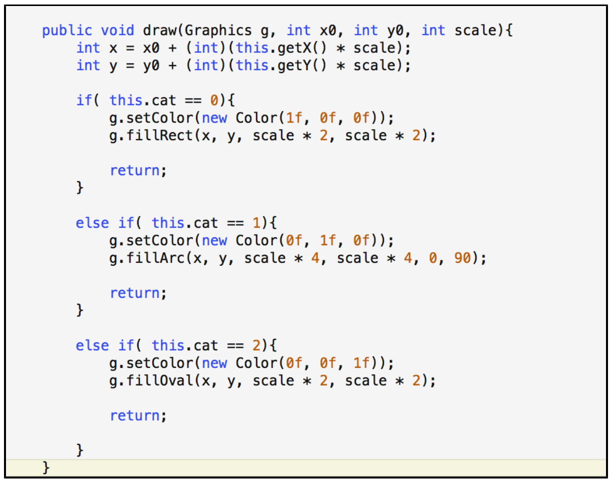


Extension 1. For the first extension, I made a lot of command line inputs for my Simulation class. I now have 6 command line arguments. They are for the number of iterations, the type of cell(1,2,or3), the sleep time, the number of cells in the landscape, the width of the landscape, and the height of the landscape. Here is the code:



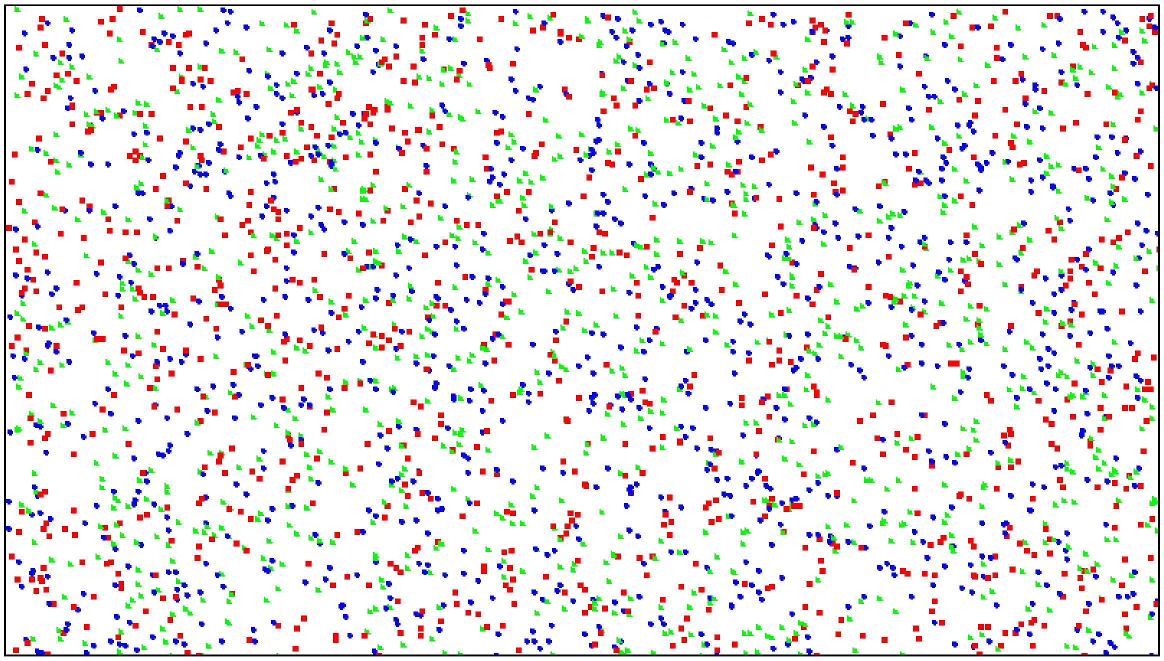
```
public static void main(String[] args) throws InterruptedException{
    Landscape scape = new Landscape(dimensionW, dimensionH);
    Random gen = new Random();
    if( args.length != 6){
        System.out.println("You need 6 command line arguments!");
        System.out.println("The command line arguments should follow this format:");
        System.out.println("java Simulation iterations type sleep numCells landscapeWidth landscapeHeight");
        System.out.println("IMPORTANT:");
        System.out.println("Input 1 for type PreferenceCell, Input 2 for type ClumpingCell, Input 3 for type LifeCell")
        return;
    }
    int iterations = Integer.parseInt(args[0]);
    int type = Integer.parseInt(args[1]);
    int sleep = Integer.parseInt(args[2]);
    int numCells = Integer.parseInt(args[3]);
    int dimensionW = Integer.parseInt(args[4]);
    int dimensionH = Integer.parseInt(args[5]);
```

Extension 2. For this extension, I decided to create a more interesting visualization of my Cells. To do this, I simply updated the three different categories in the PreferenceCell class. I used Graphic documentation to find out how to draw circles, squares and other polygons. I also made the polygons larger. So, I now have blue squares, red circles, and green triangle arcs AND they are twice as big. Here is a snippet of my code:



```
public void draw(Graphics g, int x0, int y0, int scale){
    int x = x0 + (int)(this.getX() * scale);
    int y = y0 + (int)(this.getY() * scale);

    if( this.cat == 0){
        g.setColor(new Color(1f, 0f, 0f));
        g.fillRect(x, y, scale * 2, scale * 2);

        return;
    }

    else if( this.cat == 1){
        g.setColor(new Color(0f, 1f, 0f));
        g.fillArc(x, y, scale * 4, scale * 4, 0, 90);

        return;
    }

    else if( this.cat == 2){
        g.setColor(new Color(0f, 0f, 1f));
        g.fillOval(x, y, scale * 2, scale * 2);

        return;

    }
}
```

Here is the GIF: Also, they still clump!



What I learned. Overall, the goal of this project was to update some of our old code to make more advanced visual representations. Through this project, I gained a better understanding of abstract classes and inheritance and I became more experienced with LinkedLists, Landscapes and Cells. Also, I learned about the Java Graphics package. In the end, this was an interesting project that made use of our old projects and incorporated a visual component.

Who helped me. I received help from Professor Maxwell and CP Majgaard. Also, I worked alongside fellow CS 231 classmate Steven Parrot and Jay Moore.