Project 2: Conway's Game of Life

Abstract
Tasks
  Task 1 - Cell.java
  Task 2 - Landscape.java
  Task 3 - LifeSimulation.java
Results of Simulation
  Required Picture 1 and 2
  Required Animation
Extensions
  Extension 1 - Support for Command Line Arguments
  Extension 3 & 4 - Age feature for Cells and Better Visualization
  Extension 5 - Interactive Mode (Buttons to Control the Simulation)
Learning Outcomes
Acknowledgment

Abstract

In this week's project, we implemented the famous Conway's Game of Life, a specific version of cellular automata. I used 2D array to store the grid data that represents the landscape of the simulation where each element in it represents a Cell, with some use of ArrayList and HashMap for convenience. The simulation, models the live and death status for all the cells within the landscape, where all cells' status are updated constantly. Besides, I also learnt about Java Graphics, and used Swing to make a faded-colored interactive visualization window of the simulation.

Tasks

Task 1 - Cell.java

This class is to represent the Cell object in the Landscape, there are some critical methods for this class.

1. `updateState()` This method updates whether this cell is alive or dead in the next frame. The neighboring cells will be passed into this method in the form of `ArrayList<Cell>` and it will check the total number of live cells and

update the cell's state accordingly. Since I added the `age` field for extension, this method will also update the `age` if a live cell continues to be alive (so getting older).

```java
public void updateState(ArrayList<Cell> neighbors) {
    int count = 0;
    for (Cell neighbor : neighbors) {
        count += neighbor.getAlive() ? 1 : 0;
    }
    if (this.getAlive()) {
        // When the cell is currently alive
        if (count == 3 || count == 2) {
            // Continue living, increase age
            this.increaseAge();
        } else {
            // Turn dead, back to age 0
            this.setAlive(false);
            this.setAge(0);
        }
    } else {
        // When the cell is currently dead
        if (count == 3) {
            // Turn alive, initialize age to 1
            this.setAlive(true);
            this.setAge(1);
        }
    }
}
```

2. `toString()` This method does a tiny job of returning a text message that represents the cell's state. It is still important because rather than using text, I chose emoji to represent alive 🌲 or death 🍂, so it can make my text-based visualization very easier (see below).

```java
public String toString() {
    return alive ? "\uD83C\uDF32" : "\uD83C\uDF42";
}
```

3. `draw()` This method does the most critical job for my extension of making a better visualization. It will determine the color of the cell depending on its age, and the color ranges from light green to dark green, all of which are `Color` object and are accessed by a pre-defined `HashMap<Integer, Color> palette` from the `Landscape` class. This method will also determine whether the rectangle (representing the cell in the visualization) needs to be raised to 3D rectangle.

```java
public void draw(Graphics g, int x, int y, int scale, HashMap<Integer, Color> palette) {
    if (this.getAlive()) {
        Color cellColor = palette.get(this.age);
        // If age is greater than the maximum option, use the maximum one
        g.setColor(cellColor != null ? cellColor : palette.get(10));
    } else {
        // If cell is dead, use plain gray
        g.setColor(new Color( r: 206,  g: 212,  b: 218));
    }

    if (this.getAlive()) {
        // Use 3DRect for alive cells
        g.fill3DRect(x, y, scale, scale,  raised: true);
    } else {
        // Use normal Rect for dead cells
        g.fillRect(x, y, scale, scale);
    }
}
```

Task 2 - Landscape.java

This class represents one Landscape class, which contains the `grid` of all cell, in the form of `Cell[][]` . It also contains several critical methods.

1. `Landscape()` This is the constructor for the class, which creates new `Cell` in the memory, and initialize all cells by a 2 dimensional iteration.

```java
public Landscape(int rows, int cols) {
    this.rows = rows;
    this.cols = cols;
    this.grid = new Cell[rows][cols];
    this.palette = getPalette();

    // iterates through and sets all cell status dead
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            grid[i][j] = new Cell();
        }
    }
}
```

2. `getNeighbors()` This method is the most significant one. It returns all the neighboring cells of the given cell in the form of `ArrayList<Cell>`. I implemented this method by first checking if the given index of the `cell` is out of bound. If yes, it will throw a `IndexOutOfBoundsException`, warning the

user. Then, it will iterates through all the eight surrounding `cells` of the given `cell` and add them to the `ArrayList`. I didn't use nested conditions which will make the program unreadable; instead, I simply iterates through all of them, and ignore those ones that are out of bound (i.e. when the given `cell` is at the `grid`'s boundary) by a `try-catch` block that does nothing to the exception.

```java
public ArrayList<Cell> getNeighbors(int row, int col) {

    if (row < 0 || row >= this.rows || col < 0 || col >= this.cols) {
        // check if the given index is not in the range of this grid
        throw new IndexOutOfBoundsException("Index Out Of Bounds: given index " +
                "(" + rows + "," +cols + ") not in the grid");
    }

    ArrayList<Cell> neighbors = new ArrayList<>();
    for (int i = row - 1; i < row + 2; i++) {
        for (int j = col - 1; j < col + 2; j++) {
            try {
                // Ignore if the cell is out of bound
                if (i != row || j != col) {
                    // Avoid adding itself
                    neighbors.add(this.getCell(i, j));
                }
            } catch (IndexOutOfBoundsException ignored) { }
        }
    }
    return neighbors;
}
```

3. `advance() & getDuplicatedGrid()` These two methods are the combo for advancing the simulation into the next frame. It follows the guidance from the project description, duplicating another `grid`, updates the temporary `grid` based on the original one (since the in-place change on the original one will affect next `cell`), and assigns back to the original `grid` after all `cells` have gone through the `updateState()`.

```java
public void advance() {
    Cell[][] tempGrid = this.getDuplicatedGrid();
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            tempGrid[i][j].updateState(this.getNeighbors(i, j));
        }
    }
    this.grid = tempGrid;
}

/**
 * @return a duplicate of field grid of Cell[][]
 */
public Cell[][] getDuplicatedGrid() {
    Cell[][] copiedGrid = new Cell[this.rows][this.cols];
    for (int i = 0; i < this.rows; i++) {
        /*if (this.cols >= 0) {
            System.arraycopy(grid[i], 0, copiedGrid[i], 0, this.cols);
        }*/
        for (int j = 0; j < this.cols; j++) {
            copiedGrid[i][j] = new Cell(grid[i][j].getAlive());
            copiedGrid[i][j].setAge(grid[i][j].getAge());
        }
    }

    return copiedGrid;
}
```

4. `randomize()` This method randomize the `Landscape` and set the `density` of live `cells`.

```java
public void randomize(Random randomGenerator, double density) {
    this.reset();
    for (int i = 0; i < this.getRows(); i++) {
        for (int j = 0; j < this.getCols(); j++ ) {
            this.getCell( i, j ).setAlive(randomGenerator.nextDouble() <= density);
        }
    }
}
```

5. `getPalette()` This is a `static` method that generates a color palette in the form of `HashMap<Integer, Color>`. Its key represents the `age` range for the `cells`, and the value corresponds to the `Color` object that a `cell` at a particular `age` has. As you can see here, the color generally follows a green trend, as it starts from the lightest green and fades to the darkest

when the `age` gets older. In the extension of visualization (see below), you will see the how the color fades in the simulation.

```java
public static HashMap<Integer, Color> getPalette() {
    return new HashMap<>() {
        {
            put(1, new Color( r: 255,  g: 243,  b: 220));
            put(2, new Color( r: 180,  g: 225,  b: 192));
            put(3, new Color( r: 149,  g: 213,  b: 178));
            put(4, new Color( r: 116,  g: 198,  b: 157));
            put(5, new Color( r: 82,  g: 183,  b: 136));
            put(6, new Color( r: 64,  g: 145,  b: 108));
            put(7, new Color( r: 45,  g: 106,  b: 79));
            put(8, new Color( r: 27,  g: 67,  b: 50));
            put(9, new Color( r: 11,  g: 45,  b: 30));
            put(10, new Color( r: 8,  g: 28,  b: 21));
        }
    };
}
```

Task 3 - LifeSimulation.java

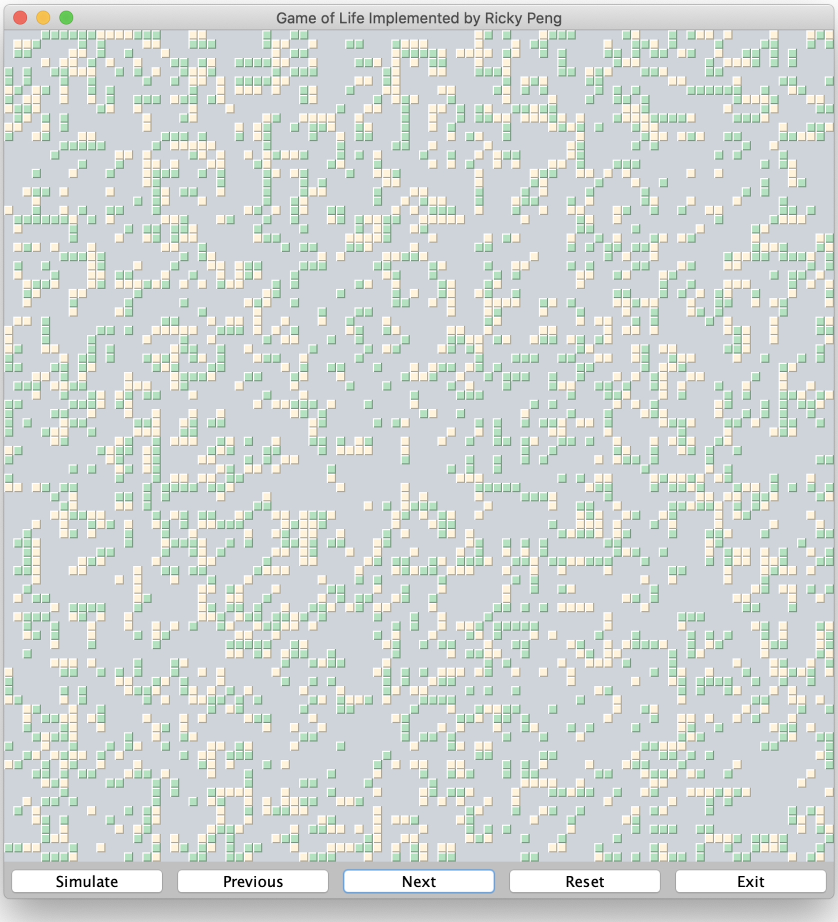
This is the class that represents a simulation of one type. I created three types of simulations here, `graphics`, `text-based` and `interactive`. Few methods need to be addressed here.

1. `simulate()` This method opens a window that presents the visualization of the game of life.
2. `simulateToText()` This method visualizes the simulation of the game of life by printing grid of emojis to the console. Will discuss below (see extension).
3. `simulateInteractive()` This method visualizes the simulation, and allows user to control the process by clicking the buttons. Will discuss below (see extension).

Results of Simulation

Required Picture 1 and 2

The following two pictures are the initial stage of one simulation and its second stage, the cells whose color are darker are the ones older, meaning they remain alive longer.



Required Animation

The attached animation includes not only basic simulation visualization, but also extensions for command line argument support, enhanced visualization and user control feature.

See the video in the folder.

Extensions

I implemented 3 suggested extensions as follow.

Extension 1 - Support for Command Line Arguments

I added support so that users can customize the simulations directly from the terminal, using command line arguments.

1. When you directly enter `java LifeSimulation`, you will receive a notice that the simulation `mode` is required, plus a message to invite you to add `--help`

for usage document.

```
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation
[WARNING] Please specify display mode
[WARNING] Add "--help" in command line argument to view usage
```

2. When you then enter `--help`, you will see the usage document as follow. The default value and required input formats are also listed.

```
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation --help
Usage: --mode <String> -D <double> -RC <int int> -I <int>

       --mode <String>  (NO_DEFAULT) : select between "graphics", "interactive" and "text",
                                     : for "text", add "> fileName.txt" at the end to store the result
       -d     <double>  (0.3)        : density of cells in the grid
       -rc    <int int> (100, 100)   : rows and columns of the landscape
       -i     <int>     (20)         : number of iterations
```

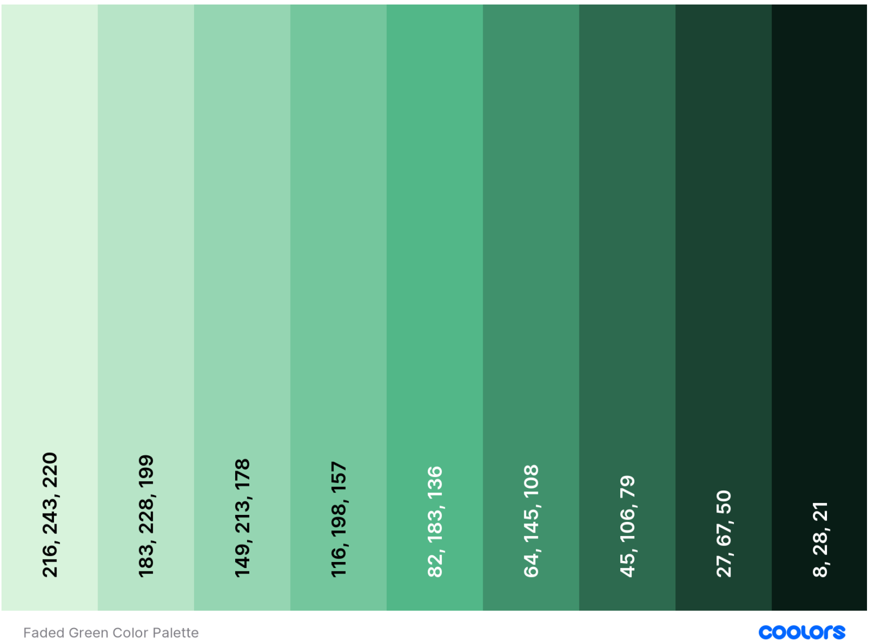
3. After you are familiar with the command line arguments, you will then customize the simulations based on `mode`, `density`, `rows`, `columns`, and number of `iterations`. If no customized values of these are entered, the default value will be passed into the simulation.

```
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation --mode graphics -d 0.4 -rc 60 60 -i 10
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation --mode interactive -d 0.3 -rc 120 120 -i 30
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation --mode graphics -d 0.3
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation --mode interactive -rc 90 90 -i 5
➜  Project_2_Game_of_Life git:(master) ✗ java LifeSimulation --mode text
```

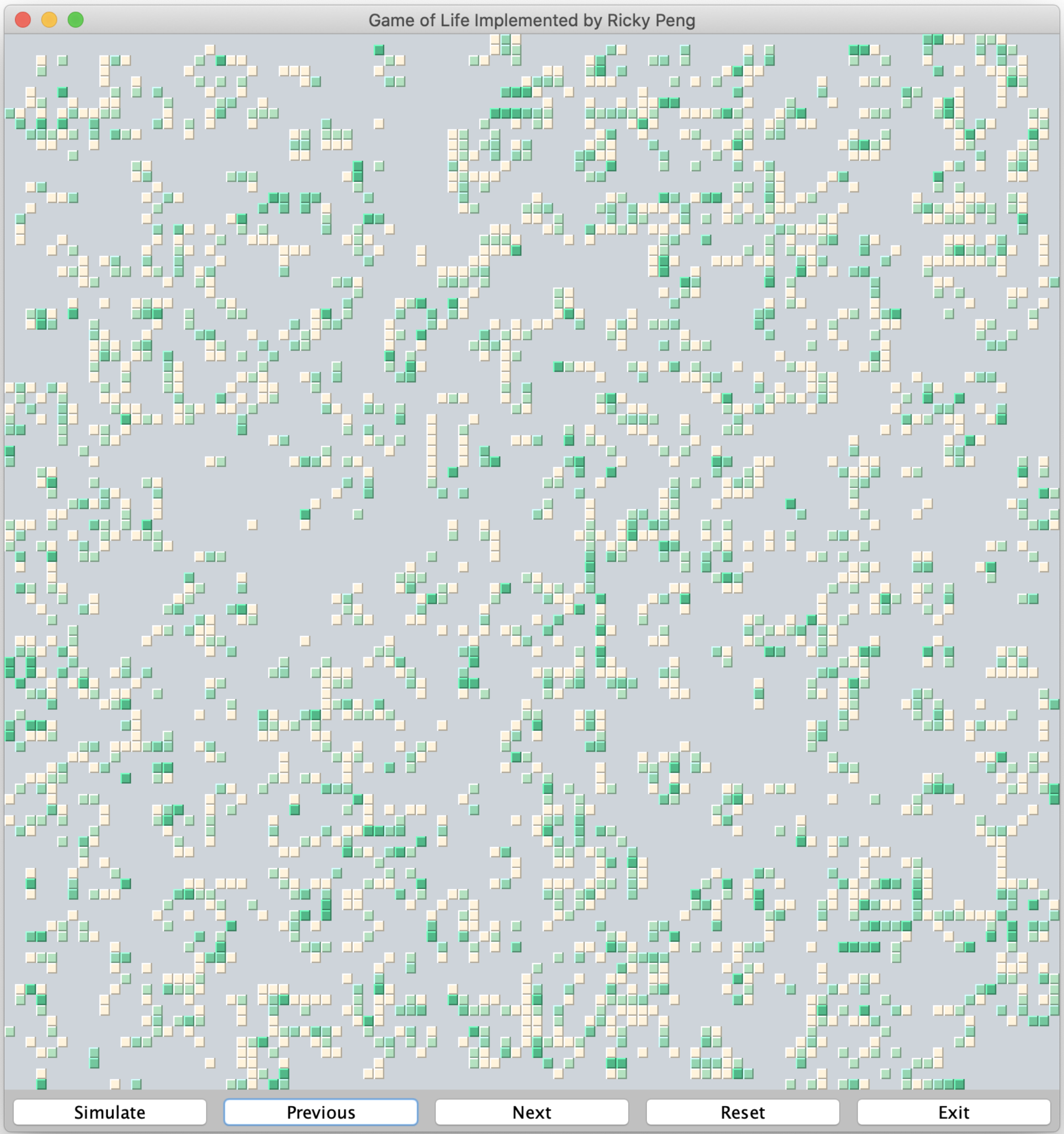
Extension 3 & 4 - Age feature for Cells and Better Visualization

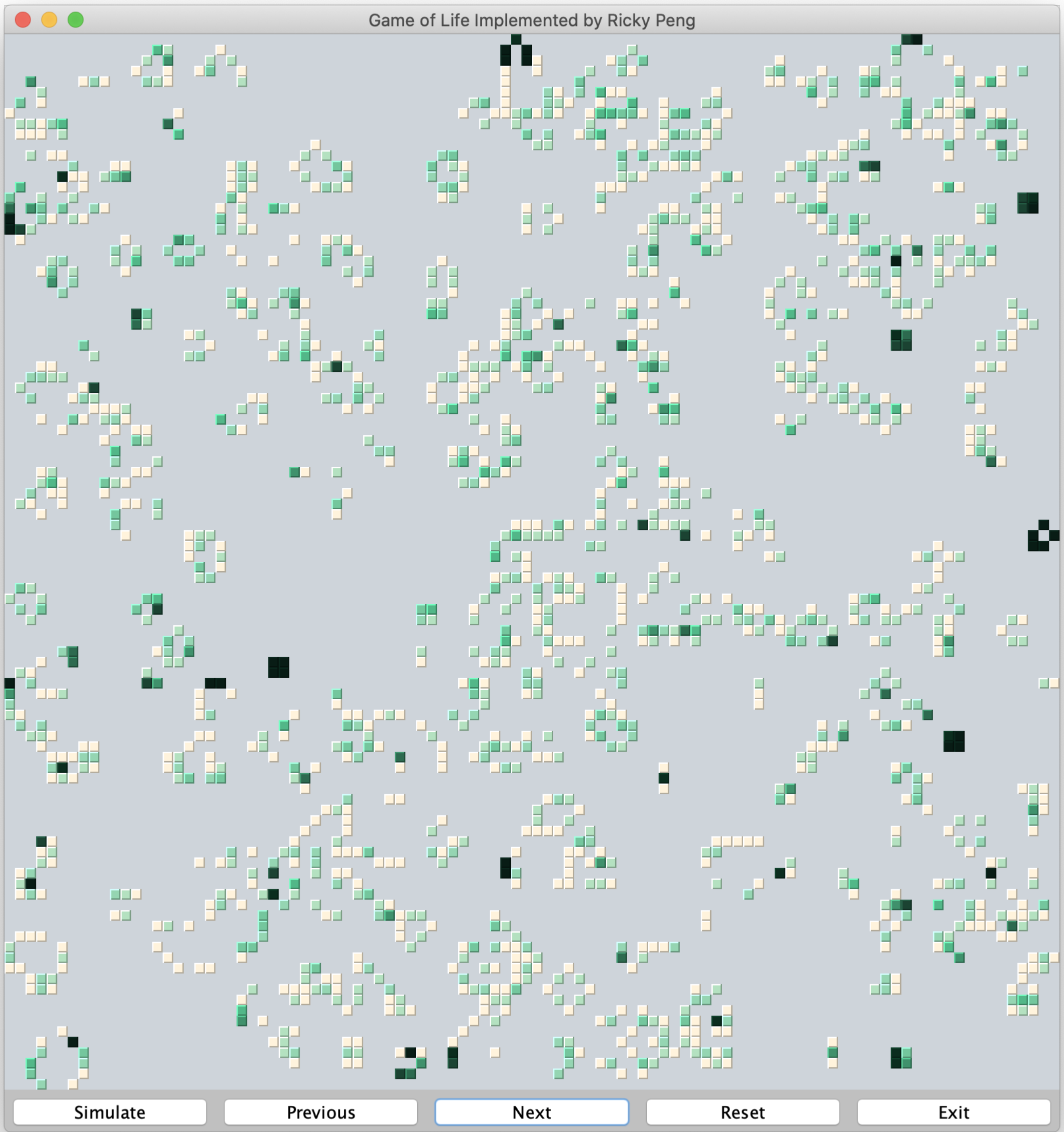
In this extension, I added one feature to the `Cell` object by giving each of them a `age` attribute. This represents the number of frames a live cell has experienced, i.e. the number of times it has survived. Its `age` will be increased by 1 when `updateState()` is called on such `Cell`. This is similar to the idea of giving different roles to cells, however, the role mentioned in the Project's description seems discrete, such as mother/father/child. In my extension, the role here becomes continuous, representing the `age` of one `Cell`.

For the visualization, I used a Green color as my main color element. The rule here is, the younger the `Cell` is, the lighter green color it has, and the older it is, the darker green color it has. I obtained this green color palette when I played with http://coolors.com.



Thus, after implementing this rule, I used this `HashMap<Integer, Color>` color palette (see `draw()` method above) to store the color, and return the `Color` according to the `Cell` 's `age`. For example, at the fourth year (frame), there are some cells with dark green meaning they are older than those of lighter green. In the second picture of year 14, there are some even darker, meaning those cells have remained alive for a long time.


Game of Life Implemented by Ricky Peng
Simulate
Previous
Next
Reset
Exit



I also used `fill3DRect()` for the live cells and `fillRect()` for the dead cells to distinguish between them.

```
if (this.getAlive()) {
    // Use 3DRect for alive cells
    g.fill3DRect(x, y, scale, scale, raised: true);
} else {
    // Use normal Rect for dead cells
    g.fillRect(x, y, scale, scale);
}
```

Extension 5 - Interactive Mode (Buttons to Control the Simulation)

In this extension, I enabled the simulation to be controlled by users. User can simulate the entire process by clicking the Simulate button, move to next frame or go back to the previous frame by clicking the Next or the Previous button, and reset the simulation to the first frame by clicking the Reset button. They can of course exit the program by clicking the Exit button.

To do this, I created another class called `LandscapeInteractiveDisplay`. It remains the same compared to `LandscapeDisplay`, except for the operations related to the following buttons.

```
this.initializeButtons();

this.canvas.setLayout(null);

this.canvas.add(simulateButton);
this.canvas.add(nextButton);
this.canvas.add(backButton);
this.canvas.add(resetButton);
this.canvas.add(exitButton);
```

For all five buttons, I used the `JButton` object. When I initiated each of them, I set its name, its bounds, its background color (consistent with the cell's green color), and add a ActionListener that listens to any event (clicking) happening to that button. I will use the Simulate and Previous buttons as examples here.

```java
this.simulateButton = new JButton( text: "Simulate") {
    {
        setBounds(startX,startY,buttonWidth,buttonHeight);
        setBackground(new Color( r: 180,  g: 225,  b: 192));
        addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                LifeSimulation.setInteractive(1);
            }
        });
    }
};

this.backButton = new JButton( text: "Previous") {
    {
        setBounds( x: startX + buttonWidth, startY, buttonWidth, buttonHeight);
        setBackground(new Color( r: 180,  g: 225,  b: 192));
        addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                LifeSimulation.setInteractive(2);
            }
        });
    }
};
```

The `addActionListener()` is passed into a anonymous `ActionListener` class that overrides the `actionPerformed()` method. Thus, when each of these buttons is clicked, there will be a response. Here in order to let the `LifeSimulation` instance know that I clicked the button so it would need to perform different actions respectively, I added a static attribute called `Interactive`. It is a integer that stores the number of button the users click. For example, when the user click Simulate button, the `interactive` value will be set to 1, and the `LifeSimulation` instance will hear it and perform the following procedures that presents the entire simulation.

```java
int status = getInteractive();
System.out.print("");
if (status == 1) {
    for (int i = arrayPointer; i < numIterations; i++) {
        displayInteractive.scape = landscapes.get(i).getCopiedLandscape();
        displayInteractive.repaint();
        Thread.sleep( millis: 500);
    }
    arrayPointer = numIterations - 1;
    setInteractive(101);
```

In order to have the features of going back to the previous frame or moving to the next frame, I used an `ArrayList<Landscape>` here to store all the upcoming `landscape` before everything. So when the user clicks Next or Previous, the program will just access the `landscape` at a particular index from this `ArrayList`. After all the `landscape` are simulated beforehand, the visualization window will go back to the first frame (see `landscapes.get(1).getCopiedLandscape()`) and presents it on the window. There is a small trick here related to reference and memory. I was first using `landscapes.add(landscape)` when I wanted to add to the `ArrayList`, however, I found in this way all landscapes will be the same, because they points to the same location in the memory. To solve this, I wrote this `getCopiedLandscape()` method that creates a new `Landscape` in the memory.

```java
/**
 * Allow users to control the simulation process
 * @throws InterruptedException from Thread.sleep()
 */
public void simulateInteractive() throws InterruptedException {

    LandscapeInteractiveDisplay displayInteractive = new LandscapeInteractiveDisplay(landscape, GRID_SCALE);

    ArrayList<Landscape> landscapes = new ArrayList<>();

    for (int i = 0; i < numIterations; i++) {
        landscapes.add(landscape.getCopiedLandscape());
        landscape.advance();
    }

    displayInteractive.scape = landscapes.get(1).getCopiedLandscape();
    displayInteractive.repaint();

    int arrayPointer = 1;
```

```
public Landscape getCopiedLandscape() {
    Landscape out = new Landscape(this.getRows(), this.getCols());
    out.grid = this.getDuplicatedGrid();
    out.palette = getPalette();

    return out;
}
```

The code will perform different actions according to the which button is clicked. For example, when `getInteractive()` returns 2, it means the Back button is clicked. Inside this endless `while` loop that constantly listens to the actions, the code will go to the conditionals where `status == 2`. In this case, the program will first check if the current frame is already the first frame, which means there's no previous frame to go back to. If not, it will get the previous frame by accessing the `ArrayList` at index `(arrayPointer-1)`, where `arrayPointer` is the integer of the index of the current frame. This value is also updated throughout the conditionals, based on the circumstances of each action (for example, in simulation button action, it will eventually be updated to `numIterations-1` because that's the index of the last frame, which is shown on the window when a complete simulation is done).

```java
int arrayPointer = 1;

while (getInteractive() != -1) {
    int status = getInteractive();
    System.out.print("");
    if (status == 1) {
        // When Simulate button is clicked
        for (int i = arrayPointer; i < numIterations; i++) {
            displayInteractive.scape = landscapes.get(i).getCopiedLandscape();
            displayInteractive.repaint();
            Thread.sleep( millis: 500);
        }
        arrayPointer = numIterations - 1;
        setInteractive(101);
    } else if (status == 2) {
        // When Back button is clicked
        try {
            if (arrayPointer == 1) {
                throw new IndexOutOfBoundsException();
            } else {
                displayInteractive.scape = landscapes.get(arrayPointer - 1).getCopiedLandscape();
                arrayPointer--;
                displayInteractive.repaint();
            }
        } catch (IndexOutOfBoundsException ignored) {}
        setInteractive(103);
    } else if (status == 3) {
        // When Next button is clicked
        try {
            displayInteractive.scape = landscapes.get(arrayPointer + 1).getCopiedLandscape();
            arrayPointer++;
            displayInteractive.repaint();
        } catch (IndexOutOfBoundsException ignored) {}
        setInteractive(102);
    } else if (status == 4) {
        // When Reset button is clicked
        displayInteractive.scape = landscapes.get(1).getCopiedLandscape();
        arrayPointer = 1;
        displayInteractive.repaint();
        setInteractive(104);
    }
}
```

See the attached video that demonstrates this feature.

Learning Outcomes

1. I learnt about Java Graphics by using Java Swing.
2. I learnt about Java Color class as well and Font class (although not used here).
3. I explored command line arguments, which I seldom did.

4. I got to know about ActionListener in Swing, somewhat different from what I previously knew in JavaFX.
5. I enhanced my understanding of memory and reference.
6. 2D array which I was quite familiar with, is also reviewed.

Acknowledgment

I didn't ask anyone during this project. I was going to ask Professor Al Madi about the ActionListener, but the Office Hour was too early for me. I figured them out later by debugging. Nonetheless, I used Java Swing documents as reference.