The program GameOfLife allows a user to play a game called “Life”, invented by the mathematician John Conway. The program simulates and animates the birth, life and death of a population of individuals inhabiting a square grid. The program begins by presenting the user with an empty grid. The user has the following options: Press the “Randomize” button to generate an initial population that is randomly distributed over the cells of the grid; Click on an individual cell to create or destroy life in the designated cell; Press the “Next” button to process one generation of the simulation; Press the “Start” button to commence a simulation that runs for multiple generations, until the user presses the “Stop” button; Press the “Clear” button to kill all life in the grid; Press the “Exit” button to quit. During the simulation, the program simulates the birth and death of objects in the grid, according to the following rules:

- **Birth**: If a dead cell is surrounded by exactly three living cells, it becomes alive in the next generation. Otherwise, it remains dead in the next generation.
- **Life**: If living cell is surrounded by either two or three other living cells, it is happy, and therefore remains alive in the next generation.
- **Death**: If a living cell is surrounded by less than two, or more than three living cells, it dies of loneliness or overcrowding, and is therefore dead in the next generation.

The program uses two data structures to keep track of the simulation. A variable called “world” is a two-dimensional array of GOval objects. The entry world[r][c] is a GOval representing the object that inhabits the cell at row r and column c. A variable called “alive” is a two-dimensional of boolean values. The entry alive[r][c] indicates whether the object inhabiting the cell at row r and column c is dead or alive.

A partially complete version of this program can be found in the NetBeans project folder GameOfLife in the files for this assignment. The program compiles and executes, but does not actually carry out all the user-initiated commands, or the simulation itself. Your job is to complete the program. To this end, you must implement each of the methods described below:

```java
protected void setUpWorld() {
    Initialize the world and alive variables to be two-dimensional arrays of (respectively) GOval and boolean values. Each array has WORLD_SIZE rows and WORLD_SIZE columns. Use the (already defined) GOval newCell(int r, int c) method to generate an object that inhabits the cell at row r and column c in the world array. Each entry in the alive array should be initialized to false.
}
```

```java
protected void randomize() {
    Fill the alive array with random boolean values. Each entry in this array should be true with probability (1/SPARSITY) and false otherwise. Call updateColors to update the colors of the GOval objects in the world array, after updating alive with random values.
}
```

```java
protected void nextGeneration() {
    First call updateLife(), which will update the boolean values in the alive array, according the rules described above. Then call updateColors(), which will update the colors of the GOval objects in the world array.
}
```

```java
protected void updateLife() {
    Update the boolean values in the alive array, according the rules described above. Make use of the numNeighbors method to determine the number of neighboring cells with living occupants. Note: You should not change any values in the alive array until you have computed all the new values for this array. Therefore you should construct and store a temporary array to...
}
```
hold the new values as you are computing them. At the end of the method, you should assign the temporary array to the `alive` variable.

```java
int numNeighbors(int r, int c) Examine the eight cells surrounding the cell at row r and column c. Determine how many of them are occupied by living objects, and return that number. Locate the surrounding cells by considering all combinations of (dr = -1, 0,1) and (dc = -1, 0,1) except for (dr = 0) and (dc = 0). For each combination of dr and dc, the coordinates of the neighboring cell are given by (r+n+dr)%n and (c+n+dc)%n, where n is WORLD_SIZE (the width and height of the grid). These expressions arrange that the sides of the grid wraps around so that the top most row borders the bottom most row, and the left most column borders the right most column.
```

```java
protected void updateColors() Set the color of each GOval stored in the world array. If `alive[r][c]` is false, then the GOval at `world[r][c]` should be black. Otherwise, if `numNeighbors(r,c)` is 0, 1, 2 or 3, the GOval at `world[r][c]` should be COLOR0, COLOR1, COLOR2 or COLOR3, respectively. If `numNeighbors(r,c)` is 4 or higher, then it should be COLOR4.
```

```java
protected void clear() Set all the entries in the `alive` array to false. Call `updateColors` at the end of this method.
```

I recommend that you implement these methods in the following order: First implement `setUpWorld`, `randomize`, `clear` and a simplified version of `updateColors` (which makes objects black if they are dead and white if they are alive). At this point, you can generate and clear random arrangements and see the results in a graphics window. Then go ahead and implement `nextGeneration`, `updateLife` and `numNeighbors`. At this point you can run the simulation. Then go back and implement the complete version of `updateColors`.

Once your program is running, you may wish to modify the constants `COLOR1`, ..., `COLOR4` that define the color scheme used in the animation of the Game of Life. You may find colors to your liking in the documentation of the class `java.awt.Color` which you can find by following the “Java Platform Documentation” link at the bottom of the class web page.

Check out the Wikipedia entry on “Conway’s Game of Life”, where you will find a variety of structures you can build in the grid world and simulate in your program.