**GREEN*vs*RED**

The main method of our Main class is our entry point of our java program. We have used a Scanner class for getting user input. The user should enter the rows and columns of the grid, the grid itself (as one-dimensional array of strings at first), the number of generations and the coordinates of the target cell.

Then we create an instance of Game class. We are calling the createGenerationZero method from the game object, which creates and validates (using the ValidateInput class) the one-dimensional array, entered by the user.

While validating, we are checking whether the values of the rows and the columns are in the stated range. We check whether the coordinates of the target cell are in range, as well. And finally, we check whether the elements of the grid are with specific length (= columns of the grid) and whether the elements contain only 1s and 0s. If all of these conditions are met, we can continue with the actual game. But before that, the one-dimensional array is transformed to two-dimensional array, in order to work with a matrix (or grid).

From the main method we are calling the start method of the game object. We are creating two objects of type Rule, one is instance of the RedCellRule class and the other one is instance of the GreenCellRule class, because the last two classes are extending the abstract class Rule. Then we create a few more variables and an instance of the NewElement Class, in which we would store the element after the rules are applied and if it has changed its value to 1 (turned to green) the boolean variable isGreen would change to true.

We continue by checking whether the target cell’s value is 1 and if it is, we increase the value of our counter (timesBeenGreen) with 1. Then by using a while loop, we keep track of the passed generations. In each generation we create and initialize a buffer array, called nextGrid, in which we would store the new values of the elements. The buffer array has the same size as the grid. Then we go through each element of the grid and check its value. If it 1 (the cell is green), then we apply the rules implemented in the applyRules method of the object greenCellRule. If it is 0 (the cell is red), we apply the rules implemented in the applyRules method of the object RedCellRule.

In the applyRules method we call the countGreenNeighbours method to see how many green neighbours has the specific cell. We do this by first defining the bounds of the grid (MIN\_X, MAX\_X, MIN\_Y, MAX\_Y). Using ternary operator, we define the range of the neighbours.

We go through each neighbour and make sure we include only the neighbours and not the cell itself. After we have the amount of green neighbours we see if the if-condition is met (the condition is different if the cell is green or red) and change the value of the cell for the next generation. The changed value is stored in a buffer array and we check if the target cell has changed its value to 1 and count it. When we are done with all of the elements of the grid and the buffer array is filled with the new values of the next generation, we give the grid the values of the nextGrid and continue with the next generation. Our expected result would be our counter – **timesBeenGreen**, which would be displayed on the console.