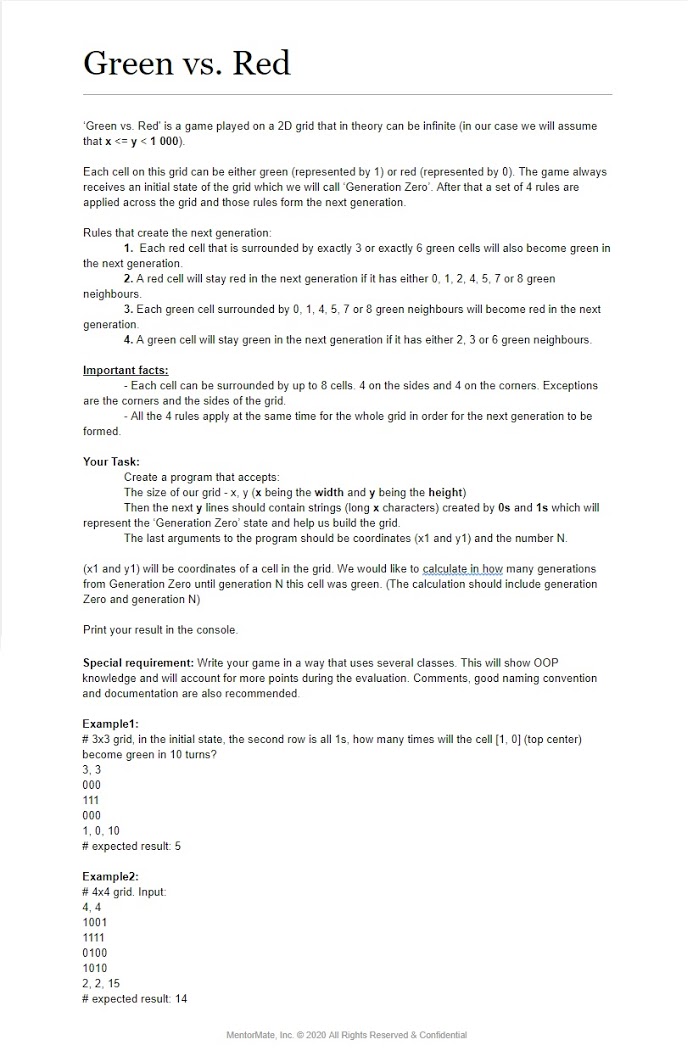
Green vs. Red

Assignment:

First, we have a grid, whose initial state is called “Generation Zero”. With the Generation Zero, we form another grid, who is called Next Generation. This means that we have to create one parent class for the Grid which will have the following attributes = > x (the width), y (the height) and a two dimensional array where we will store the cell states of a grid.

**public** **class** Grid {

**int** x,y;

**int**[][] grid = **null**;

}

Then, we need to create the subclasses Generation Zero and GenerationN who inherit Grid (the superclass) (because they will have the attributes of the superclass). The Generation Zero has attributes x1 and y1 (the coordinates of a cell in the grid) and N, which is the number of the n-th generation. In addition, we will create the input method in this class and call it in the main. Our program should accepts the width and the height of the grid, so we create a method gridWidthAndHeightValues() with a return value – an array whose elements are the length of the generation zero.

**public** **int**[] gridWidthAndHeightValues() {

String gridSize;

String[] parts = **null**;

**do** {

gridSize = input.nextLine();

gridSize = gridSize.replaceAll(", ", ",");

parts = gridSize.split(",");

}**while**(parts.length!=2 || !isNumeric(parts[0]) || !isNumeric(parts[1])

|| (Integer.*parseInt*(parts[0]))>=1000||(Integer.*parseInt*(parts[1]))>=1000

|| (Integer.*parseInt*(parts[0]))>(Integer.*parseInt*(parts[1])));

**int**[] array = Arrays.*stream*(parts).mapToInt(Integer::*parseInt*).toArray();

**return** array;

}

We create a variable from type String and assign it with value entered in the console. We use do-while loop for this purpose. Then with replaceAll method we replace delimiter “, ” with delimiter “,” in order not to have problems while entering the values (for example: we can enter 3, 3 as well as 3,3). Then we split the String into parts and fill the String array that we created in the beginning. We check if the input numbers are two or if the numbers are numeric, not letters or if x<=y<1000. If one of the conditions is not true, the loop will repeat as long as the condition is true. Finally, we create a 2D int array and assign it the parts array as we change the array to an integer array, using stream with lambda expressions. Similarly, we create a method coordinatesAndNValues() for the last arguments we have to enter in the console.

**public** **int**[] coordinatesAndNValues() {

String cellCoordinatesAndNString;

String[] lastInputParts = **null**;

**do** {

cellCoordinatesAndNString = input.nextLine();

cellCoordinatesAndNString = cellCoordinatesAndNString.replaceAll(", ", ",");

lastInputParts = cellCoordinatesAndNString.split(",");

}**while**(lastInputParts.length!=3 || !isNumeric(lastInputParts[0]) || !isNumeric(lastInputParts[1]) || !isNumeric(lastInputParts[2])

|| (Integer.*parseInt*(lastInputParts[0])<0 || Integer.*parseInt*(lastInputParts[0])>=x)

|| (Integer.*parseInt*(lastInputParts[1])<0 || Integer.*parseInt*(lastInputParts[1])>=y));

**int**[] array = Arrays.*stream*(lastInputParts).mapToInt(Integer::*parseInt*).toArray();

**return** array;

}

We check if the input contains only three numbers, are they numeric, or are they valid indexes. Now, we can create userInput() method.

**public** **void** userInput() {

**int**[] sizeArray = gridWidthAndHeightValues();

**int** x = sizeArray[0];

**int** y = sizeArray[1];

**this**.y = y;

**this**.x = x;

**this**.grid = **new** **int**[y][x];

**int** height = y;

StringBuilder stringBuilder = **new** StringBuilder();

**while**(height!=0){

String cellValues;

**do**

{

cellValues = input.nextLine();

} **while** (cellValues.length()!=x || !*checkString*(cellValues));

stringBuilder.append(cellValues);

cellValues = **null**;

height--;

}

fillGrid(stringBuilder);

**int**[] coordinatesAndNArr = coordinatesAndNValues();

**this**.x1 = coordinatesAndNArr[0];

**this**.y1 = coordinatesAndNArr[1];

**this**.N = coordinatesAndNArr[2];

}

Firstly, we declare an int array sizeArray and initialize it calling the method gridWidthAndHeightValues(). Then, initialize the attributes x, y and determine the size of the GenerationZero array(new int[y][x] -> y is in the first place, because the first argument is for the rows of the array and in our case y is the height = rows). In order to build the grid, we enter in the console as many times as the height of our array. Again, we use do-while loop and check if the input lines length are the same as the width of our grid and check if the input parts contains only the numbers 0 and 1 (checkString method). The StringBuilder method in the next line (stringBuilder.append(cellValues);) helps us to concatenate all entered lines.

We invoke fillGrid(stringBuilder) method to set state to the grid. Finally, we set values to x1, y1 and N by creating new array and initializing it with the return method coordinatesAndNValues();

In the GenerationN Class we create a constructor with parameters x and y, the width and the height of the generationN grid. Then we have to create a method that finds how many green neighbours have each cell and return the count. We create variables for the count and for the neighbours and with a try-catch block check if a neighbour exists and if yes, we check if it is green and increment the count, when condition is true.

**Int** topLeftCorner,topCenter,topRightCorner,leftSide,rightSide,

bottomLeftCorner,bottomCenter,bottomRightCorner;

**try** {

topLeftCorner = gridArray[y-1][x-1];

**if**(topLeftCorner==1)greenCount++; }

**catch**(ArrayIndexOutOfBoundsException exception) {

topLeftCorner = -1;

}

Now we have to apply the rules to the cells of the grid and create the next generation. For this purpose, we create the method applyRulesToACell(). As parameters, it accepts the grid array and the coordinates of a cell. In this method we declare two variables – a temporary variable (initialize it with -1) and the green neighbours variable.

public int applyRulesToACell(int[][] gridArray, int row, int col) {

int temp = -1;

int greenNeighbours=findGreenNeighboursCount(gridArray, row, col);

if(gridArray[row][col]==0) {

switch(greenNeighbours) {

case 3: temp=1; break;

case 6: temp=1; break;

default: temp=0; break;

}

}

if(gridArray[row][col]==1) {

switch(greenNeighbours) {

case 2: temp=1; break;

case 3: temp=1; break;

case 6: temp=1; break;

default: temp=0; break;

}

}

return temp;

}

Then we check if the cell with the given coordinates is zero or one. With switch statements check how many green neighbours the cell has, apply the rules to it and set the new value to the temporary variable.

Yet we can create the method who finds the next generation applying the rules to the given array’s cells.

**public** **int**[][] createNextGeneration(**int**[][] gridArray) {

**for**(**int** i=0;i<grid.length;i++)

**for**(**int** j=0;j<grid[i].length;j++) {

**this**.grid[i][j]=applyRulesToACell(gridArray, i, j);

}

**return** **this**.grid;

}

Finally, we create a method that checks how many times a cell is green from generation Zero to generation N. We will keep the result in an int variable. In order to include the generation Zero we have an if-clause to check if the given cell is green in the initial grid and if the condition is true, we increment the result variable. Then, we create a temporary 2D array, which will become the next generation and we reset the GenerationN array. Then we check if the cell is green in the next generation and if yes, increment the result and when the for-loop ends, we print result.

**public** **void** ifCellIsGreenInNextGenerations(**int**[][] gridArray, **int** x1,**int** y1,**int** N) {

**int** result = 0;

**if**(gridArray[y1][x1]==1)result++;

**int**[][] temp=**new** **int**[**this**.x][**this**.y];

**for**(**int** i=1; i<=N; i++) {

temp = createNextGeneration(gridArray);

gridArray=temp;

**this**.grid = **new** **int**[y][x];

**if**(temp[y1][x1]==1)result++;

}

System.***out***.println("# expected result: "+result);

}

In the main, we create objects of type GenerationZero and Generation N and we invoke the methods to input the required data and to return the result.

**public** **static** **void** main(String[] args) {

GenerationZero generationZero = **new** GenerationZero();

generationZero.userInput();

GenerationN generationN = **new** GenerationN(generationZero.x,generationZero.y);

generationN.ifCellIsGreenInNextGenerations(generationZero.grid, generationZero.x1, generationZero.y1, generationZero.N);

}