American University of Armenia
CS 120 Intro to OOP
Spring 2019

Homework Assignment 4, Part 1

In the next few homework assignments you will be developing a Game of Life[1] implementation. Please read the Wiki article and make sure you understand how Game of Life works. We will be developing the game by adding more features or by rewriting the code to be neater/more robust/more extensible/better.

1. **(20 points)** The game board can be represented using arrays. Produce an `ArrayLife.java` file containing the definition of the class `ArrayLife` that represents a Game of Life where the underlying board is represented by a 2D array of `boolean` values. Thus, your class should look like:

```
public class ArrayLife {

  private int width;
  private int height;
  private boolean[][] world;

  // ...
}
```

These instance variables are `private` because we are applying good encapsulation/information hiding so the only way to interact with the array is via our own methods (that presumably enforce the rules of the game). Your class should include methods with the following headings:

- `boolean getCell(int col, int row)`: accessor for a specific cell
- `void setCell(int col, int row, boolean value)`: mutator for a specific cell
- `void print()`: printing the state of the whole board
- `int countNeighbours(int col, int row)`: counting the number of neighbours alive
- `boolean computeCell(int col, int row)`: determining if the cell will be alive or dead in the next generation, based on the rules of the game
- `void nextGeneration()`: updating the game board to the next generation
- `void play()`: while the user inputs the character `'s'`, printing the game board and advancing to the next generation; stops when the user inputs `'q'`

For instance, the `getCell` method implementation may look like:

```
public boolean getCell(int col, int row) {
  if (row < 0 || row >= height) {
    return false;
  }
  if (col < 0 || col >= width) {
    return false;
  }
  return world[row][col];
}
```

[1] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Note that your methods may need to invoke each other or you may need to define some extra methods. For each of the methods, determine and add the correct access modifier and explain your choices in a brief report (1–2 paragraphs).

2. (10 points) Your next task is to add a specification of a board that uses `String`s.

The Format

The format you need to support is

NAME:AUTHOR:WIDTH:HEIGHT:STARTUPPERCOL:STARTUPPERROW:CELLS

where NAME is the name given to the board layout, AUTHOR is the name of the author, and WIDTH and HEIGHT describe the board dimensions. Rather than specifying the state of all cells in the world, the format assumes that cells are dead unless explicitly specified otherwise. Consequently CELLS is used to represent a subset of the board that contains all the live cells. (Most cells in a typical world are dead so this optimisation is frequently useful.) The values of STARTUPPERCOL and STARTUPPERROW specify the location of the cells recorded in CELLS in the world. The details of the format are perhaps best explained with the aid of the following example:

Glider:Richard Guy:20:20:1:1:010 001 111

The above example describes a world of 20 cells by 20 cells with a "Glider" in it. Gliders were discovered by Richard Guy in 1970. The contents of CELLS is 010 001 111 and records live cells with a one (1) and dead cells with a zero (0) in row order, using spaces to separate rows. Therefore, in the above example, the CELLS part of the format states that cells (1,0), (2,1), (0,2), (1,2) and (2,2) are alive. The values of STARTUPPERCOL and STARTUPPERROW should be added to the values recorded in CELLS and therefore the live cells in the world at generation zero are (2,1), (3,2), (1,3), (2,3) and (3,3); all other cells are dead. The layout of this world is shown in Figure 1.

Remember that this format assumes that cells are initialised as *dead* unless they are explicitly marked as alive in the CELLS section of the format string. There is no requirement for the CELLS to contain equal-sized rows: anything not explicitly given is implicitly false. So, for example:

Glider:Richard Guy:20:20:1:1:01 001 111

is actually the same as:

Glider:Richard Guy:20:20:1:1:010 001 111

since the "0" on the end of the "010" is implicit.

Using Strings in the Game of Life

Our aim is to be able to specify patterns as strings on the command line:

```
> java ArrayLife "Glider:Richard Guy:20:20:1:1:010 001 111"
```

Explain in your report why it is essential to use the double quotes (") around the format string.

Now you need to write a new constructor that initialises an `ArrayLife` object using strings:

```
public ArrayLife(String format) {
  //TODO: Determine the dimensions of the game board
  width = ...
  height = ...
```

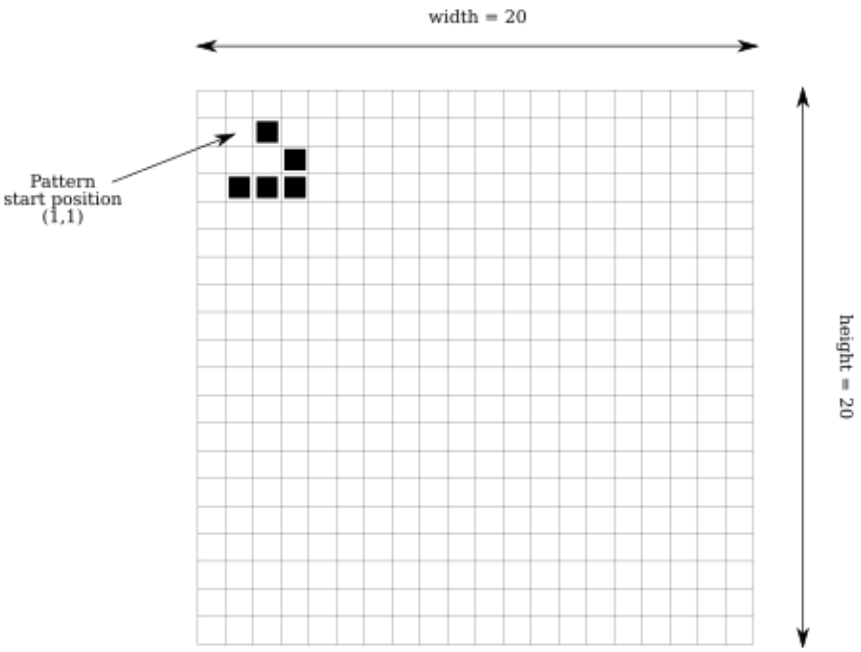

Figure 1: The Glider in a 20-by-20 world.

```
  world = new boolean[height][width];

  //TODO: Using loops, update the appropriate cells of world to 'true'
}

// ...

public static void main(String[] args) {
  ArrayLife al = new ArrayLife(args[0]);
  al.play();
}
```

3. (20 points) Each format string contains a lot of information, some of which we just discard (author, name). We'd like to retain all of the information and make it accessible. We could just add some more state to `ArrayLife` (e.g. `String author`) but actually each pattern is a well-defined concept that will benefit from being represented as a class.

By creating a file of the correct name, create a class called `Pattern`. Use the following code as the basis for writing the class `Pattern`:

```
public class Pattern {

  private String name;
  private String author;
  private int width;
  private int height;
  private int startCol;
  private int startRow;
  private String cells;

  //TODO: write public 'get' methods for ALL of the fields above;
```

```
  //      for instance 'getName' should be written as:
  public String getName() {
    return name;
  }

  public Pattern(String format) {
    //TODO: initialise all fields of this class using contents of
    //      'format' to determine the correct values (this code
    //      is similar to that you used in the new ArrayLife constructor
  }

  public void initialise(boolean[][] world) {
    //TODO: update the values in the 2D array representing the state of
    //      'world' as expressed by the contents of the field 'cells'.
  }
}
```

Test your class by adding a main method to `Pattern` that takes in a pattern string as an argument, creates a `Pattern` object and then prints out the various components of the pattern. E.g.

```
> java Pattern "Glider:Richard Guy:20:20:1:1:010 001 111"

Name:  Glider
Author:  Richard Guy
Width:  20
Height:  20
StartCol:  1
StartRow:  1
Pattern:  010 001 111
```

To integrate `Pattern` into `ArrayLife` we want to provide each `ArrayLife` instance (object) with its own `Pattern` object. Therefore we make a new member field:

```
public class ArrayLife {

  private int width;
  private int height;
  private boolean[][] world;
  private Pattern pattern;

  public ArrayLife(String format) {
    // TODO: setup pattern, world, height and width
    pattern.initialise(world);
  }

// ...
```

Adapt your `ArrayLife` as shown, completing the constructor.

Test your class.