6 Instruction 2IP67.5

6.1 Files

1. The following program (last few lines are omitted) reads the file “rhubarb.txt” line by line and prints it, line by line, to the console.

```
// reads and prints file line by line
import java.util.*;
import java.io.*;

public class FileEcho {
    Scanner scanner;
    File file ;
    String filename = "rhubarb.txt";

    void echo() {
        try {
            String line = null;
            file = new File( filename);
            scanner = new Scanner( file );

            while (scanner.hasNext()) {
                line = scanner.nextLine();
                System.out.println( line );
            }
        } catch(FileNotFoundException e) {
            System.out.println( "Could not open file "+filename+"\n"+e );
        }
    }
```

   Download this file from the wiki (`http://www.win.tue.nl/~keesh/dokuwiki/doku.phpid=2ip65_en_2ip70`), make a text file[1], with the name “rhubarb.txt” and execute it. NetBeans users should put this file in the top level folder of the NetBeans project. This folder has the same name as the project and contains folders such as “build”, “dist“, “src”. An alternative is to use a complete path, something along the lines of `"C:\\My Documents\\Java\\Week6\\rhubarb.txt"`, or to change the working directory of your NetBeans project.

2. Modify this program to have it print each line backwards. There are numerous ways to do this, but as a preparation for the following exercise, use the method call `line.toCharArray()` which returns the contents of `line` an array of characters (`char[]`). Then print the characters one by one, backwards.

3. Now have the program leave out every character `e` when printing. Note that a `char` value can be expressed in Java with single quotes, such as `'e'`, `'0'`, `'*'` . Furthermore, the type `char` is similar to `int` and values can be compared with `==`.

6.2 The Game of Life

The Game of Life was devised by the British mathematician John Horton Conway in 1970. The “game” actually describes how an initial configuration evolves over time according to simple, precise rules. Despite the simple and precise rules, the evolution from the initial configurations is hard to predict and quite similar looking initial configurations may give rise to quite different developments.

The universe of the Game of Life is a two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its maximally eight neighbours, the immediately adjacent cells in orthogonal and diagonal direction. Cells on the border have less than eight neighbours. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if by loneliness.
2. Any live cell with more than three live neighbors dies, as if by overcrowding.
3. Any live cell with two or three live neighbors lives, unchanged, to the next generation.

[1]Use a program of your choice, a text editor, MS Word (save in plain text format), NetBeans, etc.

4. Any dead cell with exactly three live neighbors comes to life.

An initial generation is read from file. A new generation is created by applying the above rules on every cell, using the current generation, as if the rules were applied simultaneously on all cells.

Write a program, using the following steps.

1. Represent the playing field as a two dimensional array. Add a border of always-dead cells around the playing field (grid), in order to simplify the neighbour counting. It is up to you how you represent cells. Declare the grid as an instance variable.
2. Write a method that intialises the grid to a simple generation that is suited for testing.
3. Write a method that displays the matrix on the screen, each cell a square filled with a * indicating life, and empty if dead. Use a space of one space between cells to get a more square-like output.
4. Write a function that has as parameters the coordinates of a playing field cell and returns the number of living neighbors that that cell has.
5. Write a method that runs through all cells and, using the number-of-neighbors function and the game's rules, fills a new matrix with the appropriate dead/alive values.
6. Write a method that copies the new matrix into the old one.
7. Write a method that, using the above parts, puts the whole game together, asks for the number of generations to display, and generates the generations. If you use a sleep between two generations and if you adapt the window height to the height of the output of one generation, you will get a nice animation, which means originally "instilling with life", by the way.
8. To enable experimenting, add a method that reads the initial generation from file. The format of such a file is:
   (a) two integers specifying the width and height of the grid
   (b) lines with spaces and stars in the same format as the output

   There should be no empty lines between the integers and the first line of the grid.

   Make the input method a little bit robust by allowing fewer lines than the height of the grid and lines that are shorter than the width of the grid. Make an option in your program to read the first generation from a file. Preferably with the possibility to specify a file or use a default.
9. Add a method that creates a text file with the current generation, in the format that the input method expects. Have the program write the final generation to a file.
10. Create an option for the user to write the current generation to a file of his choosing.

Extras

Just choose some, or dream-up your own variants.

1. Enable to repeatedly play the game.
2. Output with each generation its number of occurrence.
3. Let the game stop as soon as a stable generation is reached.
4. Make sure that the game stops if in an interval of 5 generations the same generation occurs.
5. Make sure that the game stops if within a number chosen by the user the same generation occurs.
6. Change the representation so as to not just indicate being alive in a cell, but a number stating its age, i.e, how long this cell was alive uninterruptedly (if the number gets too large too put in the cell, use a different symbol, like #, to indicate that it is older than, e.g., 9).
7. In a two-party version, live cells exist in a two symbols variety. A player wins when all cells of the opponent's kind are eliminated. When a dead cell becomes live, its symbol is determined by the dominating symbols of its neighbor live cells (which are exactly three). Start with a random or pre-chosen starting pattern with half the live cells of each kind. After one iteration, the first player may add one cell of his own and remove one cell of his opponent. After the next iteration the other player can do the same, and so forth.

Kees Huizing – 5 October 2009