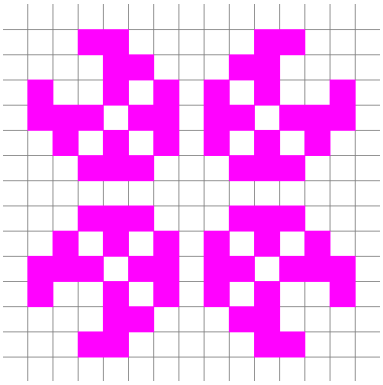
intro2cs/introcsp/introcst

Ex5: Game of Life

Objective: practice working with 2-dimensional arrays and indices in Java.

In this exercise, your task is to implement the well-known **Game of Life**. Invented in 1968 by John Conway, the game of life is not really a game in the usual sense, but rather a simulation of an evolution of a colony of creatures.



Game Description

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 eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent to it. The game consists of a series of generations. A generation is a state of the world – which cells are alive, and which cells are dead. The game advances from the initial generation to the next generation and so on. In the process of calculating generation n+1 from generation n, the following transitions occur (all 4 rules are applied simultaneously to every cell in the current generation – births and deaths occur simultaneously):

1. Any cell that is alive and has **fewer than two live neighbours** in generation n, will die of under-population, and will thus be dead in generation n+1.
2. Any cell that is alive and has **two or three live neighbours** in generation n, will stay alive in generation n+1.
3. Any cell that is alive and has **more than three live neighbours** in generation n, will die of overcrowding, and will thus be dead in generation n+1.
4. Any cell that is dead and has **exactly three live neighbours** in generation n, will revive by reproduction, and will thus be alive in generation n+1.

See http://en.wikipedia.org/wiki/Conway's_Game_of_Life for more details. Advancing through the generations may produce interesting patterns. Examples of patterns can be found here.

As stated earlier, the world of the Game of Life is a two-dimensional orthogonal grid of square cells. The grid is composed of *height* rows with the top row numbered 0, and *width* columns with the left-most column numbered 0.

This world can either be a **finite world**, in which cells along a border do not have all 8 potential neighbours (e.g., a cell in the corner of the grid has only 3 neighbours); or a **cyclic world**, in which every cell has exactly 8 neighbours, by defining that stepping over the border brings one to the opposite side of the grid. For example, the top neighbor of cell (0,5) is the cell (*height*-1, 5) and the right neighbor of cell (*height*-1, *width*-1) is the cell (*height*-1, 0). This cyclic world has the topology of a torus (see http://en.wikipedia.org/wiki/Torus for more details).

Implementation

- Your main task in this exercise is to implement the API of the class **GameOfLife**. Your implementation must support both finite and cyclic worlds. As defined in the API, note that you will have to implement three constructors:
  - one that constructs a cyclic world of specified dimensions,
  - one that decides which kind of world to construct according to a boolean value passed as a parameter,
  - one that is also given a String describing the initial state of the world. (to read and process a String variable s, you can use a Scanner, created as follows : `Scanner sc = new Scanner(s);`).
- In addition, you should implement a class named **GameOfLifeDriver** that contains the *main* method that calls the methods of the **GameOfLife** class. In this class you will have to
  - parse the program's command line arguments (see below);
  - create a new game object according to the values of these arguments;
  - create a graphics window for displaying the progress of the game (using the **GridWindow** API, provided to you as part of the intro package);
  - repeatedly advance the game to the next generation and pass the result to the display window.
- To run the program, a user should type a command line of the form:
  java GameOfLifeDriver <height> <width> <file> <cyclic or acyclic>

  Each of these **command-line arguments** can be accessed in the main method via the `args` parameter (of type `String[]`), as `args[0]`, `args[1]`, and so on. The first two arguments specify the height and the width of the world, and the last specifies whether or not it is a cyclic world.

- The *<file>* argument above should be the name of a text file containing a string description of the initial state of the game world. For example, the following string describes a 5x15 world, which has some live cells in the first, second, and fifth rows (indicated by the '+' characters):

```
----------+++--

-+++-----------

---------------

---------------

+++------------
```

- Note that the string description contains a newline character (not visible above) at the end of each line (including the last line).
- In order to read the string description from a file you should follow the following instructions:
    a. The file should be located at the same path (folder) as your java class files (for simplicity).
    b. The filename should be in English, for example "pattern1.txt".
    c. In your **GameOfLifeDriver** implementation you should use the static method **`IntroUtil.newScannerFromFile(String fileName)`** to create a Scanner instance that you will use to read the file. The **`IntroUtil`** class is provided as part of the **`il.ac.huji.cs.intro`** package. Don't forget to download an up-to-date copy of the intro package, if working from home!
- You may assume that the command line parameters are valid and that the file contains a valid string description.
- You must use arrays. You may not use ArrayList objects in this exercise.
- In order to introduce a short time delay between successive game generations, you may use the method **`IntroUtil.sleep(double seconds)`**, provided as part of the **`il.ac.huji.cs.intro`** package. Don't forget to download an up-to-date copy of the intro package, if working from home!
- In order to graphically display your game as it progresses, you should use the class **`il.ac.huji.cs.intro.GridWindow`**. It is extremely easy to use: once it is created you only need to keep calling the **`update`** method of **`GridWindow.`** Closing the window will terminate your program.
- At the first stages of debugging your program, you may find it helpful to display the state of the world by printing a String description to the terminal, using the **`GameOfLife.toString()`** method. For example, a simulation of a small 5x5 cyclic world might look like this:

```
initial state:

+++--
```

```
-+---
-+---
-+---
-----
next generation:
++---
-----
++---
-----
+-+--
```
(the two creatures in this row would not be alive if this was a finite world!)

```
next generation:
+-+--
-----
-+---
+-+--
+-+--
next generation:
-----
-+---
-+---
+-+--
+-+++
next generation:
+++++
-----
++---
+-+--
+-+++
next generation:
-----
```

```
-----
+-+--
-----
-----
next generation:
-----
-----
-----
-----
-----
next generation:
-----
-----
-----
-----
-----
```

and so on (when all the cells are dead, none of them can revive).

School solution

An executable version of the school solution can be invoked from the lab computers by typing: `~introcsp/bin/ex5/gameOfLifeDriver <args>` where args is a list of command line arguments as described earlier. You can find two example input files here and here (you are welcome to use them for testing your implementation, and you are encouraged to write your own input files). Put the input file in the same directory from which you invoke the above command.

Testing

It is recommended to check your JAR file by copying it to an empty directory, and running the automatic testers by executing the command: `~introcsp/bin/testers/ex5 ex5.jar` and verifying that all the tests pass successfully.
You may also download the tests JAR from here. Extract the contents of the JAR file using the shell command: `jar xvf ex5testing.jar` and follow the instructions in the file called `TESTING`.

Further Guidelines

- Please remember to follow the all the course coding style guidelines.
- In particular, remember to use constants, where appropriate, and refrain from duplication of code.
- Document your code – add comments before all major code blocks.
- Write complete Javadoc comments whenever appropriate.
- Don't forget to write a short description of each class in your **`README`** file.

Submission

Submit a JAR file called **`ex5.jar`** containing **only** the following three files:

1. **`GameOfLife.java`**
2. **`GameOfLifeDriver.java`**
3. **`README`**

Use the "Upload File" link on the course home page, under the ex5 link.