User instructions.

Cellular automaton

Table of contents

0.1 Running the project

The **main** is located in the package **gridAutomaton** in **Main.java**. GUI is located in the package **gui**. There is two ways to run the automaton :

0.1.1 Graphic interface

Set the boolean **console** to false :

```
boolean console = false;
```

0.1.2 Console

If you can't use the GUI, note that it's still possible to run it in the console. Set the boolean **console** to true :

```
boolean console = true;
```

The output looks like this :

```
Rules :
<html>
- Any dead cell with exactly 3 live neighbors becomes a live cell.<br />
- Any dead cell with exactly 6 live neighbors becomes a live cell.<br /></html>
<html>
- Any live cell with exactly 2 live neighbors lives.<br />
- Any live cell with exactly 3 live neighbors lives.<br /></html>
-------------------------

Turn : 0
--------------
|*|*| | |*| |*|
--------------
|*| |*|*| | | |
--------------
|*| |*| |*| | |
--------------
| |*|*| | |*| |
--------------
| | | | | | |*|
--------------
|*|*| |*|*|*|*|
--------------
|*| |*|*|*| |*|
--------------
[ Live cells : 25 ]
Press enter to continue or type any key to stop.
```

0.2 Settings

Before running the automaton, you can modify the grid's settings : the outputs will be the same in console or GUI. In **main.java** :

0.2.1 Grid's size, initial live cells

• To change the grid's size, set the values of **rows** and **columns**, for example :

```
int rows = 10;
int columns = 10;
```

• To change initial live cells, set the value of **liveCells**, for example :

```
int liveCells = 20;
```

Note that for a grid's size superior or equal to 25×25, a stackoverflow might happens or slowing downs may appear. For that reason, set the grid's size between 2 and 24. Besides, initial live cells' location are randomly selected, if the number of live cells' generated is bigger than the grid's size then not any live cell is set.

To set live cells on a specific location, just after this line :

```
SquareGrid grid = new SquareGrid(rows, columns, rules);
```

Comment this :

```
grid.initialize(liveCells);
```

and add :

```
grid.cell[i][j].change();
```

where (i, j) refers to the cell's location.

0.2.2 Rules to pass onto the next generation

The default rule is Highlife's, stating that :

- any dead cell with 3 or 6 live neighbors becomes a live cell onto the next generation,
- any live cell with 2 or 3 live neighbors stays alive onto the next generation otherwise it dies.

Another rule that can be added is Conway's, uncomment these lines :

```
/* Conway's rules */
lifeRule.add(3);
deathRule.add(2);
deathRule.add(3);
```

and comment these ones :

```
/* Highlife's rules */
lifeRule.add(3);
lifeRule.add(6);
deathRule.add(2);
deathRule.add(3);
```

To add its own rules, the object **LifeRule** states that if a dead cell matches it then it becomes a live cell. And the object **DeathRule** states that if a live cell matches it then it dies. It's not necessary to know how those have been written. What you only need to do is adding the number of live neighbors that a cell needs to match to born/die.

Use the method **add** of these objects and comment the previous rules (Highlife and Conway's), here's an example of adding its own rules :

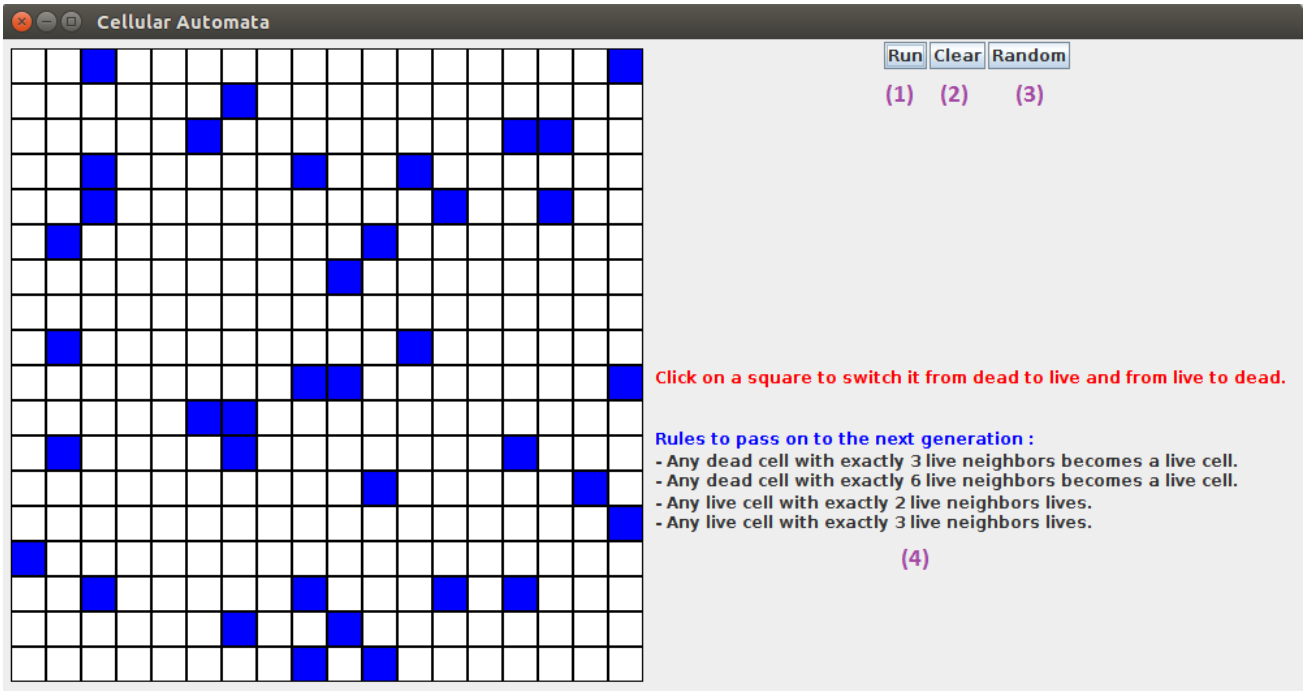
```
/* My rules */
lifeRule.add(2); /* a dead cell with 2 live neighbors becomes
                    a live cell onto the next generation */
lifeRule.add(8); /* a dead cell with 8 live neighbors becomes
                    a live cell onto the next generation */

deathRule.add(4); /* a live cell with exactly 4 live neighbors
                     stays alive onto the next generation */
deathRule.add(7); /* a live cell with exactly 7 live neighbors
                     stays alive onto the next generation */
```

If a cell can have more than 2 states, be sure that only and only one of the implemented rules can be true for a given state (this will be more detailled in the rule's section).

0.3 In the console

To pass onto the next generation, press **Enter** on your keyboard. A live cells is shown by an asterisk *. The program will stop if the user types any character or if all cells are dead.

0.4 In graphic interface



1. **Run** : executes one turn.
2. **Clear** : all the cells become dead.
3. **Random** : generates random live cells.
4. Describes the rules for a cell to pass onto the next generation.

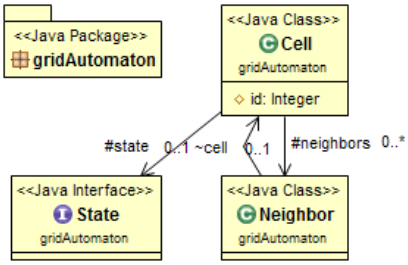
A live cell is a blue square ■ and a dead cell is a white square □. You can switch the state of a cell by clicking on it.

0.5 Implementation

For methods's description, cf. documentation.

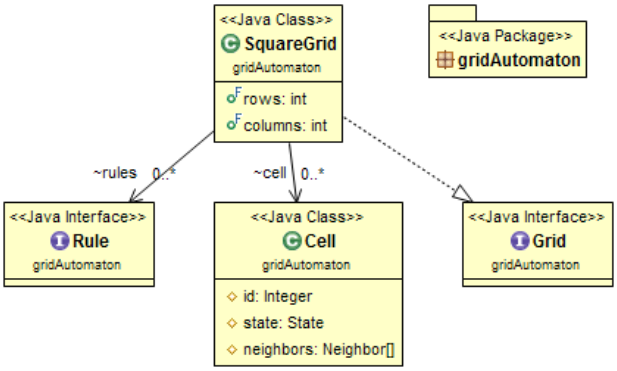
0.5.1 Cell

A cell contains its state and its neighbors :



0.5.2 Grid

A grid contains a list of the cells and rules :

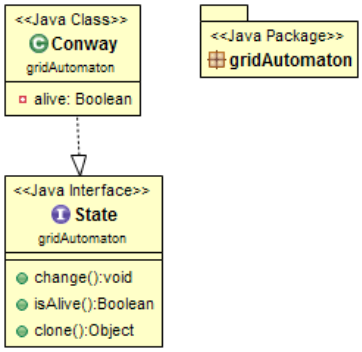


The default grid is square and toric. Each cell has exactly 8 neighbors :

[North-West]	[North]	[North-East]	(1)
[West]	[c]	[East]	(2)
[South-West]	[South]	[South-East]	(3)

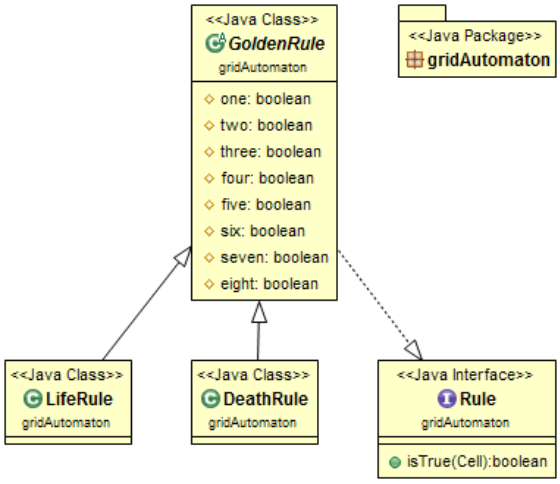
0.5.3 State

There is only one state implemented, Conway's one where a cell can be either alive or dead but not both at the same time :



0.5.4 Rules

Rules **LifeRule** and **DeathRule** are efficient when a cell's state can be representated with a boolean :



If a rule needs to implement more specific conditions, create a new class implementing the interface **Rule** and add it to the grid's list of rules : do it in the **main.ava**, with the list **rules** e.g. :

```
rules.add(myNewRule);
```

As it was said, when adding a new rule, make sure that only one of these rules can be true for a given cell's state. Let's assume there is 3 states : live, dead and zombie and 3 rules : lifeRule, deathRule and zombieRule. So for a given cell's state, a cell can't become dead and zombie at the same time.

0.6 Next generation

The next state of a cell is kept in a deep clone (cf. Cell's **deepCopy** method). It was intended to have a button "Previous generation" but eventually, it was not implemented. Besides, this deepCopy method requires a great quantity of RAM that's why a stackoverflow might occurs.