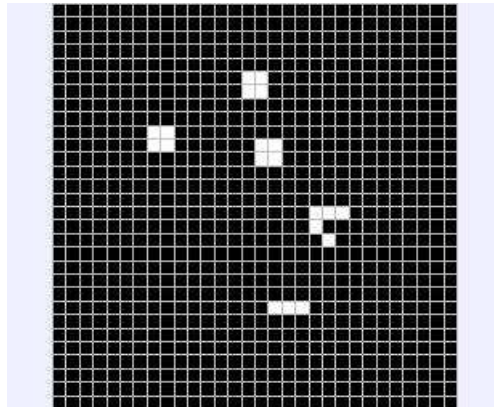


## John Conway's Game of Life



### Description

Taken from **Wikipedia**, the free encyclopedia.

See [http://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton.

The "game" is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. A variant exists where two players compete.

### Rules

The universe of the Game of Life is a two-dimensional orthogonal grid of square cells with periodic boundaries (leaving the grid at one side makes you appear at the opposite side of the grid), each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbours, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

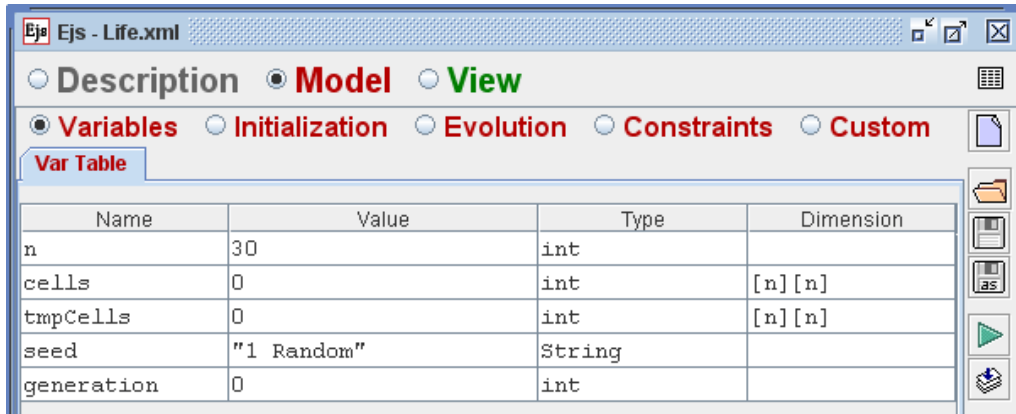
- Any live cell with fewer than two live neighbours dies, as if by loneliness.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours comes to life.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed-- births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

## Model

### Variables

We only need the following basic set of variables:



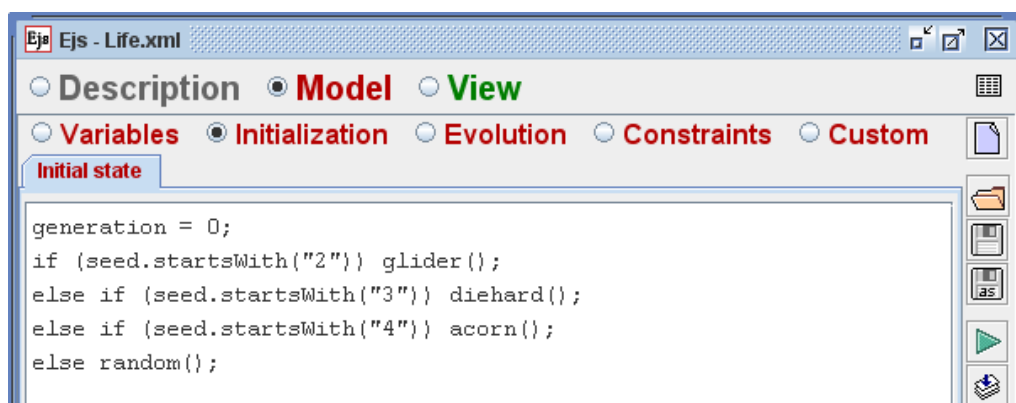
Name	Value	Type	Dimension
n	30	int	
cells	0	int	[n] [n]
tmpCells	0	int	[n] [n]
seed	"1 Random"	String	
generation	0	int	

- The integer  $n$  indicates the number of cells in each direction (horizontal and vertical).
- The array *cells* contains either a 0 (dead) or a 1 (alive).
- The auxiliary *tmpCells* array is used to hold a copy of a generation when computing the next one.
- The string *seed* is used to allow the user to choose particular configurations for the initial state.
- Finally, *generation* indicates the generation number currently visualized.

### Initialization

We have chosen to have four possible configurations. The *seed* string variable is selected using a combo box in the view. Depending on the initial numeric characters of the seed, we will choose one of the predefined initial states which are described in the **Custom** section below. (The reason to use only the numeric values is that it allows for easy internationalization of the configuration names.)

The code for the initialization is, hence:



```
generation = 0;
if (seed.startsWith("2")) glider();
else if (seed.startsWith("3")) diehard();
else if (seed.startsWith("4")) acorn();
else random();
```

Notice however that the *On Press* action property of the drawing panel in the view will allow us to switch the state of individual cells by simply clicking on them.

## Evolution

The evolution consists in applying the rules described above to compute a new generation. We need, however, to make a copy of the current state before modifying it. Otherwise, undesired side-effects could occur. (If we used the *cells* array all the time, changing the state of a given cell would mean that the next generation of the cell to its right will be decided using the next generation of the first cell, not its current generation!)

We have preferred to create a separate custom method called *checkNeighbors* for the (a bit complex) task of computing how many neighbouring cells are alive.

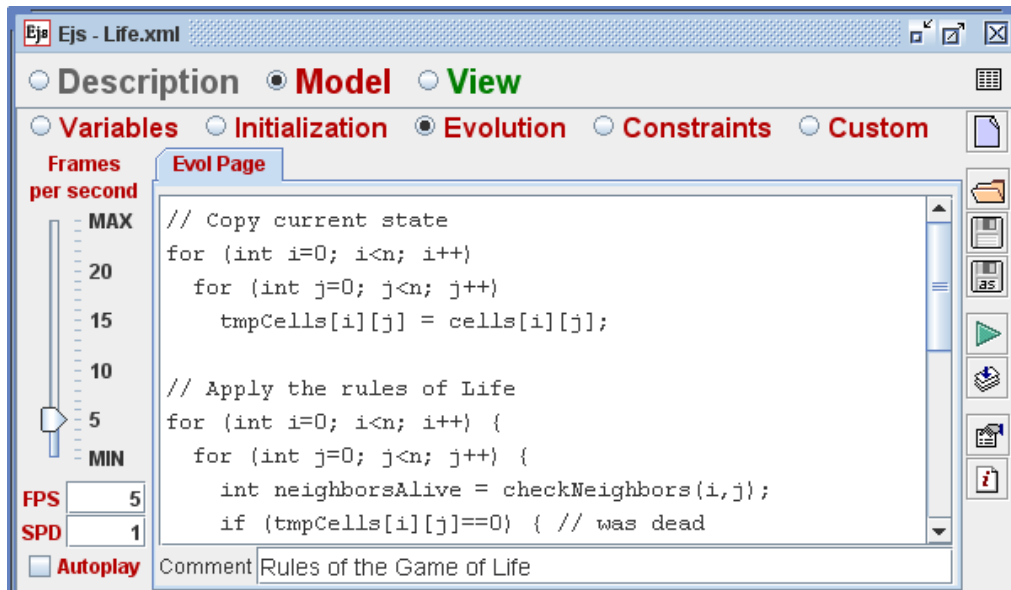
The Evolution code is then:

```
// Copy current state
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        tmpCells[i][j] = cells[i][j];

// Apply the rules of Life
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        int neighborsAlive = checkNeighbors(i,j);
        if (tmpCells[i][j]==0) { // was dead
            if (neighborsAlive==3) cells[i][j] = 1; // Birth
        }
        else { // was alive
            if (neighborsAlive<2) cells[i][j] = 0; // Loneliness
            else if (neighborsAlive>3) cells[i][j] = 0; // Overcrowding
        }
    } // end of for j
} // end of for i

// Next generation
generation++;
```

Notice in the picture below that we have set a number of 5 frames per second and chose not to start the game automatically at start up (i.e. the *Autoplay* field is unchecked).



## Constraints

No constraints are required.

## Custom code

We need two pages of custom code. The first one holds the *checkNeighbors* method. Its code is as follows:

```
public int checkNeighbors(int i, int j) {
    int i1 = i-1, i2 = i+1, j1 = j-1, j2 = j+1;
    // Periodic boundaries
    if (i1<0) i1 = n-1;
    else if (i2>=n) i2 = 0;
    if (j1<0) j1 = n-1;
    else if (j2>=n) j2 = 0;
    int counter = 0;
    if (tmpCells[i1][j1]!=0) counter++;
    if (tmpCells[i1][j ]!=0) counter++;
    if (tmpCells[i1][j2]!=0) counter++;
    if (tmpCells[i ][j1]!=0) counter++;
    if (tmpCells[i ][j2]!=0) counter++;
    if (tmpCells[i2][j1]!=0) counter++;
    if (tmpCells[i2][j ]!=0) counter++;
    if (tmpCells[i2][j2]!=0) counter++;
    return counter;
}
```

In the secondpage, we write teh code for the intiaal configurations allowed for in our combo box. These are implemented by the following methods:

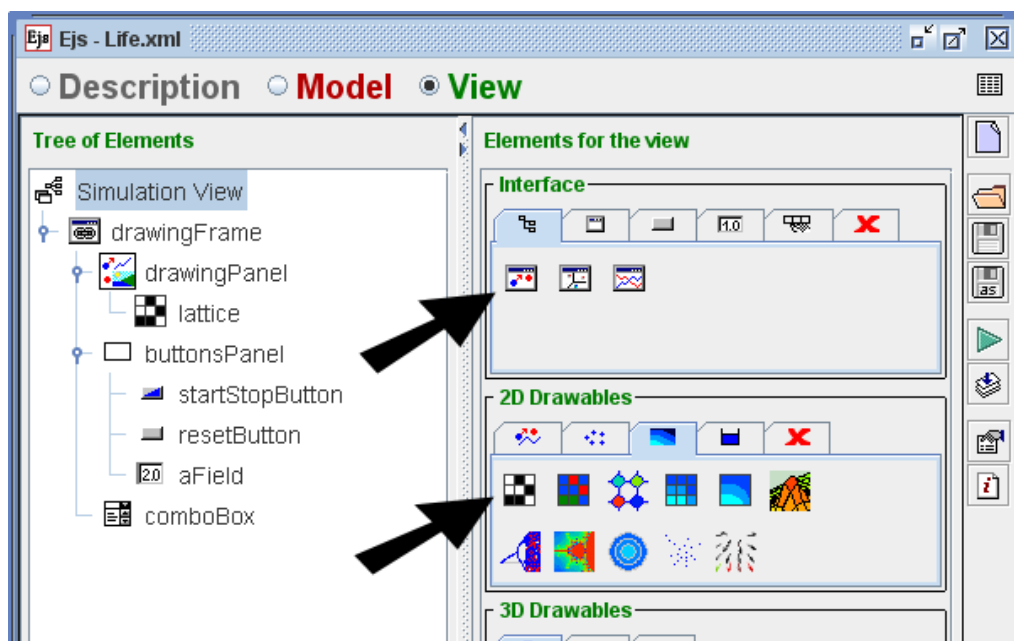
```
public void random () {
    for (int i=0; i<n; i++) for (int j=0; j<n; j++) {
        if (Math.random(<0.9) cells[i][j] = 0;
        else cells[i][j] = 1;
    }
}
```

```
public void glider () {  
    for (int i=0; i<n; i++) for (int j=0; j<n; j++) cells[i][j] = 0;  
    cells[1][0] = 1;  
    cells[0][1] = 1;  
    cells[0][2] = cells[1][2] = cells[2][2] = 1;  
}  
  
public void diehard () {  
    for (int i=0; i<n; i++) for (int j=0; j<n; j++) cells[i][j] = 0;  
    cells[1][0] = cells[5][0] = cells[6][0] = cells[7][0] = 1;  
    cells[0][1] = cells[1][1] = 1;  
    cells[6][2] = 1;  
}  
  
public void acorn () {  
    for (int i=0; i<n; i++) for (int j=0; j<n; j++) cells[i][j] = 0;  
    cells[0][0] = cells[1][0] = cells[4][0] = cells[5][0] = cells[6][0] = 1;  
    cells[3][1] = 1;  
    cells[1][2] = 1;  
}
```

## View

The view starts with the compound element based on a drawing panel in which we have removed the default particle included in this panel and replaced it with the Lattice 2D drawable. The compound element and the drawable are pointed to big arrows in the figure below.

We have also added a ComboBox element (in the *Input and output* subpanel of the *Interface* group) for the selection of the seed.



The main properties that need to be modified are shown in the colored fields of the property panels below:

Properties for lattice (Lattice)					
Configuration			Graphical Aspect		
Data	cells		Visible		
Minimum X	-1.0		Dead Color		
Maximum X	1.0		Alive Color		
Minimum Y	-1.0		Show Grid		
Maximum Y	1.0		Grid Color		

Properties for aField (Field)					
Main			Graphical Aspect		
Variable	generation		Size		
Value			Background		
Format	Generation = 0		Foreground		
Editable	false		Font		
Action			Tooltip		

Properties for comboBox (ComboBox)					
Main			Graphical Aspect		
Options	1 Random;2 Glider		Size		
Variable	%seed%		Field Bkgd		
Value			Background		
Editable			Foreground		
Action	initialize();		Font		
			Tooltip		

The Options field lists the four initial configurations: 1 Random; 2 Glider; 3 Diehard; 4 Acorn. Click on the Options editor icon to view and edit the list.

Resides these properties, the *On Press* action property of the drawing panel has the following code associated to it:

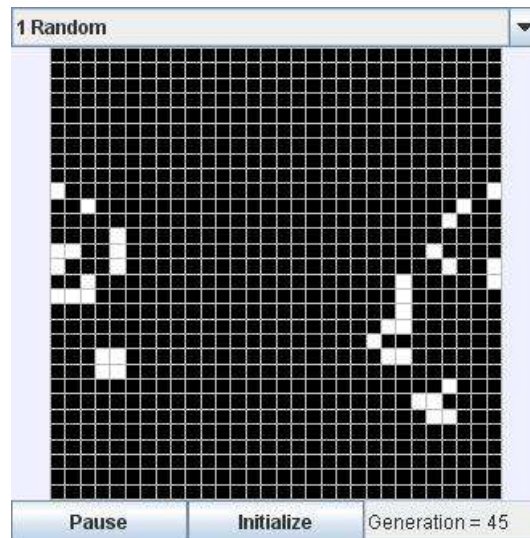
```
// get the coordinate chosen
double x = _view.drawingPanel.getMouseX();
double y = _view.drawingPanel.getMouseY();
// compute the indexes
int i = (int) (n*(x + 1)/2.0);
int j = (int) (n*(y + 1)/2.0);
// Switch the state of the cell
cells[i][j] = 1 - cells[i][j];
```

(The code makes use of the *getMouseX* and *getMouseY* methods of the drawing panel which return the coordinates of the point clicked upon.)

The code above has the effect of switching the state of a cell when the user clicks on it. This allows the user to explore interactively further configurations.

## ***Running the simulation***

Here is a sample execution (from a random configuration):



## ***Author***

Francisco Esquembre  
Universidad de Murcia, Spain  
July 2007