Project 2

The Game of Life

100 points

Assignment posted on February 26, 2018.

DUE: March 24, 2018

Last day of submission: March 31

Last day of project advising: March 21

Learning Outcomes

- Recognize the need for arrays in programming tasks, and manipulate data in one and multi-dimensional arrays (b, c, i, j)
- Utilize the ArrayList class for managing lists of objects (i, j)
- Design and implement multi-class solutions to programming problems (b, c, i)
- Apply and utilize fundamental GUI components and operations (c, h, i)
- Utilize event driven programming and interfaces in advanced GUI applications (b, c, i)

Objectives

In completing this project you will gain experience with the following Java features:

- ArrayLists
- building GUI with various components
- listeners, event driven GUI elements

Problem Statement

The Game of Life is a model that simulates how a population of living things evolves as a result of interaction with each other and the environment. This is not a traditional game of playing for a stake, but rather a computer produced series of generations in which the distribution of life is unpredictable and it leads to several interesting and difficult questions.
The environment is not borderless, we may assume that it is confined to an island. This island is represented in the game as a rectangular grid of squares such that within each square and at any given time life may or may not be present, thus we speak about living and dead squares. From one generation to the next a living square may die or survive, a dead square may come to life or stay dead. All these events are controlled by the number of living neighbors of the square and the pre-set life parameters. Consider the following example of typical life parameters applied in the game (including your program):

(i) life preserving limits for living squares: 2 and 3
(ii) life creating limits for dead squares: 2 and 3

Given these limits the following rules apply:

If a square is currently alive

- it remains alive if it has two or three living neighbors
- it dies of loneliness if it has fewer than two living neighbors
- it dies of overpopulation if it has four or more living neighbors

For a square currently dead

- it comes to life if it has exactly two or three living neighbors
- it remains dead in all other cases

The purpose of the game and of your project is that starting out from a given initial generation the program creates and stores a sequence of subsequent generations until life goes extinct on the island or life is perpetual in a cycle of recurring generations.

Analysis and Requirements

You will be able to choose quite freely the life parameters, but it is recommended that in **this project their values be always be kept between 1 and 4.** The simulation process requires the following steps.

- An initial generation is specified by an arbitrary selection of the living squares. In this project the user is allowed to choose the living fields directly on a GUI that represents the island. Offering a random selection of the living squares by the computer is optional.
- The program determines the subsequent generations according the rules described above. Note that given a generation the new state of life must be determined for all the squares before any change is executed.
- All the generations created in the process must be stored in a data structure.
- The game terminates when either
  (i) Life is extinct (no living square exists) or
  (ii) A previous generation returns, that is life becomes cyclic on the island or
  (iii) The user terminates the procedure direct or by closing the GUI.

Apart from small worlds like the 2 x 2 or 3 x 3 grids, various initial life distributions may lead to a huge number of subsequent generations and it is usually impossible to predict the outcome of the game.
For this project you will write a program that implements the Game of Life in a simplified version such that making the computer to create the generation one after each other is controlled manually by the user. Your program

1. provides a GUI that represents the game board that is the view of the island where the subsequent generations are displayed
2. provides an independent non-GUI class which contains the logic of the game, its code determines the next generation according to the game rules, stores the whole sequence of consecutive generations, collects output data, signals if the game ended for extinction or for a recurring cycle of generations.

Input supplied by the user.

- Life color
- Dimensions of the world
- Life parameters
- Initial generation (an alternate random selection is optional)
- Various clicks on the control buttons

Input data such as dimension numbers, life parameters and a color choice for living and dead squares may be directly assigned to variables in the code, or they can be solicited in another GUI (a recommendable option). The user shall set up an initial generation on the island GUI.

Output

- The series of subsequent generations displayed on the GUI
- A final message showing the number of generations, cycle length or extinction. The message is displayed in the GUI.

Design

This design is confined to the simplest choices out of the many option as described above.

`1. Workshop (a non-GUI class).` This class represents the logic of the game without display. A generation (the grid of squares) is represented by a two-dimensional boolean array, an entry of which is true if and only if the corresponding square is alive. Details are in the UML below

Class Workshop	
DATA FIELDS (package access is allowed)	

`message: String`
`finished: boolean`
`mirror: boolean[][]`
 	
`allFalse: boolean[][]`
	Stores output message of game outcome
	Assigned true when game ends
	Represents the current generation, stores true/false for living/dead grid squares
	`final;` **named constant** for the array of all false entries; applies for checking extinct generation

`history: ArrayList<boolean[][]>`	to store the subsequent generations (mirrors)

`rows: int`
`columns: int`
	grid dimensions

`minLifeToLife: int`
`maxLifeToLife: int`
`minDeadToLife: int`
`maxDeadToLife: int`
	Life parameters
	(initialized by direct assignment, or by a modified constructor, or by setters)

METHODS	
Constructor	
`+Workshop(r: int, col: int)`	Initializes the grid dimensions Instantiates mirror Instantiates array allFalse
Instance methods	
Setters and getters	if needed
`+resetMirror(): void`	Resets all entries in mirror to false (an easy way: instantiate default mirror again
`+resetHistory( ) : void`	Clears all elements from history (calls the clear( ) method)

`-isAlive(k:int,j:int):boolean`	private helper method; returns false if k or j is not a valid index for mirror, otherwise returns mirror[k][j]
`-neighborCount(k:int,j:int):int`	private helper; Counts and returns the number of living neighbors of mirror[k][j] ; Hint: calls isAlive for the neighbor indices; Note: an array entry is not a neighbor of itself;
`-equals(arrA:boolean[][], arrB:boolean[][]):boolean`	private helper; checks if arrays arrA and arrB have the same boolean entries;
`-listPositionOf( target:boolean[][]):int`	private helper; determines and returns the position number of target in history; sets up a counter with initial value 1, iterates thru the list and updates counter if target is not the current entry and iteration continues; otherwise returns the counter value; returns 0 if target never found; Note: 0 is not a position number
`+nextGeneration():void`	Called from the GUI class when GUI finished to fill up mirror that is, a generation displayed on GUI is available for the construction of the next generation; adds mirror to history; assigns finished false; declares and instantiates a local 2-dim array **next** to store the next generation; iterates over mirror, and using neighborCount and the life parameter rules determines the value of next[k][j] from mirror[k][j]; calls equals to check if next is allFalse; if so, assigns **message** the ‘extinct message’ (see the supplement literals below); assigns mirror next; assigns finished true; method returns; otherwise calls listPositionOf for next as target (save the return value in a local variable position) if position positive, assigns **message** the ‘cycle message’ (see the supplement literals below); assigns mirror next; assigns finished true; method returns; otherwise assigns mirror next; assigns message “Go on!!”

```
String literal supplements
'extinct' message:
 "After "+history.size()+ " generations life is extinct in Island";
'cycle' message:
"After " + (position-1) + " generations life is cyclic of length "+
                              (history.size()-position+1);
```

2. Island (GUI class). This class represents the view of the island of living things in the form of a rectangular grid. Each square in the grid is a JButton object which shows the grid coordinates. A click of a button (when it is enabled) changes its color from light gray (the dead color) to yellow (the life color) or back. The user can set up the initial formation of living squares by button clicks.
Additional buttons on the south panel **mark, next** and **reset** control the game.

- At the start of the game all buttons on the grid are disabled. The **mark** button enables the grid and the user can choose the living squares. The grid buttons also make the Workshop **mirror** change the Boolean entries according to the colors of the button. Clicking mark also enables **next** and **reset**
- When the initial generation is ready, the **next** button makes the Workshop process mirror and build up the next generation which is right away displayed on the grid replacing the previous generation.
- Each generation as mirrored in Workshop is saved in the ArrayList
- Continued clicking of next keeps displaying the new generations until life gets extinct or a cycle is detected.
- The corresponding output message is displayed in the text field posted on the north panel.
- The reset button resets the board to an all dead grid, disables next and enables mark. The user can set up a different initial formation an observe the generations of another history
- If the user wants to test the game with different dimensions and life parameters, the program must be terminated and the new input assigned. The current design does not support the smooth change of these input, but the program can easily be maintained to work with enhanced options.

Class Island **extends JFrame implements ActionListener**	
DATA FIELDS (package access is allowed)	
`shop : Workshop`	Workshop is aggregated to Island to support class communication
`population: JButton[][]`	The grid buttons for living and dead squares
`additional JButtons`	as shown in Figure 1
`northField: JTextField`	shows messages of report on the game Note: GUI components referenced in several methods must be declared as fields
METHODS	
Constructor	
`+Island(rows:int, columns:int, shop: Workshop)`	super call sets the title Initializes shop; Sets the frame parameters; Instantiates population to dimensions rows and columns; Iterates over population and instantiates each population[k][j] entry, the coordinates k and j are used as Strings (hint : valueOf( )) to display them on the buttons; **this** registered to the array entries; background is set to light gray for each button
Instance methods	
`-enableBoard(flag:boolean): void`	private helper; Iterates over population and calls setEnabled with flag for all elements
`-resetBoard():void`	private helper; Iterates over population and sets background to light gray for all squares
`+buildWindow( ): void`	builds the GUI as shown on Figure 1. Note: east and west regions are not used
`-displayGeneration(mirror: boolean[][]):void`	private helper; Iterates over mirror and sets the background of population[k][j] to yellow if mirror[k][j], and to light gray otherwise

`+actionPerformed(event:ActionEvent): void`	Executes when one of the grid button has been clicked; Flips the background color of the source button Hint: use a local variable JButton source to store the return value of the getSource( ) method (do not forget the type cast)
`inner class ButtonListener implements ActionListener`	This class is defined to provide a listener for the three control buttons mark, next and reset. In the buildWindow ( ) method declare and instantiate a ButtonListener reference and register that reference to the three buttons

`+actionPerformed(event:ActionEvent): void`	This method is defined in the inner class;
	Executes when one of the three control buttons is clicked;
	if the source is **mark**
	enableBoard is called to enable the board;
	button **next** is enabled;
	shop calls resetHistory;
	otherwise if next is the source
	board is disabled;
	mark is disabled;
	shop calls resetMirror;
	iterates over **population** and mirror[k][j] is assigned true every time population[k][j] background is not light gray;
	shop calls nextGeneration;
	northField sets the text the **message** of shop;
	calls displayGeneration, parameter is mirror of shop;
	checks if **finished** (a variable of shop), if so next disables itself;
	otherwise if reset is the source
	mark is enabled;
	shop calls resetHistory;
	northField vacates;
	resetBoard is called

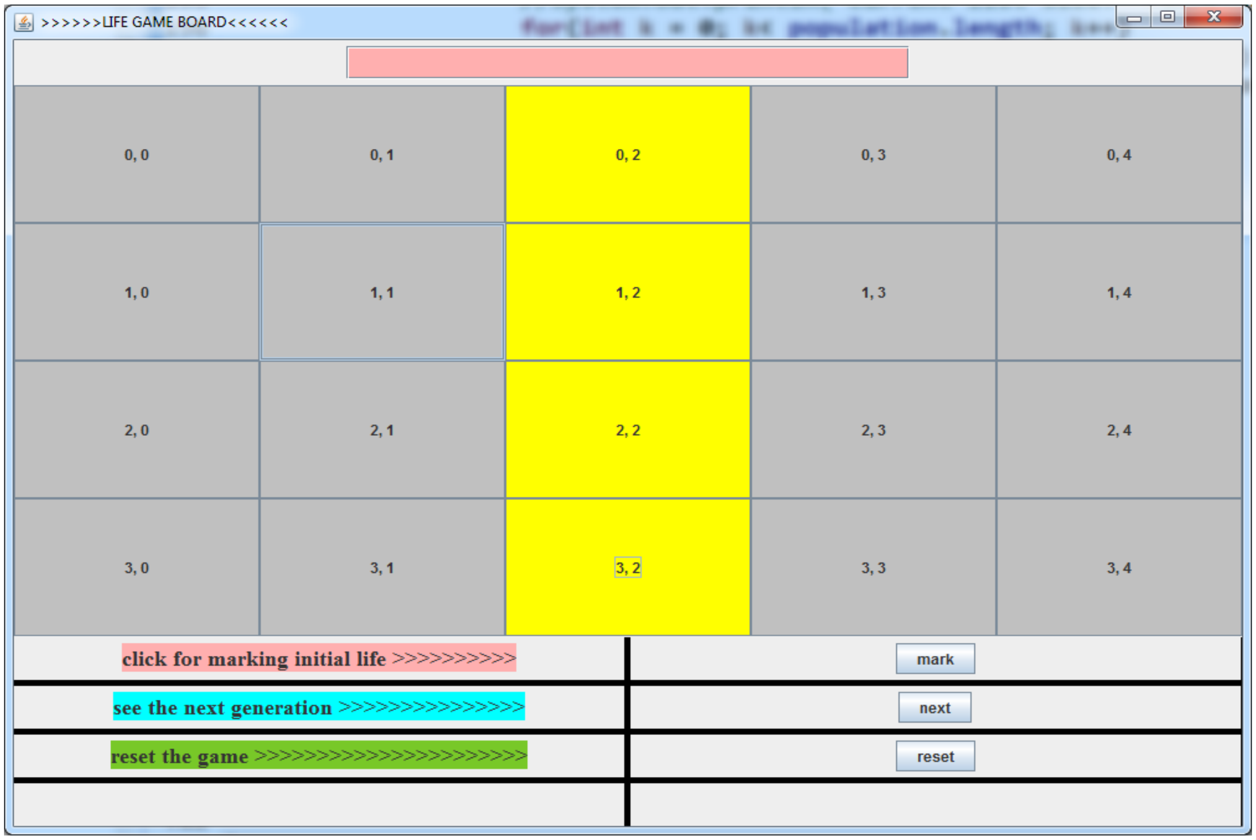
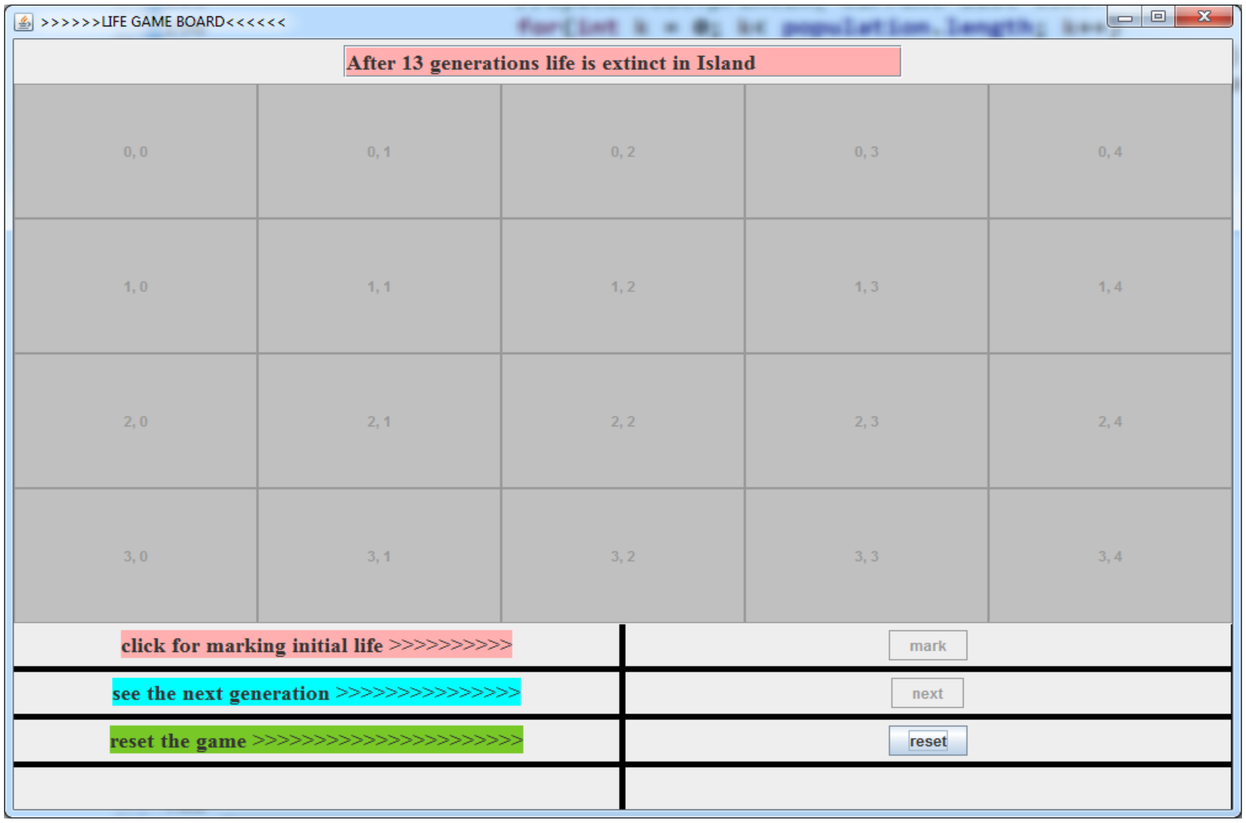

>>>>>>LIFE GAME BOARD<<<<<<
0, 0
0, 1
0, 2
0, 3
0, 4
1, 0
1, 1
1, 2
1, 3
1, 4
2, 0
2, 1
2, 2
2, 3
2, 4
3, 0
3, 1
3, 2
3, 3
3, 4
click for marking initial life >>>>>>>>>>
mark
see the next generation >>>>>>>>>>>>>>>
next
reset the game >>>>>>>>>>>>>>>>>>>>>>
reset

Figure 1: initial generation


>>>>>>LIFE GAME BOARD<<<<<<
0, 0
0, 1
0, 2
0, 3
0, 4
1, 0
1, 1
1, 2
1, 3
1, 4
2, 0
2, 1
2, 2
2, 3
2, 4
3, 0
3, 1
3, 2
3, 3
3, 4
click for marking initial life >>>>>>>>>>
mark
see the next generation >>>>>>>>>>>>>>>
next
reset the game >>>>>>>>>>>>>>>>>>>>>>
reset

After 13 generations life is extinct in Island

Figure 2: The result of Figure 1

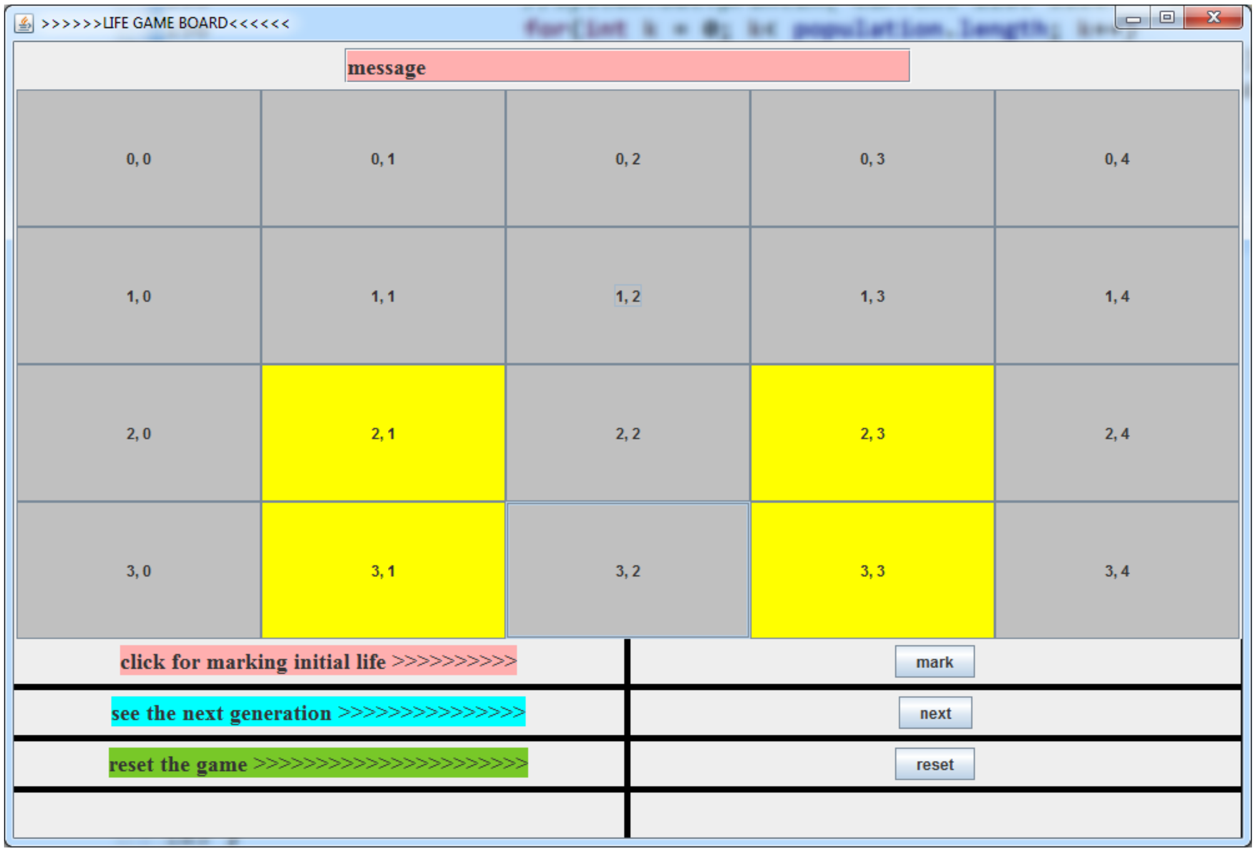
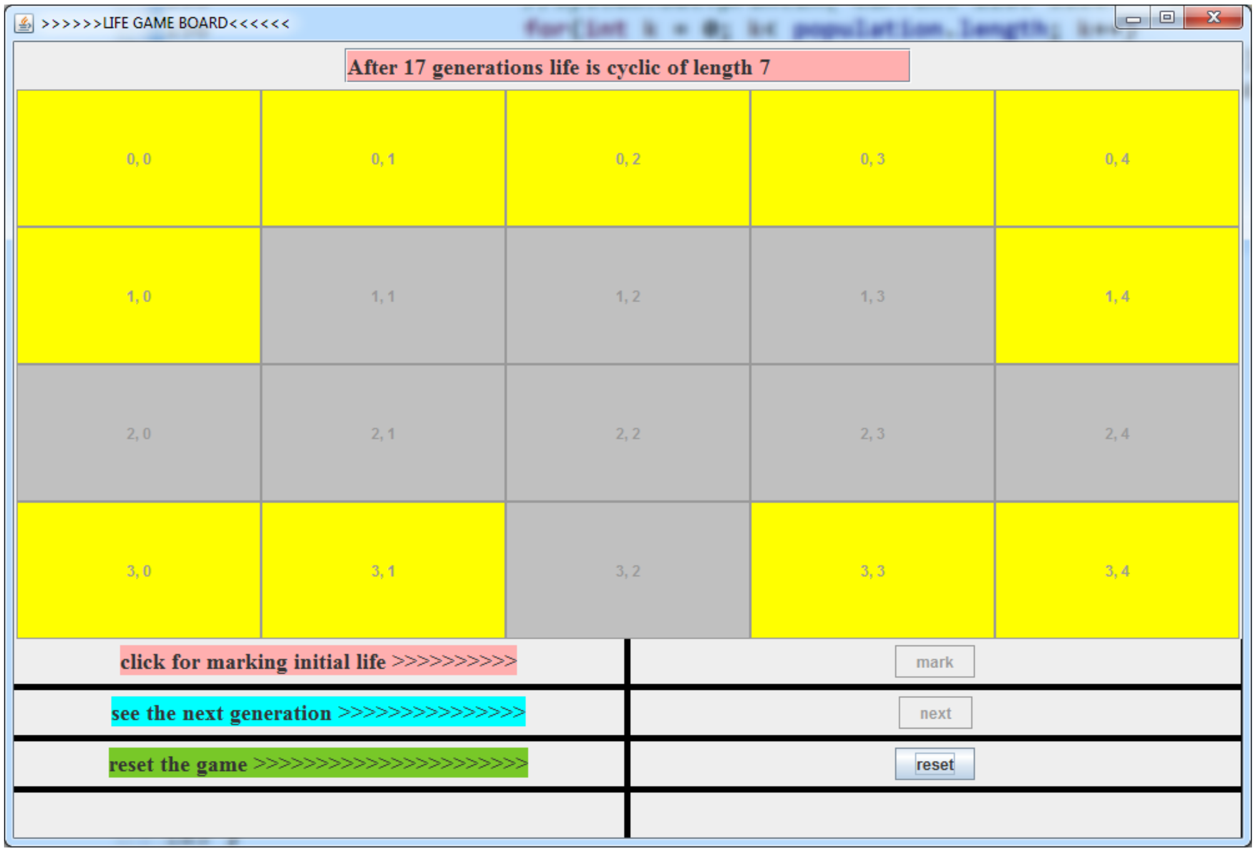
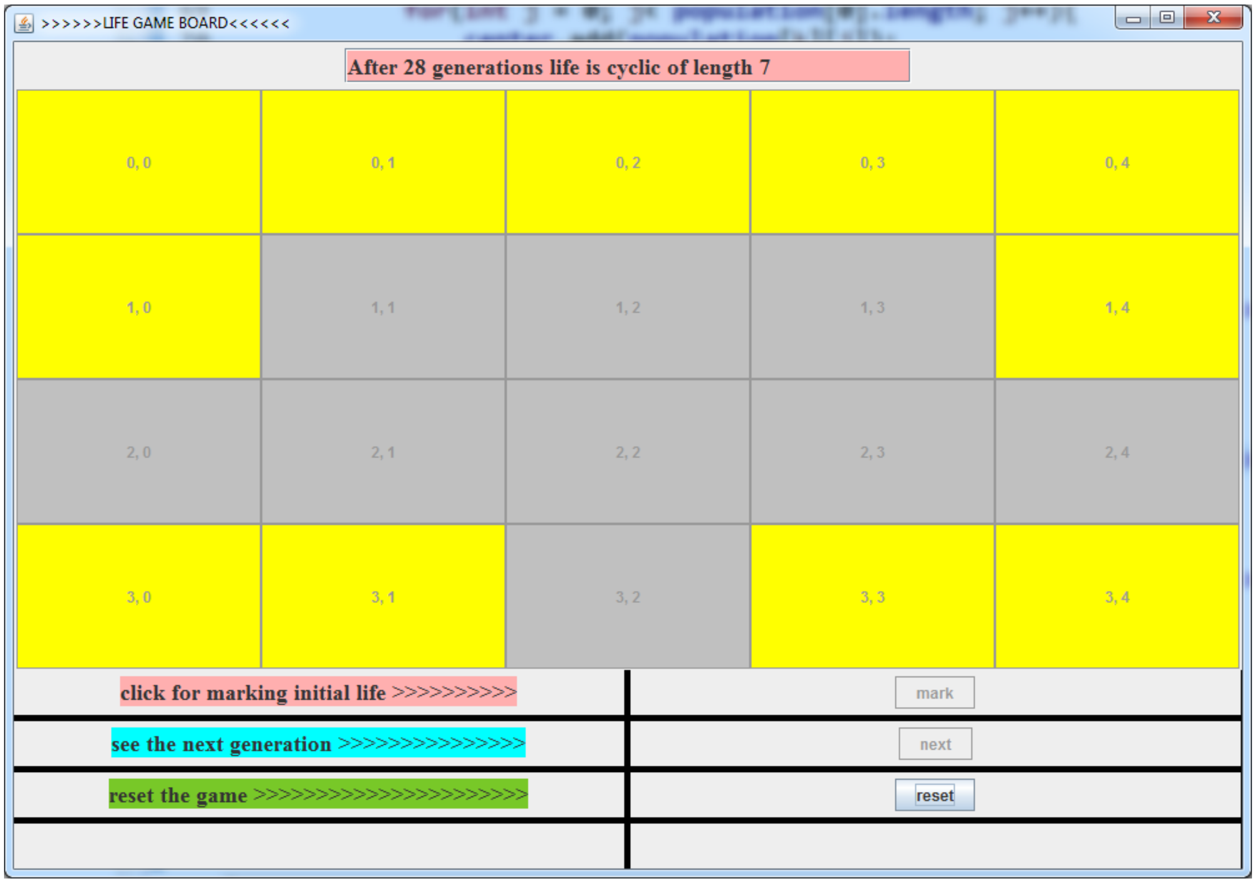

>>>>>>LIFE GAME BOARD<<<<<<
message
0, 0
0, 1
0, 2
0, 3
0, 4
1, 0
1, 1
1, 2
1, 3
1, 4
2, 0
2, 1
2, 2
2, 3
2, 4
3, 0
3, 1
3, 2
3, 3
3, 4
click for marking initial life >>>>>>>>>>
mark
see the next generation >>>>>>>>>>>>>>>
next
reset the game >>>>>>>>>>>>>>>>>>>>>>
reset

Figure 3: initial generation


>>>>>>LIFE GAME BOARD<<<<<<
0, 0
0, 1
0, 2
0, 3
0, 4
1, 0
1, 1
1, 2
1, 3
1, 4
2, 0
2, 1
2, 2
2, 3
2, 4
3, 0
3, 1
3, 2
3, 3
3, 4
click for marking initial life >>>>>>>>>>
mark
see the next generation >>>>>>>>>>>>>>>
next
reset the game >>>>>>>>>>>>>>>>>>>>>>
reset

After 17 generations life is cyclic of length 7

Figure 4: The result of Figure 3



After 28 generations life is cyclic of length 7

click for marking initial life >>>>>>>>>>

see the next generation >>>>>>>>>>>>>>>

reset the game >>>>>>>>>>>>>>>>>>>>>>>

Puzzle: what was the initial generation for the result above?

Evaluation

Documentation and Style (20 points). Your program must conform with the Java Documentation and Style Requirements as set by the Computer Science Department. Emphasis will be placed on having the required banner and internal comments, indentation, and overall professional appearance. Comments must be clearly written with correct grammar and spelling.

Correctness (80 points).

1. Workshop private methods implemented correct	18
2. Workshop nextGeneration implemented correct	10
3. Island private methods implemented correct	12
4. Island buildWindow implemented correct	10
5. Island actionPerformed implemented correct	15
6. Inner class actionPerformed implemented correct	15

Total…………………………………………100 points

Submit: Upload the zipped project folder at your lab site on Blackboard