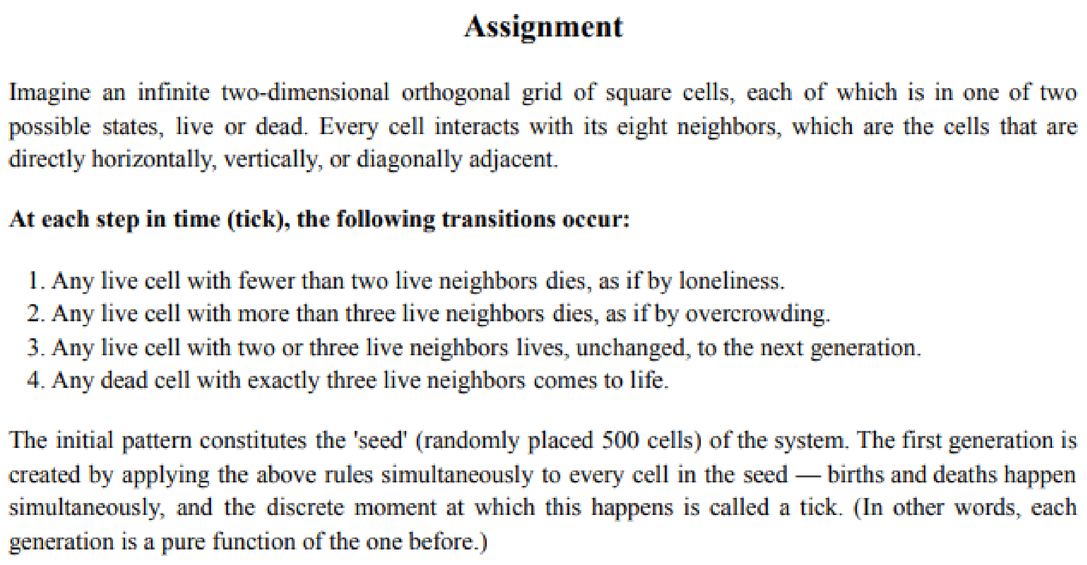
Q:No1)



## Assignment

Imagine an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent.

**At each step in time (tick), the following transitions occur:**

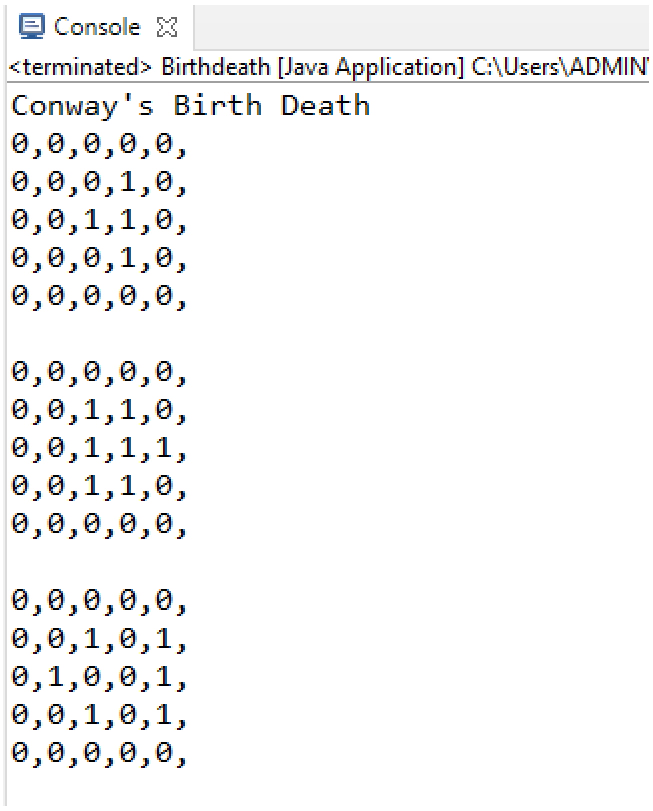
1. Any live cell with fewer than two live neighbors dies, as if by loneliness.
2. Any live cell with more than three live neighbors dies, as if by overcrowding.
3. Any live cell with two or three live neighbors lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbors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

The initial pattern constitutes the 'seed' (randomly placed 500 cells) 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 called a tick. (In other words, each generation is a pure function of the one before.)

Answer:

```
package Assignment;

public class Birthdeath {
      // The value representing a dead cell
    public final static int DEAD    = 0x00;

    // The value representing a live cell
    public final static int LIVE    = 0x01;


     public static void main(String[] args) {

           // test the birth and death  implementation
        Birthdeath bd = new Birthdeath ();
        bd.test(5);

     }


     //@nrIterations the number of times the board should be played


     private void test(int nrIterations) {
        // the starting playing board with life and dead cells
        int[][] board = {{DEAD, DEAD, DEAD, DEAD, DEAD},
                         {DEAD, DEAD, DEAD, LIVE, DEAD},
                         {DEAD, DEAD, LIVE, LIVE, DEAD},
                         {DEAD, DEAD, DEAD, LIVE, DEAD},
                         {DEAD, DEAD, DEAD, DEAD, DEAD}};

        System.out.println("Conway's Birth Death");
        printBoard(board);

        for (int i = 0 ; i < nrIterations ; i++) {
            System.out.println();
            board = getNextBoard(board);
            printBoard(board);
        }
    }


    private void printBoard(int[][] board) {

        for (int i = 0, e = board.length ; i < e ; i++) {

            for (int j = 0, f = board[i].length ; j < f ; j++) {
                System.out.print(Integer.toString(board[i][j]) +
",");
            }
            System.out.println();
        }
    }


    public int[][] getNextBoard(int[][] board) {

        // The board does not have any values so return the newly
created
        // playing field.
        if (board.length == 0 || board[0].length == 0) {
            throw new IllegalArgumentException("Board must have a
positive amount of rows and/or columns");
        }

        int nrRows = board.length;
        int nrCols = board[0].length;

        // temporary board to store new values
        int[][] buf = new int[nrRows][nrCols];

        for (int row = 0 ; row < nrRows ; row++) {
            for (int col = 0 ; col < nrCols ; col++) {
                buf[row][col] = getNewCellState(board[row][col],
getLiveNeighbours(row, col, board));
            }
        }
        return buf;
    }


    private int getNewCellState(int curState, int liveNeighbours) {

        int newState = curState;

        switch (curState) {
        case LIVE:

            // Any live cell with fewer than two
            // live neighbours dies
            if (liveNeighbours < 2) {
                newState = DEAD;
            }

            // Any live cell with two or three live
            // neighbours lives on to the next generation.
            if (liveNeighbours == 2 || liveNeighbours == 3) {
                newState = LIVE;
            }

            // Any live cell with more than three live neighbours
            // dies, as if by overcrowding.
            if (liveNeighbours > 3) {
                newState = DEAD;
            }
            break;

        case DEAD:
            // Any dead cell with exactly three live neighbours
becomes a
            // live cell, as if by reproduction.
            if (liveNeighbours == 3) {
                newState = LIVE;
            }
            break;

        default:
            throw new IllegalArgumentException("State of cell must be
either LIVE or DEAD");
        }
        return newState;
    }




    private int getLiveNeighbours(int cellRow, int cellCol, int[][]
board) {

        int liveNeighbours = 0;
        int rowEnd = Math.min(board.length , cellRow + 2);
        int colEnd = Math.min(board[0].length, cellCol + 2);

        for (int row = Math.max(0, cellRow - 1) ; row < rowEnd ;
row++) {

            for (int col = Math.max(0, cellCol - 1) ; col < colEnd ;
col++) {

                // make sure to exclude the cell itself from
calculation
                if ((row != cellRow || col != cellCol) &&
board[row][col] == LIVE) {
                    liveNeighbours++;
                }
            }
        }
        return liveNeighbours;
    }


}
```

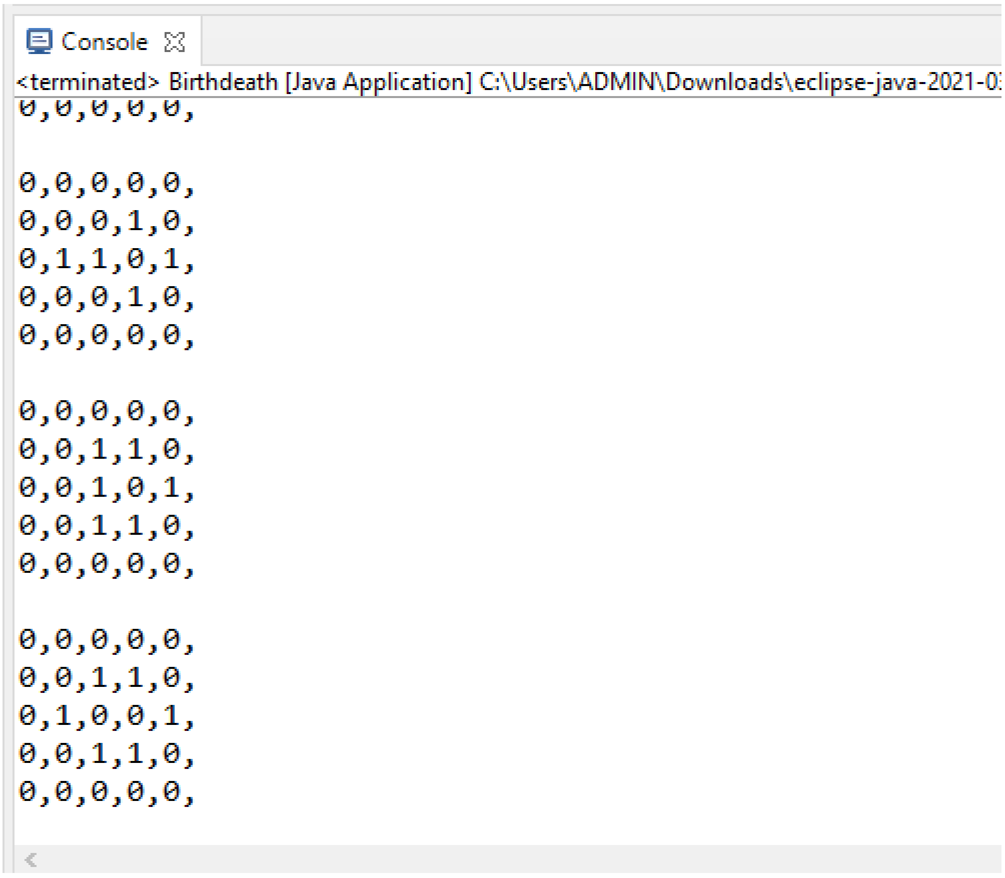
OUTPUT:



Console

<terminated> Birthdeath [Java Application] C:\Users\ADMIN\

```
Conway's Birth Death
0,0,0,0,0,
0,0,0,1,0,
0,0,1,1,0,
0,0,0,1,0,
0,0,0,0,0,

0,0,0,0,0,
0,0,1,1,0,
0,0,1,1,1,
0,0,1,1,0,
0,0,0,0,0,

0,0,0,0,0,
0,0,1,0,1,
0,1,0,0,1,
0,0,1,0,1,
0,0,0,0,0,
```

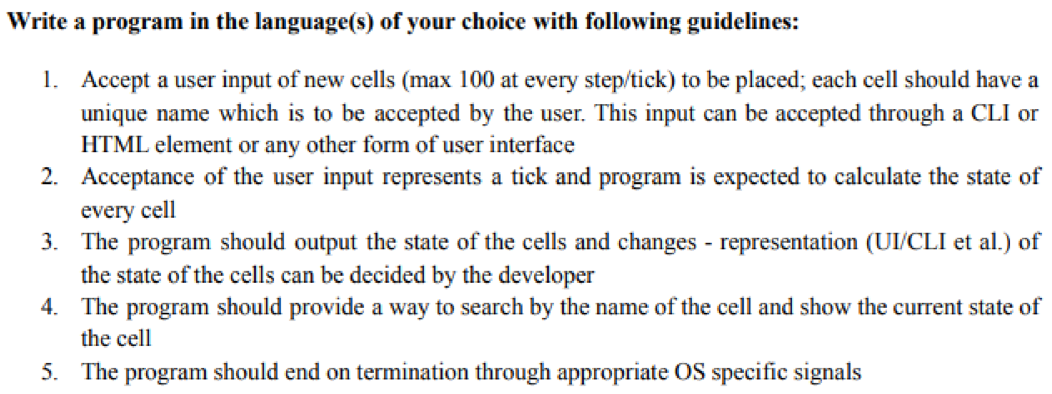


Console

<terminated> Birthdeath [Java Application] C:\Users\ADMIN\Downloads\eclipse-java-2021-0

```
0,0,0,0,0,

0,0,0,0,0,
0,0,0,1,0,
0,1,1,0,1,
0,0,0,1,0,
0,0,0,0,0,

0,0,0,0,0,
0,0,1,1,0,
0,0,1,0,1,
0,0,1,1,0,
0,0,0,0,0,

0,0,0,0,0,
0,0,1,1,0,
0,1,0,0,1,
0,0,1,1,0,
0,0,0,0,0,
```

Q:No2)



**Write a program in the language(s) of your choice with following guidelines:**

1. Accept a user input of new cells (max 100 at every step/tick) to be placed; each cell should have a unique name which is to be accepted by the user. This input can be accepted through a CLI or HTML element or any other form of user interface
2. Acceptance of the user input represents a tick and program is expected to calculate the state of every cell
3. The program should output the state of the cells and changes - representation (UI/CLI et al.) of the state of the cells can be decided by the developer
4. The program should provide a way to search by the name of the cell and show the current state of the cell
5. The program should end on termination through appropriate OS specific signals

Answer:

```
package Assignment;

import java.beans.PropertyVetoException;

public class BusinessObject extends ConstrainedObject {
       private int numericValue;
       public void setNumericValue(int newNumericValue) throws 
PropertyVetoException {
         // validate proposed value
         validate("numericValue", numericValue, newNumericValue);
         // the new value is approved, since no exceptions were 
thrown
         numericValue = newNumericValue;
       }
       public int getNumericValue() {
         return numericValue;
       }
     }
```

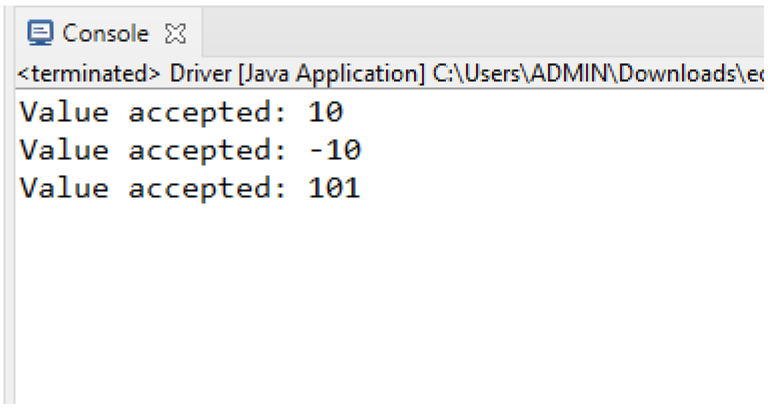
```java
package Assignment;

import java.beans.*;
public class BusinessObjectBeanInfo extends SimpleBeanInfo {
  Class targetClass = BusinessObject.class;


  public static final String MAX_VALUE = "maxValue";
  public static final String MIN_VALUE = "minValue";
  public PropertyDescriptor[] getPropertyDescriptors() {
    try  {
      PropertyDescriptor numericValue = new
PropertyDescriptor("numericValue", targetClass, "getNumericValue",
"setNumericValue");
      numericValue.setValue(Validator.MAX_VALUE, new ~~Integer~~(100));
      numericValue.setValue(Validator.MIN_VALUE, new ~~Integer~~(-100));
      PropertyDescriptor[] pds = new PropertyDescriptor[]
{numericValue};
      return pds;
    } catch(IntrospectionException ex) {
      ex.printStackTrace();
      return null;
    }
  }
}
```

```java
package Assignment;
import java.beans.*;

import javax.xml.validation.Validator;

public class ConstrainedObject {
       private VetoableChangeSupport vetoableSupport = new
VetoableChangeSupport(this);
       /**
        * Creates a new object with generic property validator
        */
      // public ConstrainedObject() {
        // vetoableSupport.addVetoableChangeListener(new
Validator());
       //}
       /**
        * This method will be used by subclasses to validate a new
value in
```

```
         * the constrained properties of the int type. It can be
easily overloaded
         * to deal with other primitive types as well.
         * @param propertyName The programmatic name of the property
that is about to change.
         * @param oldValue The old value of the property.
         * @param newValue The new value of the property.
         */
        protected void validate(String propertyName, int oldValue,
int newValue) throws PropertyVetoException {
          vetoableSupport.fireVetoableChange(propertyName, new
~~Integer~~(oldValue), new ~~Integer~~(newValue));
        }
      }


package Assignment;

import java.beans.*;


public class Validator implements VetoableChangeListener {
        public static final String MAX_VALUE = "maxValue";
        public static final String MIN_VALUE = "minValue";
        /**
         * The only method required by {@link
VetoableChangeListener} interface.
         */
        public void vetoableChange(PropertyChangeEvent evt) throws
PropertyVetoException {
          try {
            // here we hope to retrieve explicitly defined
additional information
            // about the object-source of the constrained property
            BeanInfo info =
Introspector.getBeanInfo(evt.getSource().getClass());
            // find a property descriptor by given property name
            PropertyDescriptor descriptor =
getDescriptor(evt.getPropertyName(), info);
            Integer max = (Integer) descriptor.getValue(MAX_VALUE);
            // check if new value is greater than allowed
            if( max != null && max.compareTo((Integer)
evt.getNewValue()) < 0 ) {
              // complain!
              throw new PropertyVetoException("Value " +
evt.getNewValue() + " is greater than maximum allowed " + max, evt);
            }
            Integer min = (Integer) descriptor.getValue(MIN_VALUE);
            // check if new value is less than allowed
            if( min != null && min.compareTo((Integer)
evt.getNewValue()) > 0 ) {
              // complain!
              throw new PropertyVetoException("Value " +
evt.getNewValue() + " is less than minimum allowed " + min, evt);
            }
          } catch (IntrospectionException ex) {
            ex.printStackTrace();
          }
        }
        /**
         * This utility method tries to fetch a PropertyDescriptor
from BeanInfo object by given property
         * name.
         * @param name the programmatic name of the property
         * @param info the bean info object that will be searched
for the property descriptor.
         * @throws IllegalArgumentException if a property with given
name does not exist in the given
         * BeanInfo object.
         */
        private PropertyDescriptor getDescriptor(String name,
BeanInfo info) throws IllegalArgumentException {
          PropertyDescriptor[] pds = info.getPropertyDescriptors();
          for( int i=0; i<pds.length; i++) {
            if( pds[i].getName().equals(name) ) {
              return pds[i];
            }
          }
          throw new IllegalArgumentException("Property " + name + "
not found.");
        }
      }
```

```
package Assignment;
/**
 * A simple application that tries to cause data-validation
exception
 * condition.
 */
public class Driver {
        public static void main(String[] args) {
          try {
            BusinessObject source = new BusinessObject();
            source.setNumericValue(10);
            System.out.println("Value accepted: " +
source.getNumericValue());
            source.setNumericValue(-10);
            System.out.println("Value accepted: " +
source.getNumericValue());
            source.setNumericValue(101);
            // we should never reach this line
            System.out.println("Value accepted: " +
source.getNumericValue());
          } catch (Exception ex) {
            ex.printStackTrace();
          }
        }
      }
```

OUTPUT:



```
<terminated> Driver [Java Application] C:\Users\ADMIN\Downloads\e
Value accepted: 10
Value accepted: -10
Value accepted: 101
```