In this assignment you will learn how to use objects and arrays by exploring a very simple model of computation.

A cellular automaton consists of a line of cells, each of which is either “alive” or “dead”. The automaton evolves in a series of steps we call “generations”. In each generation, every cell is updated according to a definite rule. The rule tells us the state of the cell in the next generation according to the current state of the cell and the state of its immediate left and right neighbors.

The cellular automaton you will be implementing has the following rule: if the cell is alive, it will be alive in the next generation unless both of its neighbors are also alive (it is “overcrowded”); if the cell and both of its neighbors are dead, the cell will be dead in the next generation; if the cell is dead and one or both of its neighbors are alive, it will be alive in the next generation (it is “reborn”).

The diagram below demonstrates each of the eight cases in the cellular automaton. The top row represents the state of the cell and its left and right neighbors in the current generation. The bottom row represents the state of the cell in the next generation. We use a black square to represent a live cell and a white square to represent a dead cell:

```
  T T T T T T T
```

You will write a class named `Automaton` representing a cellular automaton with 47 cells. In your class, you should represent the line of cells as a `boolean` array—live cells will have the value `true` and dead cells will have the value `false`. The line of cells should be “circular”: the leftmost cell will have the rightmost cell as its left neighbor and the rightmost cell will have the leftmost cell as its right neighbor. Your class will have three methods: `generation`, `rule` and `toString`.

- **generation** is a public method. `generation` updates every cell according to the rule. It takes no parameters and has return type `void`.

- **rule** is a private method. It takes three `boolean` parameters: the value of the cell, the value of the cell’s left neighbor and the value of the cell’s right neighbor. It returns a `boolean` value indicating whether the cell will be alive in the next generation.

- **toString** is a public method. It takes no parameters and returns a `String` indicating the current state of the automaton. Each cell should be represented by an ‘X’ if it is alive and a ‘ ’ (space) if it is dead. The string should end with a number representing the number of generations that have elapsed in this automaton.
Your class should have one constructor that takes a boolean parameter. If the parameter is true, the cell array should be initialized to random values, using the nextBoolean method of java.util.Random. If the parameter is false, the cell array should be initialized with exactly one live cell in the center (position 23).

In your main method, write a program that creates an instance of the Automaton class. The program should display the state of the automaton using the toString method and wait for the user to press Return. When the user presses Return, the program should call generation and display the new state of the automaton. The program should loop, allowing the user to press Return and display succeeding generations of the automaton, until the user types a “q” (for quit). Figure 1 shows some sample output from the program.

To read the user input, you should use the Input class. You may download the class from the course website or copy it from the course directory on CUNIX: ~cs1007/util/Input.class. The Input class provides a single static method named readQuit which will return true when the user types a “q” and false on any other input. Your program should continue until readQuit returns true.

do {
   ...
} while( !Input.readQuit() ) ;

Your main method should check for a command-line argument “--random”. If --random is specified, the automaton should be initialized with random cell values. If --random is not specified, the default initialization values (one live cell) should be used.

Your generation method will need at least two boolean arrays: one for the new generation and one for the previous generation. One should be a field in your class and the other should be a local variable in generation. At the beginning of the method, define an array variable and create a new array. Use the field array to read the values from the previous generation and write the values for the new generation in the local array. At the end of generation assign the local array to the field. (Note: you do not need to do an item-by-item copy, assigning the reference does the job.)

Submit Automaton.java and a README file briefly describing the design of your program.

Extra Credit 1 (2 points). Creating a new array in every generation is slow and wastes memory. A faster solution is to have an array buffer—a spare array on hand that we can reuse instead of tossing out old arrays in every step. Redefine your cell array as a two-dimensional array. Each element in the “outer” array should be a boolean array of 47 cells. Define the length of the outer array using a constant NUM_GENS:

private static final int NUM_GENS = 2 ;

Treat the outer array as a circular buffer of arrays. As you compute a new generation, place it in the next entry in the array buffer. The array buffer will contain a “history” of the last NUM_GENS generations of the automaton. Make no assumptions about the value of NUM_GENS inside your definition of generation.

Extra Credit 2 (2 points). Add a command-line argument “--init=n” to your main method. If “--init=n” is specified, the integer n will be used to initialize the automaton
Figure 1: Typical execution of the Automaton program.
as follows: the least-significant 47 bits of \( n \) will represent the initial state of each cell in the automaton, with a live cell represented by a 1 and a dead cell represented by a 0. If \( n \) is less than 47 bits, pad it with zeroes on the left.

You will need to define a new constructor that takes a single \texttt{long} parameter and uses it to initialize the cell array. You may choose whether to represent \( n \) as decimal or hex. You can use the Java library method \texttt{Long.parseLong} to convert a \texttt{String} to a \texttt{long} value. The method is defined as:

\[
\text{public static long parseLong(String s, int radix) ;}
\]

\texttt{radix} is the base of the number system used to encode \texttt{s}.