Cellular automata may be one-dimensional as well as two-dimensional. As with two-dimensional automata, the neighborhood of a cell is defined to be cells nearby. But since there is only one dimension, the neighbors of a cell are only to the left and right of it. For a neighborhood consisting of one cell on each side of cell, there are 8 possible patterns for each cell and its neighbors. The set of rules for such a one-dimensional automaton usually is given as an enumeration of the 8 possible patterns and the results for the center cell (called Wolfram notation). One particular interesting set of rules is given as follows:

<table>
<thead>
<tr>
<th>current pattern</th>
<th>111</th>
<th>110</th>
<th>101</th>
<th>100</th>
<th>011</th>
<th>010</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>new state for center cell</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

where 1 means the cell is alive and 0 means the cell is dead. For the cells on the end, we assume that the missing neighbor's value is 0.

For this assignment, you are given a compressed solution folder 1D-CellularAutomata.zip that is the same as the GameOfLifeInClass folder except that it creates a one-dimensional grid of Labels to represent the cells as shown below:

![1D Cellular Automata](image)

and has the grid Click and Clear button handlers already written.

**Assignment**

Download the compressed solution folder 1D-CellularAutomata.zip from the course webpage.

Complete the one-dimensional cellular automata program in the same manner as was done for the Game of Life in-class exercise. That is,

1. Write a method `ComputeGeneration` that computes the next generation using the rules given above and store it in a one-dimensional grid of strings. Note that you will need to add a variable for the backing grid of strings and add code to the `MakeGrid` method to create it. (Also note
that since there are only 2 neighbors and the rules are enumerated for each pattern, we do not need a method to count the live neighbors as was done for the Game of Life.)

2. Write a method `DisplayGeneration` that copies the backing grid strings to the grid of labels Text property to display the new generation.

3. Double-click on the Next button and write a handler that will loop for the number of generations entered by the user (textbox `numGenerations`) by calling `ComputeGeneration` and `DisplayGeneration` each iteration.

4. Add a delay after each `DisplayGeneration` using the `Thread.Sleep` method to delay the number of milliseconds (ms) entered by the user (textbox `updateTime`). Remember to add "using System.Threading;" at the beginning of the program file.

5. Finally, add code to keep track of the number of generations computed (integer variable `genCount`) and display it (Text property of label variable `generationCount`).

Try starting with a single asterisk in the middle of the grid. Experiment with your program and find at least one other starting grid state that generates an "interesting" sequence of grid states.

**What to submit**

For full credit, a compressed (zipped) solution folder containing a C# project must be submitted as an attachment to an email to Dr. Hwang (hwang@evansville.edu) no later than 4:30pm on the due date.

Also submit in your email, a starting grid pattern that generates an "interesting" sequence of grid states with a brief discussion of why you think it is interesting.

To create a compressed solution folder find the solution folder via Windows Explorer. Right-click on the solution folder, select Send To, then select Compressed (zipped) folder. This will created a compressed folder of the same name as the solution folder with extension .zip and an icon of a folder with a zipper.