Problem-Solving Task 2: Conway

Implement Conway's Game of Life in C#.

Introduction

The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

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 occur simultaneously, and the discrete moment at which this happens is sometimes called a tick. Each generation is a pure function of the preceding one. The rules continue to be applied repeatedly to create further generations.

https://youtu.be/CgOcEZinQ2I

The **Game of Life** shows how extraordinary complexity can come about from surprisingly simple rules. It is often used as an analogy of complex life.

Your task

You will complete a **C# console program** that will run the **Game of Life**.

An initial code template has been provided to you on Blackboard, which contains several empty methods that you will need to complete, as well as an empty `Main` method. Each method has been documented - your code must closely follow the documented requirements for each method.

For some of the methods, testing suites have been written so that you can check whether your code is correct. There are no testing suites written for the `Main` method. Run the testing suite by adding the following line into `Main`:

```
TestRunner.RunTests();
```

This will produce an output that checks whether your methods are correct, along with details about any exceptions that occur within your methods. When you are ready to complete the required content for the `Main` method, you should comment out this line of code.

Important information about your solution

What will the program do?

The program should start by creating a grid to represent the **Game of Life** according to the rules in the next section (The Game Grid).

Then, the program should repeatedly update the grid and display it on-screen. So that it does not run too fast, you should add a delay between each update. See the section on `Main` below for how to do this.

Here is an example of what the final product might look like:

https://youtu.be/uipCw8eAN2Q

The Game Grid

Unlike the original description of the **Game of Life** which uses an infinite grid, we will create a finite, rectangular grid using a 2D array of `bool` values, where `true` represents a living cell, and `false` represents a dead cell.

The dimensions of the grid should be easy to change, and should be governed by **named constants** in your program code. A reasonable starting point is 20x50 (20 rows by 50 columns).

The grid will initially be filled randomly with living or dead cells. Each cell will have a 50% chance of being alive.

When printed on-screen, dead cells will be represented by a `.` and living cells will be represented by a `#`.

Here is an example of a 30x15 grid displayed in a console:

```
....................##.....
..................###...##..
....................#.#...##
....................#.......#
....................#....###
.......#.............###...#
..###...#......##.......#....#
.##.#.........###....##..#...#
#...#.........#......##....##.
.###....#.....................
```

Methods to Complete

There are five methods to complete for this assignment, though you are free to add more methods yourself. You can complete the methods in any order, but you will not be able to complete the final `Main` method until the end. The order listed below is the **recommended order** that you complete the methods in. In the meantime, you should use your `Main` method as a place to test the other methods, until you are ready to replace it with its final functionality.

`MakeGrid`

```
static bool[,] MakeGrid(int rows, int columns) { ... }
```

- **Purpose**: Creates an array of boolean values with the specified number rows and columns. Each cell has a 50% chance of being filled with the value `true`. The rest should remain `false`.

- **Implementation tip:** For each cell in a new array, use a random number generator to provide a conditional expression that evaluates to `true` 50% of the time - use this to fill the cell with that conditional expression.

`DrawGrid`

```
static void DrawGrid(bool[,] grid) { ... }
```

- **Purpose**: Draw the given `grid` to the console using `#` and `.` characters.
- **Implementation tip:** You should know the difference between `Write` and `WriteLine` for this, so that you can correctly write a newline at the end of each row. You will also need to understand `for` loops for 2D array processing.

`CountNeighbours`

```
static int CountNeighbours(bool[,] grid, int row, int col) { ... }
```

- **Purpose**: Using the given `grid`, count and return the number of living neighbouring cells to the cell at `row, col`. The cell at `row, col` does not count as a neighbour.
- **Implementation tip**: It is possible to complete this method with *lots* of `if` statements, but it can also be completed with a combination of `if` statements and `for` loops. A common problem students run into with this method is trying to check for living cells in locations that do not exist! This usually happens when checking for the neighbours of a cell along the boundary of the grid. To avoid this you **need** to understand the `GetLength` method so that you can check whether a particular row or column exists before attempting to index it.

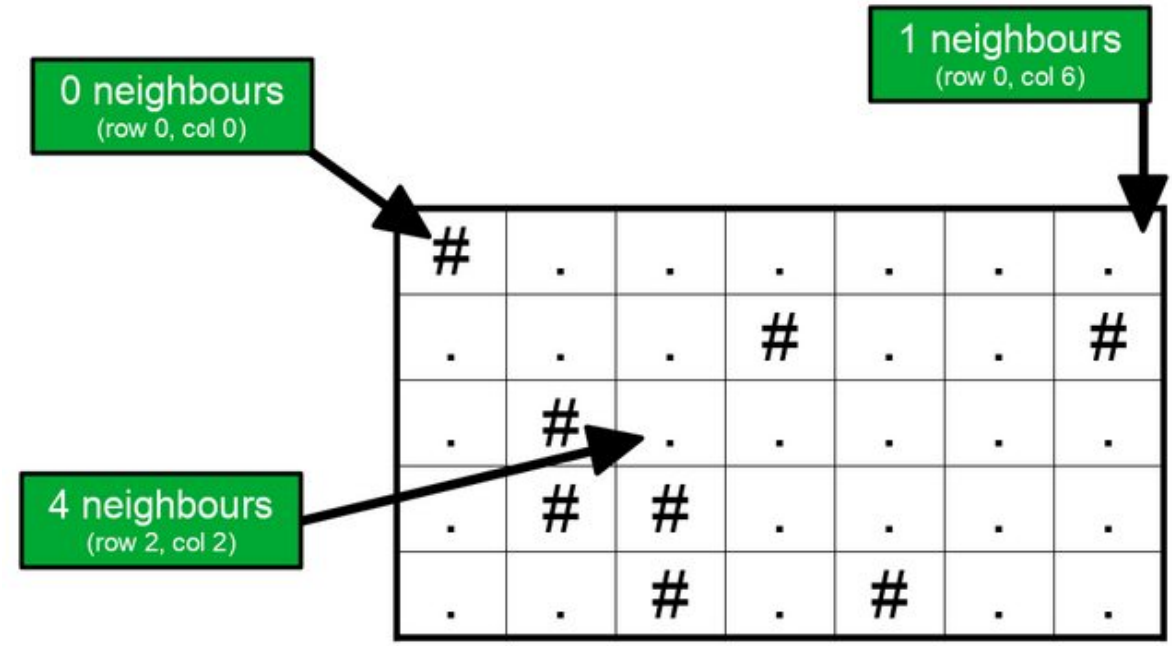
Usage example:

```
static void Main() {
    bool[,] grid = {
        { false, false, false, false },
        { false, true,  true,  false },
        { true,  true,  false, false },
        { false, false, false, false }
    };

    Console.WriteLine(CountNeighbours(grid, 0, 0));
    Console.WriteLine(CountNeighbours(grid, 1, 1));
    Console.WriteLine(CountNeighbours(grid, 2, 0));
}
```

```
1
3
2
```

Visual example:



`UpdateGrid`

```
static bool[,] UpdateGrid(bool[,] oldGrid) { ... }
```

- **Purpose**: Using the given grid, return a new grid that has been updated according to the rules of the Game of Life:
    1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
    2. Any live cell with two or three live neighbors lives on to the next generation.
    3. Any live cell with more than three live neighbors dies, as if by overpopulation.
    4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
- **Implementation tip**: You will need to create a *new* grid in this method which has the same dimensions as the old grid. You will need to fill this grid by copying cells from the old grid into it, but also acknowledge the rules of overpopulation, underpopulation and reproduction. You will need to use the `CountNeighbours` method to complete this.

`Main`

```
static void Main() { ... }
```

- **Purpose:** Runs the Game of Life by:
    1. Displaying a welcome message and pause until the user presses ENTER to start the simulation
    2. Creating a game grid (use `MakeGrid(...)`)
    3. Repeating the following
        1. Clear the console (you can use `Console.Clear()`)
        2. Draw the grid (use `DrawGrid(...)`)
        3. Update the grid (use `UpdateGrid(...)`)
        4. Wait a specified time (you can use `System.Threading.Thread.Sleep(UPDATE_TIME)`, where `UPDATE_TIME` is a constant integer such as `150`)

Marking specification

Functional requirements (7.5%)

Criteria	Marks possible	Marks awarded
MakeGrid works correctly	1	
UpdateGrid works correctly	2	
CountNeighbours works correctly	1.5	
DrawGrid works correctly	1	
Main works correctly	2	

Code quality (2.5%)

Criteria	Marks possible	Marks awarded
Base mark	2.5	2.5
Magic numbers are present in code	-1	
Code contains inconsistent or invalid indentation and curly brace placement	-1	
Some variable names are not meaningful	-1	
Code lacks comments	-1	