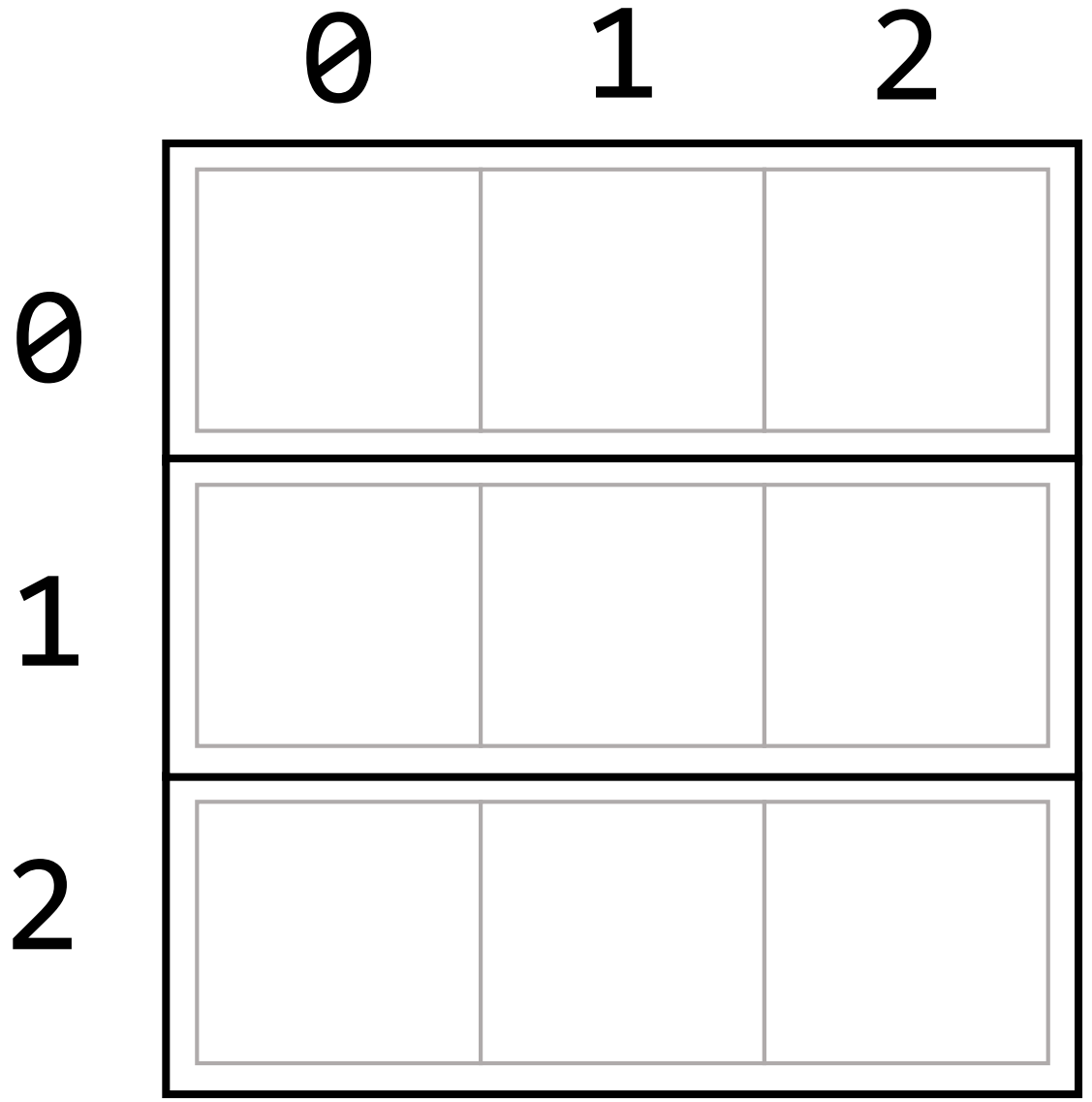
2D Arrays

Lecture 19

2D Arrays

- Easy to think of as a 2D grid
- Useful for storing data naturally represented in a table or grid
- For example: Picture Data
- Really an “array of arrays”



An “Array of Arrays”…

- The way we declare an array of some type is:

  `<type>[]`

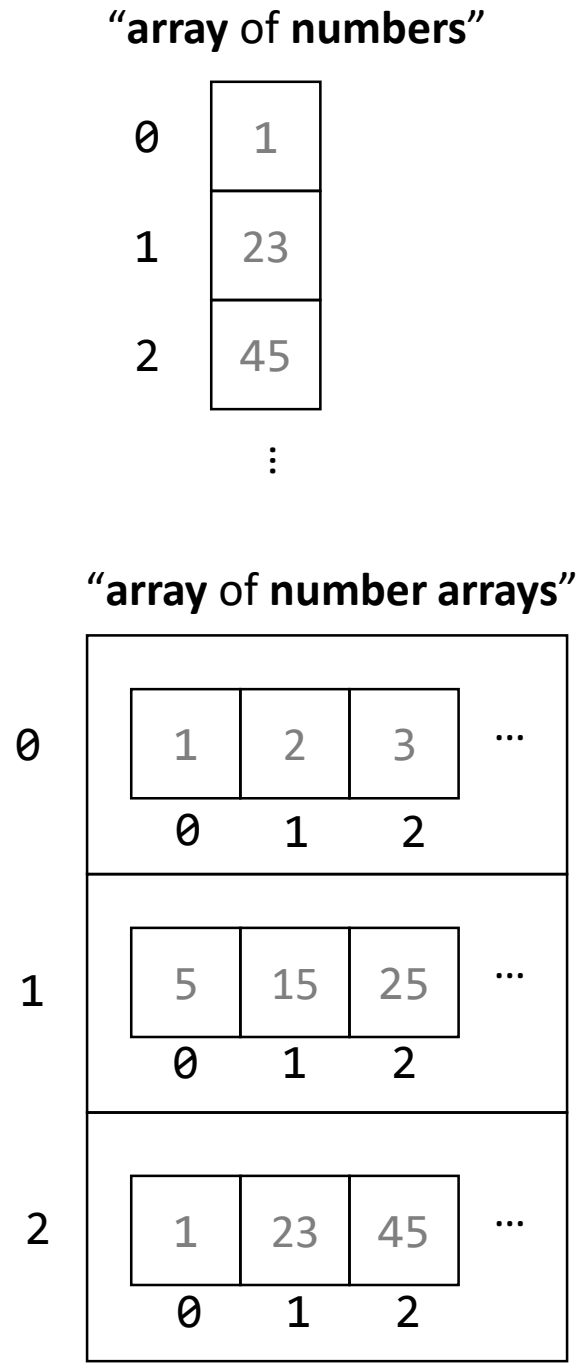
- For example, if we want an **“array of numbers”**
  - Each element’s **type** is **`number`**
  - So we’d declare:

  `number[]`

- What if we wanted an **“array of number arrays”**
  - Each element’s **type** is **`number[]`**
  - So we’d declare the type to be:

  `number[][]`

- You *can* keep doing this with arrays that are 3D, 4D, and so on…



2D Array Operations – Variable Declaration

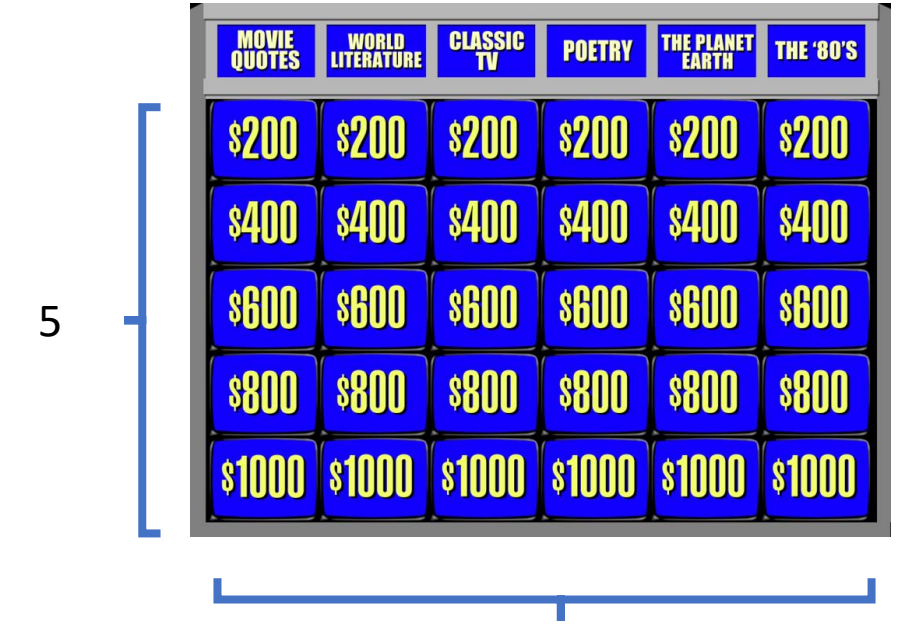
`let <name>: <type>[][]`

- For example:

```
let jeopardyBoard: number[][];
```

2D Array Operations – Initialization
Row-major Literals

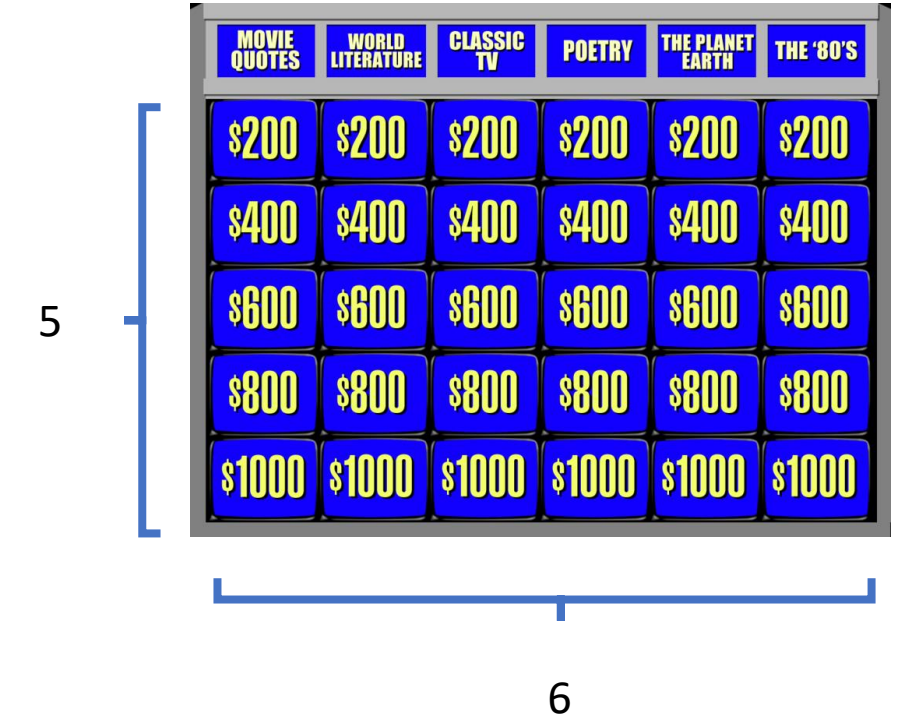
```
let jeopardyBoard: number[][] = [
    [  200,  200,  200,  200,  200,  200 ],
    [  400,  400,  400,  400,  400,  400 ],
    [  600,  600,  600,  600,  600,  600 ],
    [  800,  800,  800,  800,  800,  800 ],
    [ 1000, 1000, 1000, 1000, 1000, 1000 ]
];
```



- Arrays can be logically organized as "column-major" or "row-major" where "major" refers to the outer most array. In this example, we're initializing row-major.
- Row-major order means we're accessing elements via a first index of row and a second index of column. For example **`jeopardyBoard[0][2]`** is **`200`**.

2D Array Operations – Initialization – Column-major Literals

```
let jeopardyBoard: number[][] = [
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ],
    [ 200, 400, 600, 800, 1000 ]
];
```



- In this example, we're initializing column-major instead.
- Column-major order means we're accessing elements via a first index of column and a second index of row. For example **`jeopardyBoard[0][2]`** is **`600`**.

Row-major vs. Column-major Layouts

- **In COMP110, we'll use row-major based 2D arrays**

- For most sizes of problems you'll encounter *it doesn't matter,* as long as you are consistent and document your decisions.

- In upper-level classes you'll learn optimal layouts depend on both:
  1. How your programming language organizes 2D arrays in memory
  2. How your algorithms tend to iteratively access the arrays

2D Array Operations – Initializing with Code

- Let's implement the **2d-arrays-helpers.ts** function with the following signature:

```
<T> array2d(rows: number, cols: number, val: T): T[][]
```

- This function is given the # of rows and # of columns and its purpose is to initialize a 2D array of any type T where each element's initial value is the third argument.

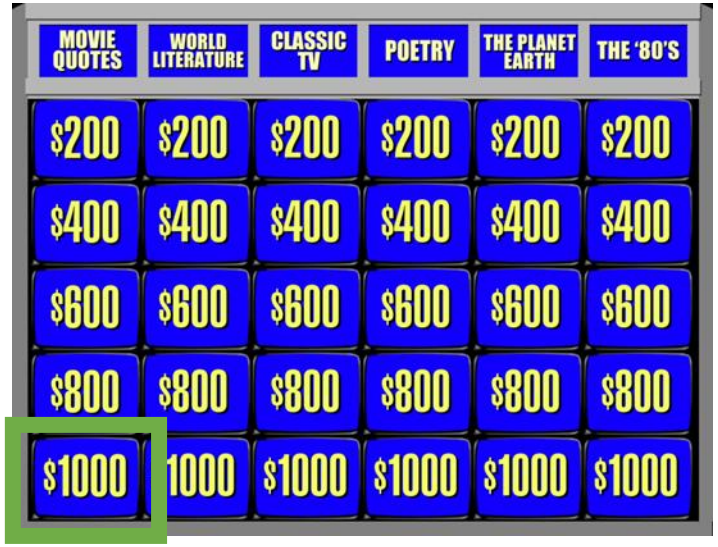
```ts
export let array2d = <T> (rows: number, cols: number, value: T): T[][] => {
    let a: T[][] = [];

    for (let row = 0; row < rows; row++) {
        // Initialize the next row as an empty array
        a[row] = [];
        for (let col = 0; col < cols; col++) {
            a[row][col] = value;
        }
    }

    return a;
};
```

2D Array Operations – Assigning to Inner Array

<name>[index$_r$][index$_c$] = <value>;

- For example:

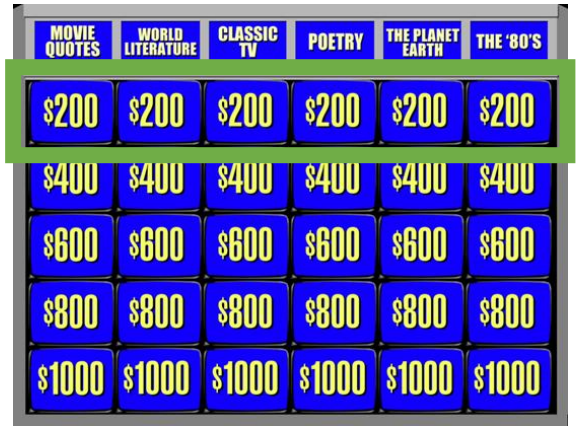
```
jeopardyBoard[4][0] = 1000;
```



2D Array Operations – Assigning to Outer Array

<name>[index$_r$] = <array of correct type>;

- For example:



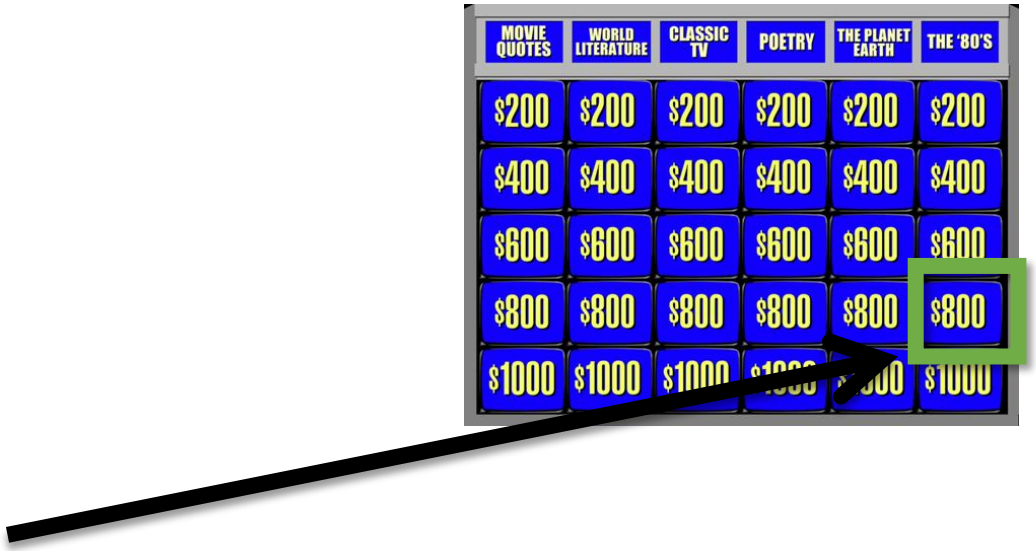

```
jeopardyBoard[0] = [200, 200, 200, 200, 200, 200, 200];
```

2D Array Ops – Accessing Element of Inner Array

<name>[index$_r$][index$_c$]

- For example:

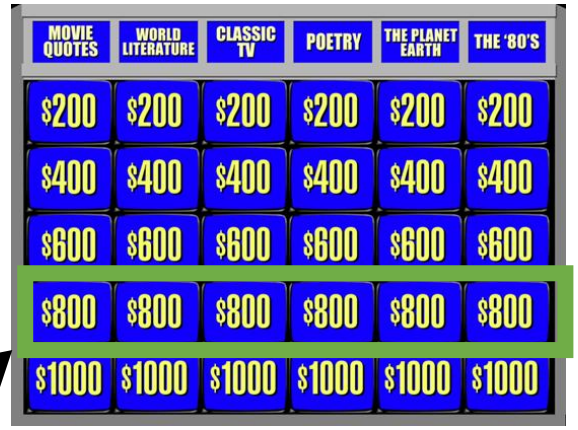
```
print(jeopardyBoard[3][5]);
```



2D Array Ops – Accessing Element of Outer Array

<name>[index$_r$]

- For example:




```
let eightHundreds: number[] = jeopardyBoard[3];
```

2D Array Ops – Get # Elements of Outer Array

<name>.length

- For example:

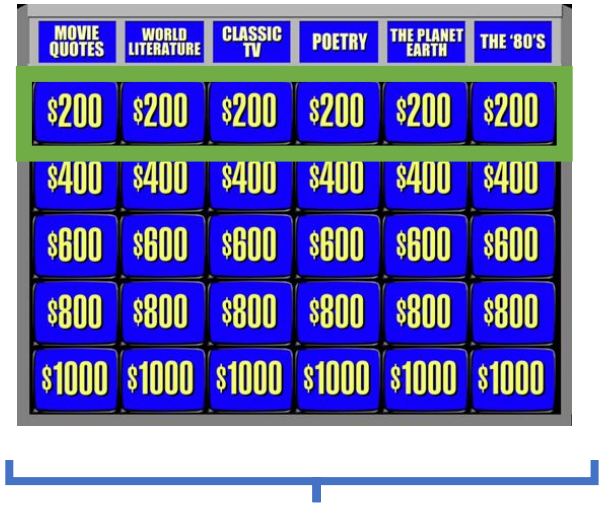
```
print(jeopardyBoard.length);
```

2D Array Ops – Get # Elements of Inner Array

<name>[index$_r$].length

- For example:

```
print(jeopardyBoard[0].length);
```



6

- *Usually* 2D arrays are perfectly rectangular in lengths. *However*, there is no guarantee each inner array has the same # of elements as one another.

```
import { print } from "introcs";
import { array2d } from "./2d-arrays-helpers";

// Declare
let a: number[][];

// Initialize using Literals
a = [
      [1, 2],
      [3, 4],
      [5, 6]
];

// Initialize using array2d helper function
a = array2d(12, 10, 0);

// Assigning to an element
a[3][5] = 1;

// Read the # of rows
print(a.length);

// Read the # of cols
print(a[0].length);

// Read from an element
print(a[1][0]);

// Read from a top-level element
print(a[3]);

// Inspect complete array in developers' tools console
console.log(a);
```

Conway's Game of Life

- A simple "simulation" involving a 2D grid of "**cells**"
  - First implemented in 1970 by John Conway

- A cell can either be **"alive"** (value is 1) or **"dead"** (value is 0)

- At each "step" of the simulation, 4 simple rules are applied to every cell to determine whether it is alive or dead at the next step
  - As these rules are applied, the outcome is assigned to a new 2D grid of cells not modifying the current step. So it's as if these rules are applied instantaneously.

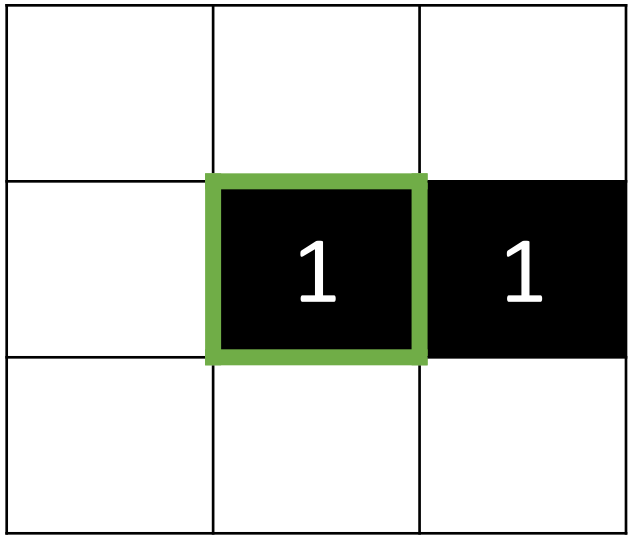
- Complex, emergent behaviors and systems arise from these simple rules.

Conway's Game of Life - Rules

- There are 4 rules, covered in the following 4 slides

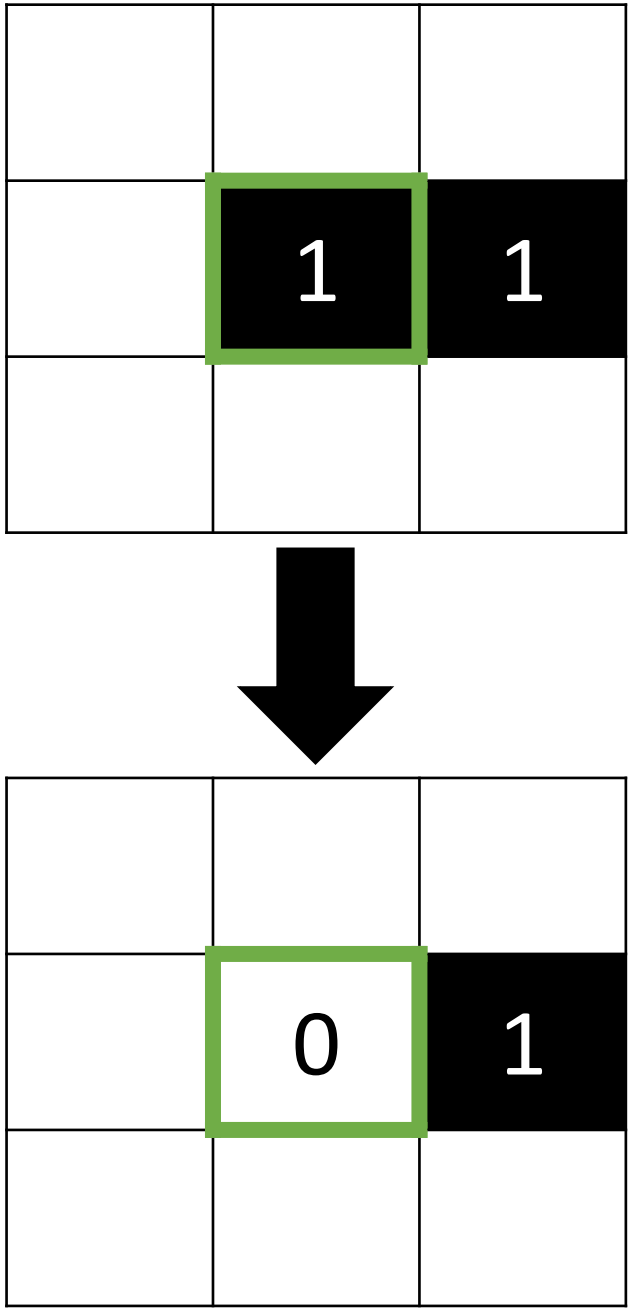
- Note that each example gives the current step and the next step for *only the cell outlined in green.*

- At each step, the same rules will also be applied to all surrounding cells, too, but we will not illustrate this in slides.



Conway's Game of Life - Rules

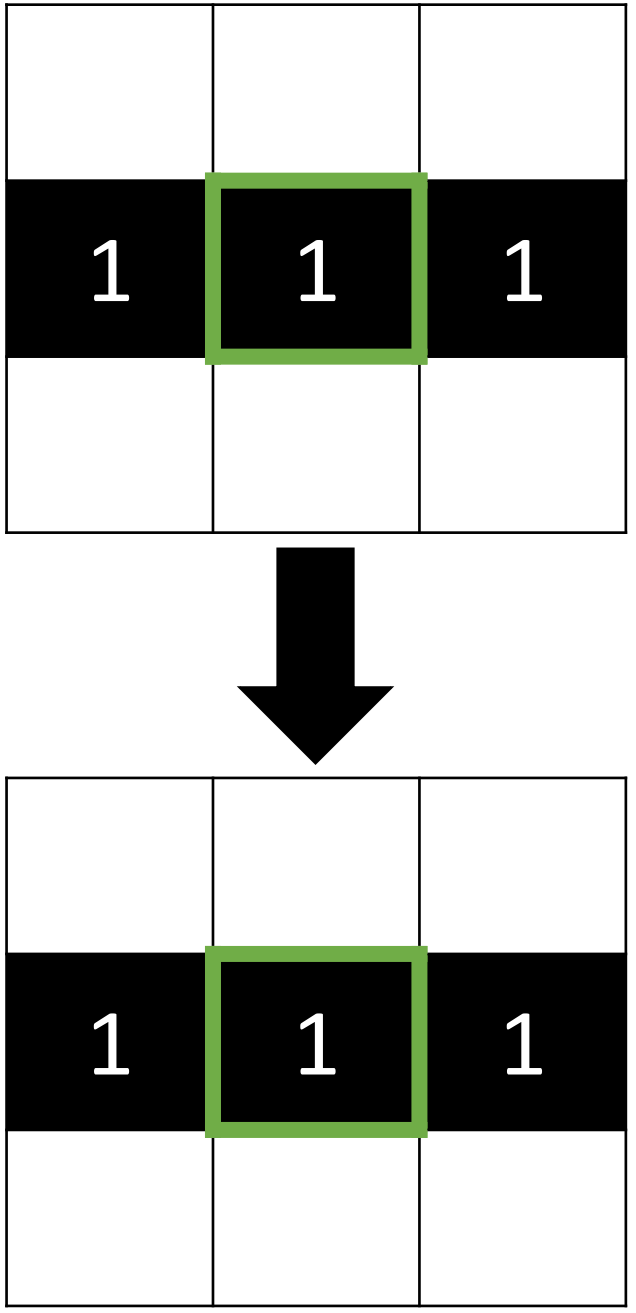
1. **Underpopulation: A live cell with fewer than 2 live neighbors dies.**



Conway's Game of Life - Rules

1. Underpopulation: A live cell with fewer than 2 live neighbors dies.

2. **Stasis: A live cell with 2 or 3 live neighbors survives.**

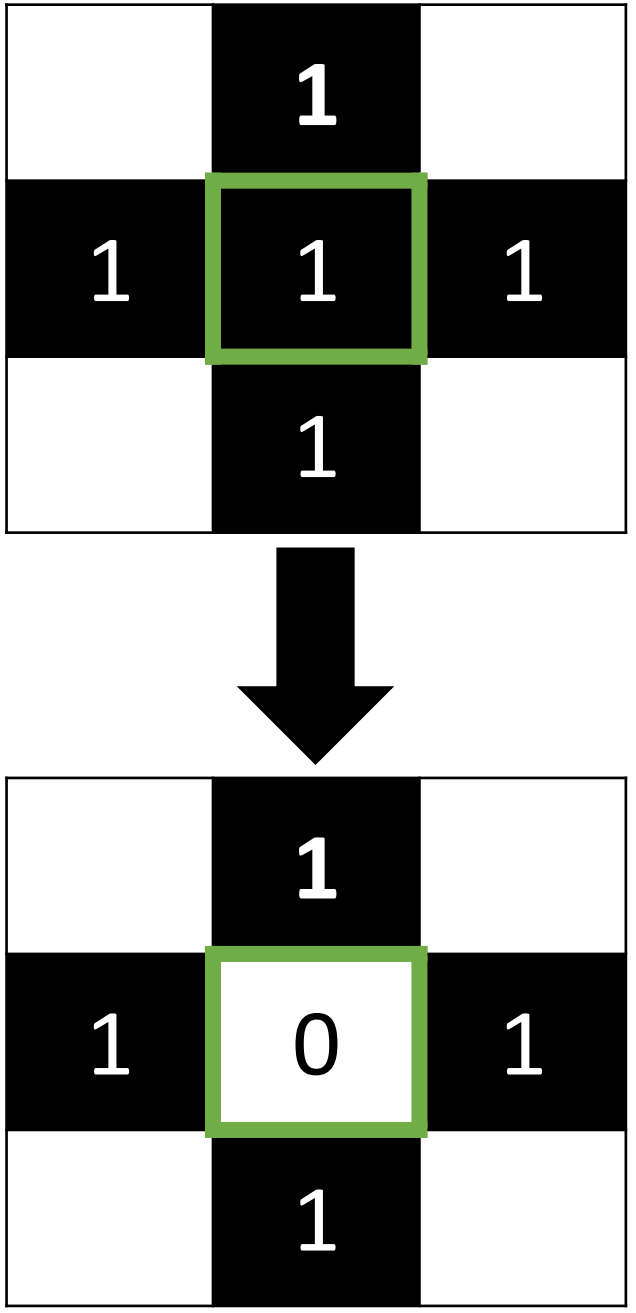


Conway's Game of Life - Rules

1. Underpopulation: A live cell with fewer than 2 live neighbors dies.

2. Stasis: A live cell with 2 or 3 live neighbors survives.

3. **Overpopulation: A live cell with more than 3 live neighbors dies.**



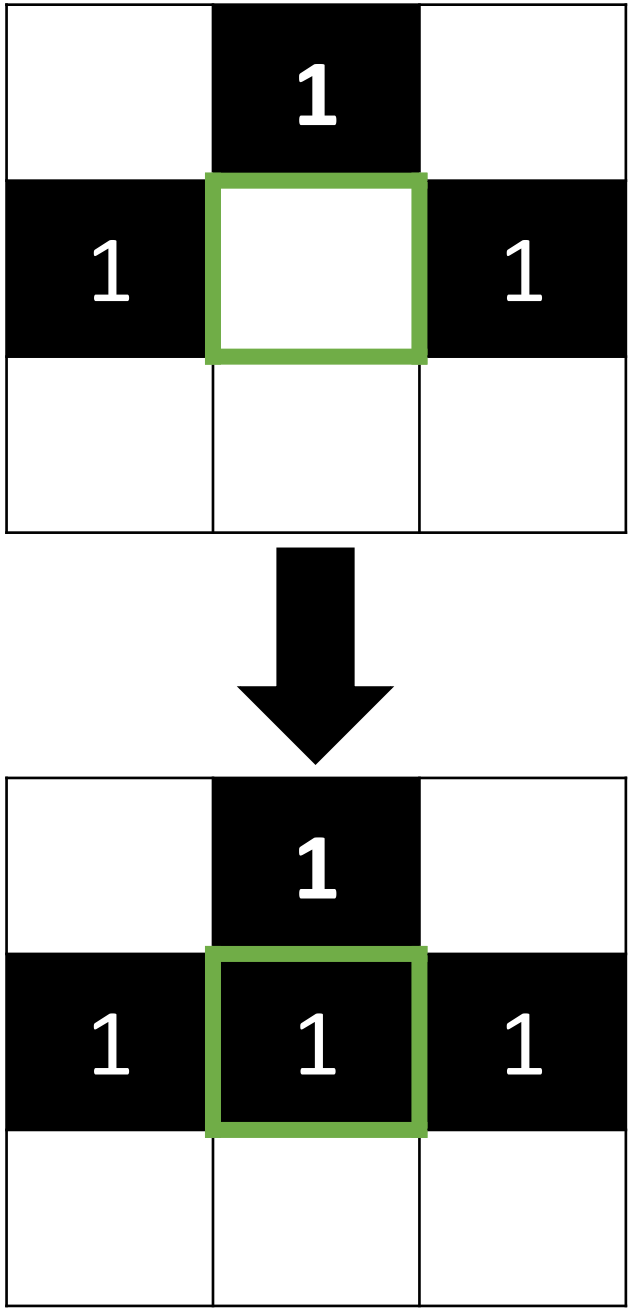
Conway's Game of Life - Rules

1. Underpopulation: A live cell with fewer than 2 live neighbors dies.

2. Stasis: A live cell with 2 or 3 live neighbors survives.

3. Overpopulation: A live cell with more than 3 live neighbors dies.

4. **Reproduction: Any dead cell with 3 live neighbors comes to life.**



Stencil Code Organization

- Today we will only focus on the *model* of Conway's Game of Life and write our code in gol-model.ts

- The "model" of a program refers to its essential data and logic

- The stencil code in today's lecture also contains the code for:
  1. The HTML document containing the user interface elements (game-of-life.html)
  2. The CSS style rules for the table of cells (styles.css)
  3. The visual representation of the grid (gol-view.ts)
  4. The event handling code for the buttons (gol-controller.ts)
  5. The main function that starts the program (game-of-life-script.ts)

- In COMP401 you'll learn about organizing your code using Model-View-Controller

Strategy

1. Write a *method* that will determine whether a cell is live or not

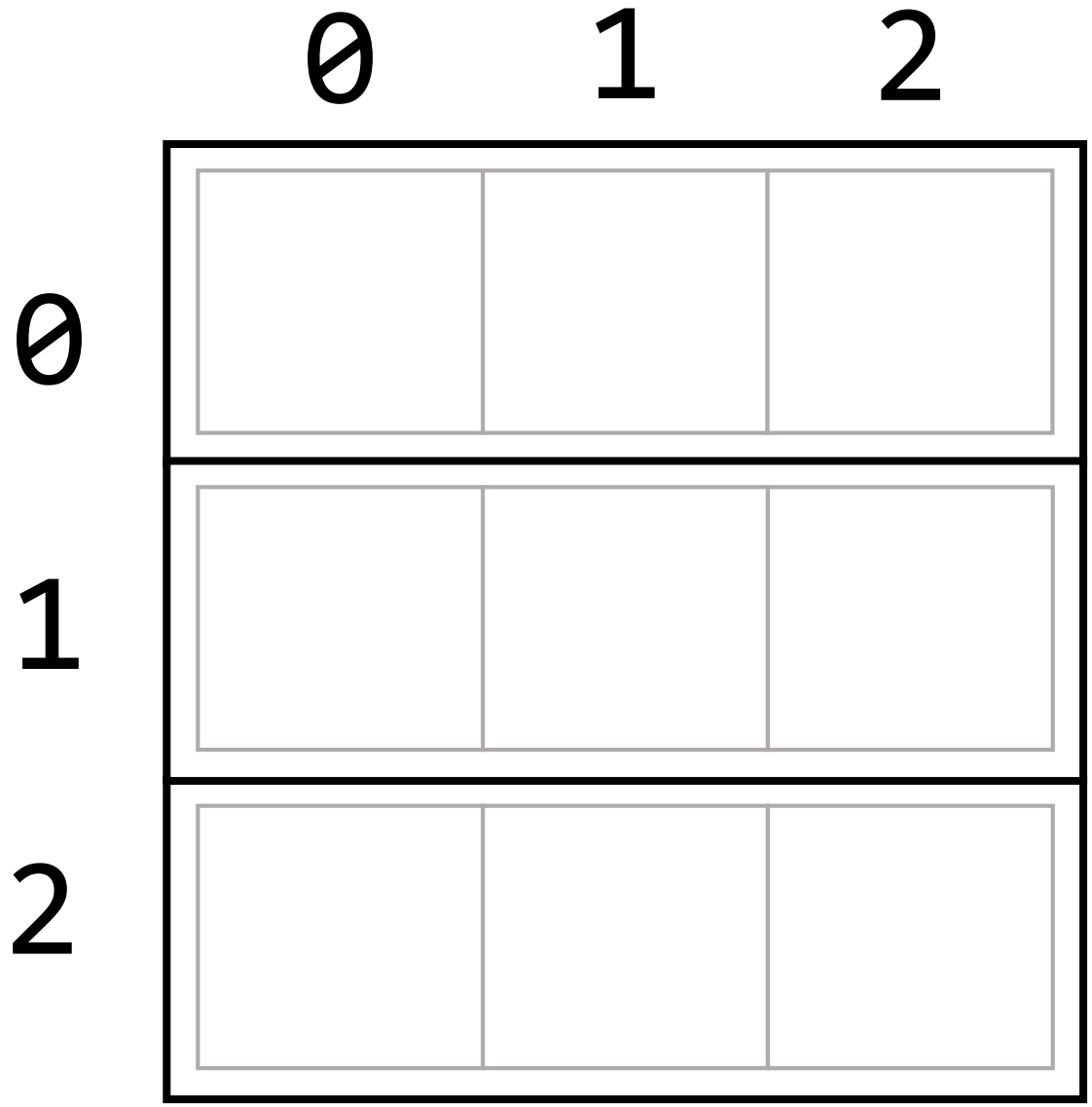
2. Write a *method* to count the number of live neighbors around a cell

3. Write a *method* to "step" through all nodes and call a helper method "rules" to determine the next state of a single cell

isLive

- Let's write a method that will test to see if a given cell is live

- This function will handle special edge cases:
  - It will "wrap around" a row/column if it is out of bounds (think: Pac-Man)
  - For example, if we ask whether the cell at row -1 and column 1 is alive we will actually test row 2 column 1.

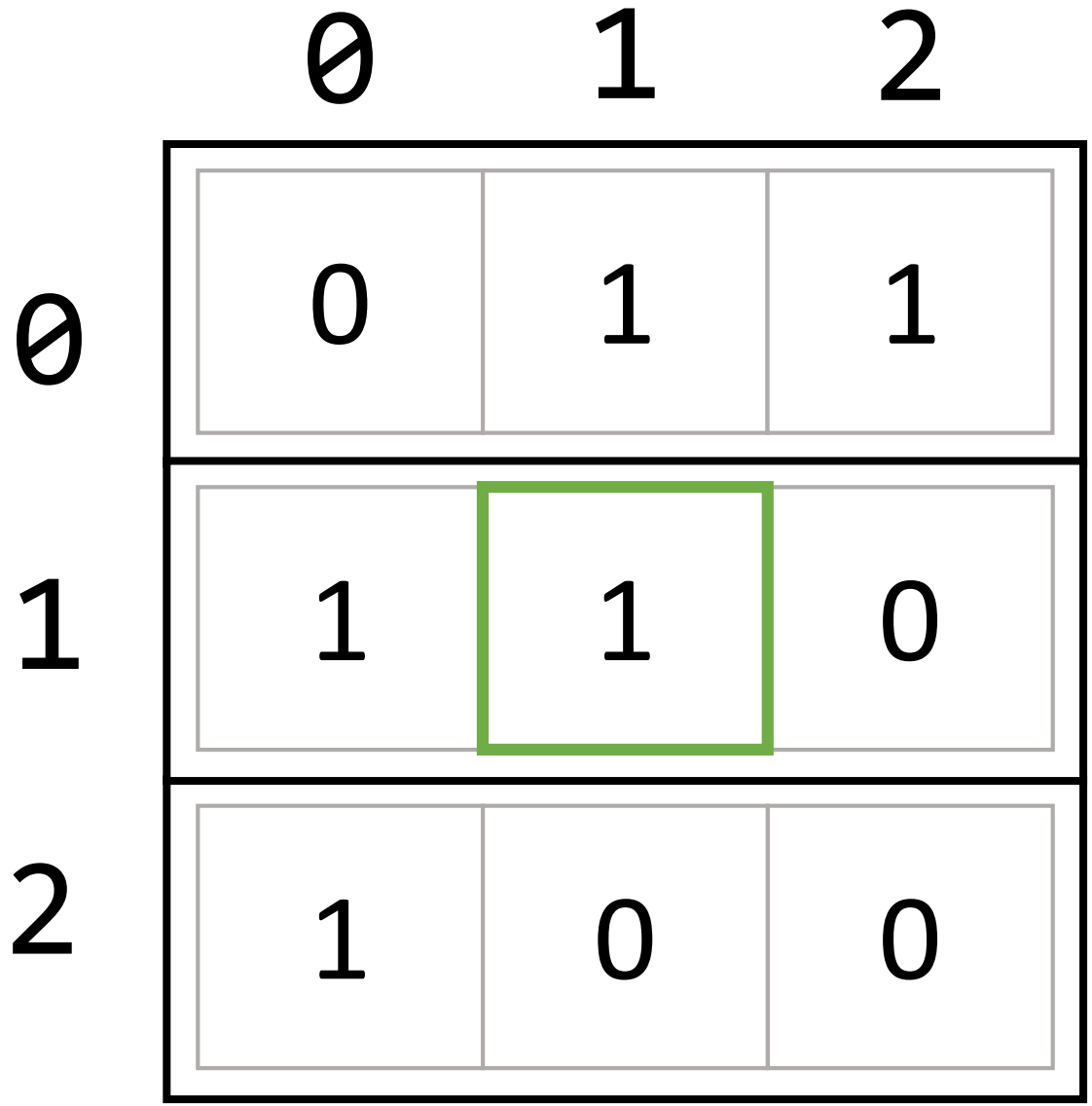


isLive

```
isLive(row: number, col: number): boolean {
    let wrappedRow = (row + this.rows) % this.rows;
    let wrappedCol = (col + this.cols) % this.cols;
    return this.cells[wrappedRow][wrappedCol] === 1;
}
```

countLiveNeighbors

- The rules of the game depend on how many live neighbors surround a given cell
- Let's write a method that checks all surrounding cells and counts the number of 1s
- We'll use this when implementing rules



countLiveNeighbors

```
/**
* Given a row and column, check the surrounding 8 cells and count
* the number which are live.
*/
countLiveNeighbors(row: number, col: number): number {
    let count: number = 0;
    for (let i: number = row - 1; i <= row + 1; i++) {
        for (let h: number = col - 1; h <= col + 1; h++) {
            if (i !== row || h !== col) {
                if (this.isLive(i, h)) {
                    count++;
                }
            }
        }
    }
    return count;
}
```

step & rules

- Let's write a method that sets up an array to contain the next generation of cells.

- It will traverse the current generation of cells in a nested for loop and call a "rules" helper method to determine the next state of the cell.

- The stencil code's controller is already calling the "step" method every time the step button is pressed.

step

```
step(): void {
    let next: number[][] = array2d(this.rows, this.cols);
    for (let row: number = 0; row < this.rows; row++) {
        for (let col: number = 0; col < this.cols; col++) {
            next[row][col] = this.rules(row, col);
        }
    }
    this.cells = next;
}
```

Hands-on: **rules** method

- Given a row and column, apply the following rules.
  - Hint: make use of your this.**isLive** and this.**countLiveNeighbors** methods

- If the cell is **alive**
  - Cell dies of underpopulation if live neighbors < 2
  - Cell survives if live neighbors is 2 or 3
  - Cell dies of overpopulation if live neighbors > 3

- If the cell is **dead**
  - Cell comes to life if live neighbors is 3
  - Otherwise cell remains dead

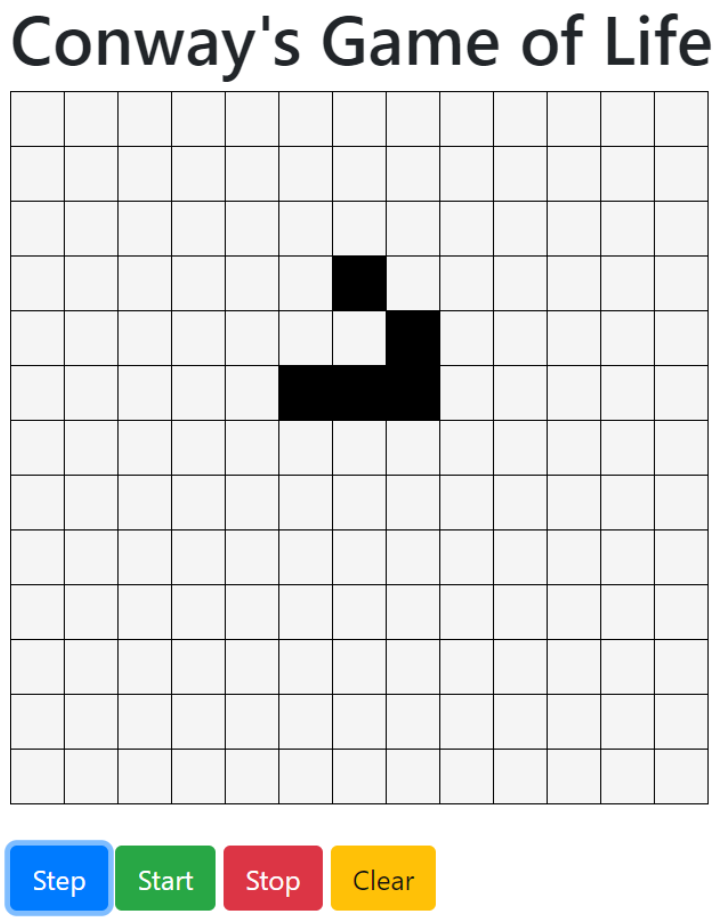
- Return 0 if cell rules result in a dead cell, 1 if cell rules result in a live cell

- Check-in on PollEv.com/compunc when complete

rules

```
rules(row: number, col: number): number {
    let neighbors = this.countLiveNeighbors(row, col);
    if (this.isLive(row, col)) {
        if (neighbors < 2) {
                return 0;
        } else if (neighbors > 3) {
                return 0;
        } else {
                return 1;
        }
    } else {
        if (neighbors === 3) {
                return 1;
        } else {
                return 0;
        }
    }
}
```

Emergent Behavior



- The shape to the left is called a Glider… try it out!
- Over the years many interesting, non-converging patterns have been found. Try searching the web for more.
- Simple example of how a few rules can lead to complex, emergent systems of behavior.