Review

- Exam 1 - Extra Credit
- Recursion
- Call Stack
- Recursive Maze
Cellular Automata

A regular grid of Cells

Cell

Two States
1. On
2. Off

Neighborhood

Cell states evolve over time according to a predefined set of rules.
1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.

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 overcrowding.

4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

An example of "Emergence"

Interesting Patterns – Conway's Game of Life

<table>
<thead>
<tr>
<th>Still lives</th>
<th>Oscillators</th>
<th>Spaceships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>Blinker (period 2)</td>
<td>Glider</td>
</tr>
<tr>
<td>Beehive</td>
<td>Toad (period 2)</td>
<td>Lightweight spaceship (LWSS)</td>
</tr>
<tr>
<td>Loaf</td>
<td>Beacon (period 2)</td>
<td></td>
</tr>
<tr>
<td>Boat</td>
<td>Pulsar (period 3)</td>
<td></td>
</tr>
</tbody>
</table>

Top-level procedure

1. Draw the current grid
2. Advance game by applying rules to all cells of current and filling next
3. Swap current and next grid
int N = 5;
boolean[] cell = new boolean[N];

← One-dimensional array
```java
int N = 5;
boolean[][][] cell = new boolean[N][N];
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
```java
int N = 5;
boolean[][] cell = new boolean[N][N];

cell[1][2] = true;
```
// 3-Dimensional Array

int N = 50;
boolean[][][] cell = new boolean[N][N][2];
cell[1][2][0] = true;
Add the necessary lines of code within `setup()` to fill the `vals` array with random numbers of your choosing. Your implementation must use `for` loops.

```cpp
float[][] vals;

void setup() {
    vals = new float[20][300];

    // Add your code here
}

} // Closing brace for setup()
```
float[][] vals;

void setup() {
    vals = new float[20][300];
    for (int i=0; i<20; i++) {
        println( vals[i].length );  // What is going on here?
    }
}
```java
float[][] ragged;

void setup() {
    ragged = new float[20][];
    for (int i=0; i<20; i++) {
        int N = int( random(100) );
        ragged[i] = new float[N];
    }
    for (int i=0; i<20; i++) {
        println( ragged[i].length );
    }
}
```
float[][] gray = new float[100][100];

void setup() {
    size(500, 500);
    rectMode(CORNER);
    noStroke();
    for (int i=0; i<100; i++) {
        for (int j=0; j<100; j++) {
            //gray[i][j] = int(random(255));
            float v = sin( 0.1*i )*sin( 0.3*j );
            gray[i][j] = map(v, -1.0, 1.0, 0.0, 255.0);
        }
    }
}

for (int i=0; i<100; i++) {
    for (int j=0; j<100; j++) {
        int r = i*5;
        int c = j*5;
        fill( gray[i][j] );
        rect(r, c, 5, 5);
    }
}
Challenge

• Modify the previous example to plot black squares whenever both the row and column of a cell are even.