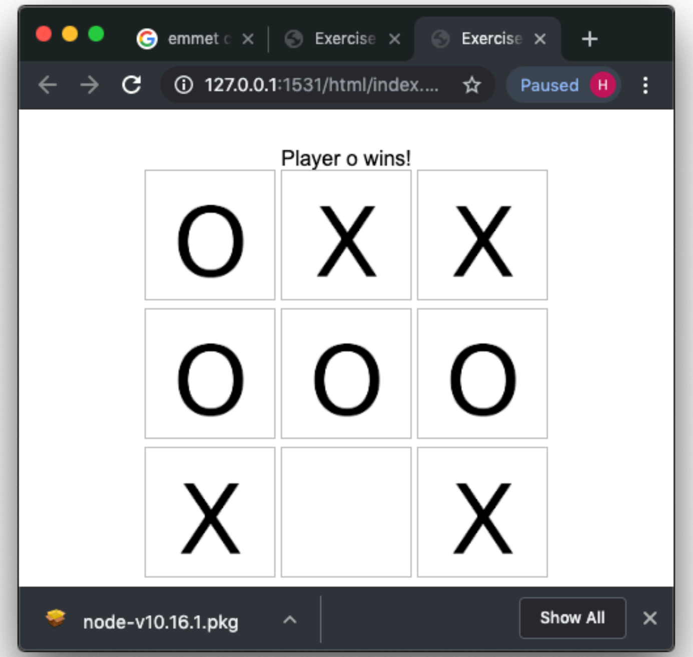
Rev. 1Aug2019

Exercise 46

BUILDING A TIC-TAC-TOE GAME

In this exercise, we'll build a tic-tac-toe game:



I've provided you with the HTML needed for this game in `html/index.html`. The "O" and "X" images are in a separate **images** directory.

Things start to get interesting when we look at `css/index.css`. The rule set for `.cell` is a little complex, but each individual rule is quite straight-forward. The most surprising thing about the CSS is that the classes for `x` and `o` have background images. This is something we haven't seen before. In future exercises, we'll see how we can use background images and then *overlay* text onto them.

A much more complex index.js

This file is quite complex. Rather than ask you to write it, I'm going to provide the code I wrote and we'll walk through it together. (Part of learning to code is seeing how *other* developers have solved a similar problem.)

```
// create a "cells" object to mimic tic-tac-toe board
let cells = {
  a1: null,
  a2: null,
  a3: null,
  b1: null,
  b2: null,
  b3: null,
  c1: null,
  c2: null,
  c3: null,
}
```

Whenever I encounter a grid, I think in terms of a spreadsheet, with letters for the columns and numbers for the rows. The cells object represents this. All of the individual "cells" are given a starting value of `null`. When a player clicks one of the cells, the appropriate cell will be updated to reflect with either an `x` or an `o`.

```
// create "nextPlayer" variable for who the next player will be
let nextPlayer = 'x'
```

We need to keep track of who the next player will be.

```
// create a "togglePlayer" function to switch out nextPlayer
let togglePlayer = () => {
  if (nextPlayer === 'x') {
    nextPlayer = 'o'
  } else {
    nextPlayer = 'x'
  }
}
```

This function will change `nextPlayer`. (If `nextPlayer` is currently `x`, it will change to `o`.)

```
   // create a function, "processClick" to be called when a player clicks a cell
   let processClick = event => {
     // add the class for the next player (which is now current player)
A.      event.target.classList.add(nextPlayer)
     // remove the click event listener since we don't want to respond to a cell that's
        been clicked on
B.   event.target.removeEventListener('click', processClick)

     // find the cell id of the cell clicked on
C.   let cellId = event.target.id

     // in the cell object, register the next player value (which is now current player)
        with the key (where key is the cell id)
D.   cells[cellId] = nextPlayer

     // change next player by calling togglePlayer
E.   togglePlayer()

     // check to see if we have a winner
F.   checkForWinner(tracks)
   }
```

A. We start by adding the appropriate `class` name (`x` or `o`) to the cell that was clicked. Remember that our CSS was done in such a way that the background image will display the appropriate image.

B. We then remove the event listener from the just-clicked cell. Why? Once the cell has been clicked, it's out of play: we don't want *another* click to the same cell to change anything.

C. The HTML was built in such a way that each cell's `id` corresponds with one of the keys in our `cells` object.

D. Since we built the HTML in that way, we can use the `id` of the clicked element as a key in our `cells` object, giving that key a value of `nextPlayer` (either `x` or `o`).

E. Now, we want to toggle the player. If `nextPlayer` was `x`, it should become `o`. This sets us up for the *next* click.

F. After the click occurs, we will call the `checkForWinner` function, passing it the `tracks` array.

```
// create a "display" function to add the event listener to each cell
let display = () => {
  Object.keys(cells).forEach(cell => {
    let element = document.getElementById(cell)
    element.addEventListener('click', processClick)
  })
}
```

The `display` function will be called at the start of a new game. Its job is to loop over the keys in the `cells` object. For each key, we find the HTML with a matching `id` of the key. Once found, we add an event listener to the element.

```
// create a 2d array of all possible way to win: horizontal, vertical, and diagonal
let tracks = [
  ['a1', 'b1', 'c1'],
  ['a2', 'b2', 'c2'],
  ['a3', 'b3', 'c3'],
  ['a1', 'a2', 'a3'],
  ['b1', 'b2', 'b3'],
  ['c1', 'c2', 'c3'],
  ['a1', 'b2', 'c3'],
  ['c1', 'b2', 'a3'],
]
```

A 2d array? Although seldom used, there are times when a 2d (two-dimensional) array is useful. A normal array has one dimension. You can think of them as columns in a *single* row. With a 2d array, we have a grid — multiple columns and multiple rows. This mimics the boxes in our tic-tac-toe game. (Another way to think of 2d arrays as an array whose values are, themselves, arrays. Yes, it's mind-blowing at first.)

```
// create "printWinner" function that will announce winner of the game
let printWinner = player => {
  document.getElementById('winner').innerHTML = `Player ${cells[player]} wins!`
}
```

We create a `printWinner` function that, when called announces the winner.

A.
B.
C.
D.

```
// create a "checkForWinner" function to check for the winner
let checkForWinner = tracks => {
  // loop over each of the tracks
  tracks.forEach(track => {
    // get the "cells" value for each of the tracks
    let trackValue = cells[track[0]] + cells[track[1]] + cells[track[2]]
    // if a given track has all x's or all o's we have a winner
    if (trackValue === "xxx" || trackValue === "ooo") {
      // if we have a winner, call printWinner function
      printWinner(track[0])
    }
  })
}
```

We need a `checkForWinner` function that will be called after each click is processed (by `processClick`).

This code is very hard to grasp. When programmers encounter such code written by another programmer, they spend a great deal of time just trying to understand how things are working. While the comments are invaluable, a great deal of study is still required.

When the programmer is tasked with writing the code themselves, they may spend a great deal of time coming up with an algorithm that will work.

A.The 2d array, `tracks`, was passed into us. We loop over each `track`.

B.`trackValue` will concatenate the values of each key in `cells` corresponding with a track. At then end of this line of code, a typical `trackValue` might be `xxo` or `oxo`. The result of this portion of the code is that every possible winning track (horizontal, vertical, and diagonal) will be checked.

C.If the `trackValue` is `xxx` or `ooo`, we have a winner!

D.If we have a winner, call the `printWinner` function, passing it the winning player.

```
// start things off by calling display function
display()
```

We'll start a new game off by calling `display`, which will set things up.

And, with that, our tic-tac-toe game is done. Now, you almost *certainly* will react to this code by saying, "Oh, no. I knew it was going to get crazy-hard. I'll *never* be able to write something like this." You're just *learning* to code. *Of course* you can't write this code at the stage you're at. (If you could, there really would be a "programming gene".) I've introduced things like looping over the keys in an object — or using a 2d array — things you've never even seen before. But as you *slowly* become accustomed to these, you'll find that, as with anything you're learning, you start to be a *little* better and, over time, you'll be able to write code like this for yourself. But now now, so don't worry!