CS 3312: Web Programming, spring 2020

Studio 7: JavaScript arrays

Follow the instructions below. **Explain the code that you write using comments.** The Mozilla Developer Network and DevDocs may be helpful.

Instructions:

1. Open your `P:\cs3312` folder. Inside it, create a new folder named `studio07`.
2. Save the following files to your `P:\cs3312\studio07` folder by right-clicking on each filename and selecting **Save link as**.
   - `example.html`
   - `example.css`
   - `example.js`
3. Open `example.html` in a standards-compliant browser such as Firefox, Chrome or Opera. Try to interact with the elements on the page and notice what happens.
4. Open `example.html` in Notepad++. Don't forget to change the tab settings: tab size 3 and replace by space. Take note of the structure of the HTML document, especially the `class` and `id` values of each element. The JavaScript code will use them to make the elements interactive.
5. Open `example.js` in Notepad++. Examine the JavaScript code and match up each piece of code to its behavior in the page. Especially notice how arrays are created and used.
6. *Completely optional:* Save the following file to your `P:\cs3312\studio07` folder (replacing the previous `example.js` file). It contains many examples of alternative, more *functional* styles of using arrays in JavaScript.
   - `example.js`

   Those examples are only for those who are interested. That material is purely optional.
7. Save the following files to your `P:\cs3312\studio07` folder.
   - `index.html`
   - `style.css`
   - `script.js`
8. Open `index.html` in Notepad++. Take note of the structure of the HTML document, especially which elements are inside which other elements and which classes are applied to them.
9. Open `script.js` in Notepad++. Add your names where indicated near the top. Write code in `script.js` according to the following instructions.
   a. Write code that uses an immediately-invoked function expression (IIFE) to create the `fibonacci` function where indicated. The IIFE should use an array named `fibonacciResults`, initialized so that `fibonacci(0)` will be `0` and `fibonacci(1)` will be `1`. The function the IIFE returns should have one parameter named `n` and return the `n`th Fibonacci number:
      - If `n` is a finite and nonnegative number (positive or zero):
         - Round `n` to the nearest integer.
         - If the result (`fibonacci(n)`) has never been calculated before, calculate the new result recursively and save it in the `fibonacciResults` array. Saving results in the array so that they don't have to be repeatedly calculated from scratch will save a lot of time in the long run.
         - Return the saved result from the `fibonacciResults` array.
      - If `n` is not a finite number or is a negative number, the function should return the default value of 0.

The On-Line Encyclopedia of Integer Sequences has many values to use when testing your code. When testing, feel free to try input numbers greater than 40. This *memoizing* recursive algorithm has a much more efficient running time than the recursive version we saw before in Studio 5.

b. Write code to make the `#dice` element work:
  - Get an array of all `div` elements inside the `#dice` element. Put the array in the `dieElements` variable.
  - Use a `forEach` loop to iterate through each of those `div` elements one by one. For each one:
    - Roll the corresponding die by generating a random integer between 1 and 6 and putting it in that `div` element.
    - Create an event handler that rerolls the die clicked and all dice to the left of it. Another `forEach` loop will be useful.

c. Write code to make the `#cards` element work:
  - Get an array of all `div` elements inside the `#cards` element. Put the array in the `cardElements` variable.
  - Initialize the `cardValues` variable as an empty array. This array will store the card values that will appear in those `div` elements.
  - Use a `forEach` loop to iterate through each of the `div` elements (the cards) one by one. For each one:
    - Generate a card value, a random integer between 1 and 99. Push it onto the end of the `cardValues` array and put it in the current `div` element.
    - Create an event handler that moves the card to the right end whenever it is clicked, leaving the other cards in the same order, and outputs all the new card values to the card `div`s. The `push`, `slice` and `concat` methods will help you reorder the card values in the `cardValues` array correctly, and another `forEach` loop will be useful to output those new values. (Do not use the `splice` method.)
  - Create an event handler that sorts the cards numerically (smallest on the left, largest on the right) whenever the "Sort" button is clicked and outputs all the new card values to the card `div`s.
  - Create an event handler that reverses the order of the cards whenever the "Reverse" button is clicked and outputs all the new card values to the card `div`s.

d. Write code to make the `#tic-tac-toe` element work:
  - Initialize the `nextToMove` variable as the string value `'X'` to indicate that player X moves first.
  - Initialize the `ticTacToeValues` variable as an empty array. This array will end up having two dimensions (for rows and columns) to store strings for the moves in the tic-tac-toe game: `'X'` for X, `'O'` for O and `''` for empty.
  - Initialize the `ticTacToeElements` variable as an empty array. This array will also have two dimensions and will store the `div` elements of the squares in the game.
  - Use the value of the `nextToMove` variable to output a status message like `X moves next.` to the `#tic-tac-toe-status` element.
  - Get an array of all `tr` elements inside the `#tic-tac-toe` element and use a `forEach` loop to iterate through each of them one by one. For each row:
    - Get an array of all `td` elements inside the current row. Put the array in a `rowElements` variable.
    - Create a `rowValues` variable and initialize it as an empty array. Then, for each element in the `rowElements` array, add

an empty string to the `rowValues` array. The `push` method will be handy here.
        - Push `rowElements` onto the `ticTacToeElements` array and push `rowValues` onto the `ticTacToeValues` array. This step will build the `ticTacToeElements` and `ticTacToeValues` arrays row by row.
    - Use nested `forEach` loops to iterate through the `ticTacToeElements` array row by row. For each `td` element in the array:
        - Use the row number and the column number to output the current value in the `ticTacToeValues` array to the current `td` element.
        - Create an event handler that, if the square clicked was empty, makes the next player's move in the square clicked by updating the `ticTacToeValues` array, changes whose turn it is next, outputs the new value in the `ticTacToeValues` array to the clicked `td` element and updates the status message in the `#tic-tac-toe-status` element.

   You shouldn't change `index.html` or `style.css` in any way. Test your code in as many different browsers and window sizes as you can, and validate your `script.js` file using JSLint. It is very important to get your JavaScript code to pass JSLint without errors before you turn it in. Please get help when you need it.

10. Make sure that you have written brief but helpful comments that make it easier for someone reading your code for the first time to understand what it does and how it works. In particular, a comment before each function should briefly and clearly describe what it does. **Also be sure to include the names of any students you exchanged help with or talked to in comments in your code.**
11. Once you're ready to turn in `script.js`, zip it up into a compressed file named `studio07.zip`:
    - In your `P:\cs3312\studio07` folder, select `script.js`.
    - Right-click on it and select **Send to** and then **Compressed (zipped) folder**.
    - Rename the new file `studio07.zip`.
12. Turn in your `studio07.zip` file using Blackboard.