CIS 111
Introduction to Web Programming

Project 4 – Utility Hub

1. Overview

The **Utility Hub** is a simple web-based application designed to integrate two key utilities:

1. A **Basic Calculator** for arithmetic operations with history tracking.
2. A **NATO Phonetic Alphabet Converter** to transform text into its NATO phonetic equivalent.

The project is divided into two main sections: **Calculator Functionality** and **NATO Phonetic Converter**.

This project will require you to examine the HTML and CSS to determine which CSS/HTML classes or ids are required.

2. Incremental Development

Incremental development means building part of a program before making it all. It is a staging strategy in which product features are developed and delivered to users at different times, increasing its functionality and complexity.

Complete each of the following project stages.

2.1 The starter code

The starter code includes the following files in the Project4 folder.

- p4.html
- p4.css
- p4.js

Important: *You should examine the starter code for required functions and elements. Before adding any code to the body of any functions, you should use single line comments to map out the steps.*

A Simple Calculator

Note that the starter code includes adding a callback function (**setup**) to the browser **window** for the **DOMContentLoaded** event.
When attempting to add callback functions as soon as the web page loads, you must wait until the DOM is completely loaded; otherwise, attempting to reference unloaded HTML elements will result in return a **null** value and the code will fail.
The **DOMContentLoaded** event is the modern event used to wait for the DOM to load.
Note that the **setup()** function adds callback functions for the four operator buttons for the button **click** event, and will call the **calculate()** function with the corresponding operator.
Note that the **calculate()** function uses the **operator** parameter to perform the calculation, update the result, and update the history.

2.2 [10 pts] The setup() Function

The first step is to add callback functions to setup for each of the buttons for the **click** event, and call **calculate()** with the correct operator. We'll use a single line of code to accomplish this task for each button. Below are the elements of the single line of code.

1. Use **querySelector()** to get a reference to the **button** using the button **id** (blue).
2. Use **addEventListener()** for the **click** event to call an anonymous function (orange).
3. Use the anonymous function to call **calculate()** for the corresponding operator (purple).

Below is the code for the **add** (+) operator.

```
document.querySelector("#add").addEventListener("click", function () {
  calculate("+");
});
```

4. Add the above code to the **setup()** function.
5. Update the **calculate()** function to output the operator parameter to the Console.

```
console.log(`Operator: ${operator}`);
```

6. Save and test your code, and confirm when clicking on the **+** (add) button the Console confirms the callback function worked.

7. Repeat this process for the remaining three buttons, making sure to save and test for each new button/operator callback.

2.3 [10 pts] The calculate() Function: Getting the Numbers

We need to get the input box numbers, but because input box values are strings, we need to convert the values into numbers. We are able to accomplish this task with a single line of code. Below are the elements of the single line of code.

1. Get a reference to the **input** box using **querySelector()** (blue).
2. Use the **value** property to the get the **input** box content (orange).
3. Use **parseFloat()** to convert the string value into a number (purple).
4. Assign the number to a variable (green).

```
const num1 = parseFloat(document.querySelector("#num1").value);
```

5. Add the above line of code to **calculate()** to get the first number.
6. Add code to output **num1** to the Console.
7. Save the code, and test by adding a number into the first number input box and click the **+** button.
8. Confirm the number appears in the Console.
9. Repeat this process for the second number, making sure to save and test that both numbers now appear in the console.
10. Once the code works correctly, you can comment out or remove the Console output code.
11. After that, add code to alert the user if they click on one of the operators without entering the numbers in the input fields:

```
if (isNaN(num1) || isNaN(num2)) {

  alert(“Please enter valid numbers”);

  return;

}
```

12. Call the **getResult(…)** function with appropriate arguments/actual parameters and store it in a result variable reference.

13. Lastly, call the **displayCalculatorResult(…)** function with appropriate arguments/actual parameters to display the result.

2.4 [10 pts] The getResult() Function

The calculation to be performed depends on the **operator** parameter, so you can either use if/else if statements or a switch statement.

Below is the implementation of the addition operation:

```
let result;

if (operator === “+”) {

  result = num1 + num2;

}

return result;
```

1. Add the above code to the **getResult()** function.
2. Output result to the Console.
3. Save, test, and confirm the **result** calculated value is correct.
4. Add the additional operator cases, and repeat the testing process.
5. Make sure you have a case for “undefined” result for the division operator.
6. Once the code works correctly, you can disable or remove the Console output code.

2.5 [10 pts] The displayCalculatorResult() Function

Now that the calculation works, we need to display the result. We're using an **input** HTML element for the output, but have included the **readonly** attribute so the calculated value cannot be changed by the user. We use the **value** property to update what's displayed in the **input** box.

Below is the code to display the result.

```
document.querySelector(“#result”).value = result;
```

1. Add the above code to the **displayCalculatorResult()** function.
2. Save and test the code and confirm the calculated result displays in the **result input** box.

The last requirement is to display the history of the calculations. The HTML code includes an empty unordered list element (**ul**). We'll use JavaScript to create a new list item element (**li**), update the list item content, and append this new element to the unordered list.

Below is the process to create and add dynamic HTML elements to a web page.

1. Get a reference to the parent element that will receive the new HTML element using **querySelector()**.
2. Create the new element using the **createElement()** function
3. Update the new element, such as using the **textContent** property.
4. Add the new element to the parent element using the **appendChild()** function.

```
const history = document.querySelector(“#history”);
const newHistory = document.createElement(“li”);
newHistory.textContent = //TODO
history.appendChild(newHistory);
```

5. After filling out the TODO part, add the above code to the **displayCalculatorResult()** function.
6. Save and test to confirm that the history capability works.

2.6 [10 pts] The clearHistory() and clearCalculatorInputs() Functions

1. Use **querySelector()** and the ids: history, num1, num2 and result.
2. You can use **history.innerHTML = “”;** and **num1Ref.value = “”;** to clear the history and input fields.
3. Please use the in-class project (Simple EventListener Demo) as a sample.

2.7 Submission

Upload simple calculator part to Canvas’s Lab9 submission to get 5 points for Lab. You will continue the next part after that submit them again to Project 4 to get all the points for Project 4.

Remember, you are responsible to submit your projects before the deadline. Late submissions will not be accepted.