LEARN FORM VALIDATION BY BUILDING A CALORIE COUNTER

Introduction:

Sometimes when you're coding a web application, you'll need to be able to accept input from a user. In this calorie counter project, you'll learn how to validate user input, perform calculations based on that input, and dynamically update your interface to display the results.

In this practice project, you'll learn basic regular expressions, template literals, the `addEventListener()` method, and more.

Step 1:

In this project, you'll learn to create a calorie counter form that enables users to input their daily calorie budget and the calorie counts of various meals. The form will then calculate and display whether the user is in a calorie deficit or surplus.

You have been provided with boilerplate CSS and HTML. However, you need to build your calorie counter form.

Feel free to explore the HTML and CSS, then add a `form` element and give it an `id` set to `calorie-counter`.

Step 2:

In your form, users will be able to input a number which represents their daily calorie budget.

Create a `label` element, give it a `for` attribute set to `budget` and the text `Budget`, then create an `input` element with the `id` set to `budget`.

Step 3:

Your `input` element needs some additional attributes. Give it a `type` set to `number` to only allow numeric inputs, a `min` attribute set to `0` to only allow positive numbers, and a `placeholder` set to `Daily calorie budget`.

Finally, mark the `input` element as `required`.

Step 4:

In your form, users should have the capability to add various meal types along with their calorie counts.

Create a `fieldset` element with the `id` set to `breakfast`.

Within that element, create a `legend` with the text `Breakfast`, and an empty `div` with the `class` set to `input-container`.

Step 5:

Next, create a `fieldset` element with the `id` set to `lunch`.

Within that element, create a `legend` element with the text `Lunch`, and an empty `div` with the `class` set to `input-container`.

Step 6:

Continuing the pattern, create a `fieldset` for `dinner` with the same nested elements.

Step 7:

You need two more of these `fieldset` code blocks – one for `snacks` and one for `exercise`.

Step 8:

When users want to select a meal type to input their calorie counts, they should be presented with a dropdown menu and a button to add the meal type.

Start by creating a `div` element and assign it a `class` attribute with the value `controls`. Then, nest a `span` element inside this `div`.

Step 9:

In your `span` element, create a `label` element for an `entry-dropdown` and give it the text `Add food or exercise:`. Then create a `select` element with the `id` set to `entry-dropdown` and a `name` set to `options`. Below that, add a `button` element with the `id` set to `add-entry` and the text `Add Entry`.

Give your `button` element a `type` attribute set to `button` to prevent automatic form submission.

Step 10:

Your select menu needs options for each of the food and exercise `fieldset` elements you created in the previous steps. Use the `option` element to create a new option for each `fieldset`. The `value` attribute of each option should be the `id` of the `fieldset`, and the text of each option should be the text of the `legend`.

Set the `Breakfast` option as the `selected` option.

Step 11:

Create another `div` element. Within it, nest a `button` to `submit` the form. This button should have the text `Calculate Remaining Calories`.

Then add a `button` with the `id` set to `clear` to clear the form (don't forget to give it a `type` attribute that prevents it from submitting the form). This button needs the text `Clear`.

Step 12:

Your form needs somewhere to display the results. Add an empty `div` element and give it an `id` of `output` and the `class` values of `output` and `hide`.

Step 13:

Finally, you need to link your JavaScript file to your HTML. Create a `script` element to do so.

Step 14:

It is time to start writing the script that makes your form work.

To access an HTML element with a given `id` name, you can use the `getElementById()` method. Here's an example of how to use this method:

Example Code:

```
<h1 id="title">Main title</h1>
```

Example Code:

```
const mainTitleElement = document.getElementById('title');
```

Begin by getting the `form` element (using the `id`) and storing it in a variable called `calorieCounter`.

Step 15:

Get your `#budget` element and assign it to `budgetNumberInput`, and your `#entry-dropdown` element and assign it to `entryDropdown`.

Step 16:

Following the same pattern, assign your `#add-entry` element to `addEntryButton`, your `#clear` element to `clearButton`, and your `#output` element to `output`.

Step 17:

In programming, prefixing a variable with `is` or `has` is a common practice to signify that the variable represents a boolean value.

Here are a few examples:

Example Code:

```
let isRunning = true;

let hasCompleted = false;
```

Declare a variable named `isError` using `let` and initialize it with `false`, allowing for its reassignment later.

Later on in the project, you will update the value of `isError` if the user provides an invalid input.

Step 18:

When the user inputs their daily calorie budget, the input field will only accept numerical values. However, if a number is entered with a `+` or `-` sign, you'll need to remove those characters.

Start by declaring a `cleanInputString` function that takes a `str` parameter.

NOTE: Values from an HTML `input` field are received as strings in JavaScript. You'll need to convert these strings into numbers before performing any calculations. Converting string values into numbers will be covered in a future step.

Step 19:

To match specific characters in a string, you can use Regular Expressions or "regex" for short.

Regex in JavaScript is indicated by a pattern wrapped in forward slashes. The following example will match the string literal `"hello"`:

Example Code:

```
const regex = /hello/;
```

Declare a `regex` variable and assign it the value from the example above. In future steps, you will update this regex pattern to match specific characters needed for the calorie counter.

Step 20:

The current pattern will match the exact text `"hello"`, which is not the desired behavior. Instead, you want to search for `+`, `-`, or spaces. Replace the pattern in your `regex` variable with `\+-` to match plus and minus characters.

Note that you need to use the backslash `\` character to escape the `+` symbol because it has a special meaning in regular expressions.

Step 21:

In regex, shorthand character classes allow you to match specific characters without having to write those characters in your pattern. Shorthand character classes are preceded with a backslash (`\`). The character class `\s` will match any whitespace character. Add this to your regex pattern.

Step 22:

Your current pattern won't work just yet. `/+-\s/` looks for `+`, `-`, and a space *in order*. This would match `+- hello` but would not match `+hello`.

To tell the pattern to match each of these characters individually, you need to turn them into a character class. This is done by wrapping the characters you want to match in brackets. For example, this pattern will match the characters `h`, `e`, `l`, or `o`:

Example Code:

```
const regex = /[helo]/;
```

Turn your `+-\s` pattern into a character class. Note that you no longer need to escape the `+` character, because you are using a character class.

Step 23:

Regex can also take specific flags to alter the pattern matching behavior. Flags are added after the closing `/`. The `g` flag, which

stands for "global", will tell the pattern to continue looking after it has found a match. Here is an example:

Example Code:

```
const helloRegex = /hello/g;
```

Add the `g` flag to your regex pattern.

Step 24:

JavaScript provides a `.replace()` method that enables you to replace characters in a string with another string. This method accepts two arguments. The first argument is the character sequence to be replaced, which can be either a string or a regex pattern. The second argument is the string that replaces the matched sequence.

Since strings are immutable, the `replace` method returns a new string with the replaced characters.

In this example, the `replace` method is used to replace all instances of the letter `l` with the number `1` in the string `hello`.

Example Code:

```
"hello".replace(/l/g, "1");
```

Use your `regex` to replace all instances of `+`, `-`, and a space in `str` with an empty string. Return this value.

Step 25:

Now it is time to test out your `cleanInputString` function.

Inside your `cleanInputString` function, add a `console.log()` statement with two arguments. The first argument should be the string `"original string: "` and the second argument should be the `str` parameter.

Step 26:

To see the results from the `cleanInputString` function, you will need to add a `console.log()` statement. Inside that console statement, call the `cleanInputString` function with the string value of `"+-99"` as an argument.

Open up the console and you should see the original string followed by the cleaned string value with the `+-` removed.

Step 27:

Once you have finished testing your `cleanInputString` function, you can remove both of your console statements.

Step 28:

In HTML, number inputs allow for exponential notation (such as `1e10`). You need to filter those out.

Start by creating a function called `isInvalidInput` – it should take a single `str` parameter.

Step 29:

Declare a `regex` variable, and assign it a regex that matches the character `e`.

Step 30:

The `e` in a number input can also be an uppercase `E`. Regex has a flag for this, however – the `i` flag, which stands for "insensitive".

Example Code:

```
/Hello/i
```

The regex above would match `hello`, `Hello`, `HELLO`, and even `hElLo` because of the `i` flag. This flag makes your pattern case-insensitive.

Add the `i` flag to your regex pattern.

Step 31:

Number inputs only allow the `e` to occur between two digits. To match any number, you can use the character class `[0-9]`. This will match any digit between `0` and `9`.

Add this character class before and after `e` in your pattern.

Step 32:

The `+` modifier in a regex allows you to match a pattern that occurs one or more times. To match your digit pattern one or more times, add a plus after each of the digit character classes. For example: `[0-9]+`.

Step 33:

There is a shorthand character class to match any digit: `\d`. Replace your `[0-9]` character classes with this shorthand.

Step 34:

Strings have a .match() method, which takes a regex argument. `.match()` will return an array of match results – containing either the first match, or all matches if the global flag is used.

Example Code:

```
const str = 'example string';

const regex = /example/;

const result = str.match(regex); // Returns ['example']
```

Return the result of calling the `.match()` method on `str` and passing your `regex` variable as the argument. You'll use this match result later on.

Step 35:

Now it is time to test your `isInvalidInput` function. For this test, you want to check if the function can detect scientific notation like `1e3` or `10e2`. While this is a valid way to represent numbers, it is not a valid input for your calorie counter project.

Below your `isInvalidInput` function, add a console statement. Inside that console statement, call the `isInvalidInput` function with an argument of `"1e3"`.

Open up the console to see the result. In the next step, you will learn more about what that result means.

Step 36:

When you open the console, you should see this result:

Example Code:

```
[ '1e3', index: 0, input: '1e3', groups: undefined ]
```

The `match` method returns an array with any matches found in the string.

Here is a complete breakdown of that information:

- `"1e3"` is the matched value against the `/\d+e\d+/i` regex.
- `index: 0` is the index of the matched value in the string.
- `input: '1e3'` is the original string that was matched.
- `groups: undefined` are the matched groups, which are not used in this case. You will learn more about groups in a later project.

Now it is time to test for a valid input. Update your console statement to the following: `console.log(isInvalidInput("10"))`.

Open up the console to see the result. You will learn more about what this result means in the next step.

Step 37:

When you open the console, you should see the result of `null`. The `match` method returns `null` when no match is found. In this case, the `isInvalidInput` function should return `null` when the input is a valid number without any scientific notation.

`null` in JavaScript is a special primitive that represents the intentional absence of a value. In a boolean context, `null` is considered falsy which evaluates to `false` in a conditional statement.

Now that you have finished testing your `isInvalidInput` function, you can remove the `console.log` statement.

Step 38:

Now you need to retrieve the value of `entryDropdown.value` to get the currently selected option from the dropdown.

Print `entryDropdown.value` to the console to see its value.

Since `entryDropdown.value` is in a static context (outside of an event listener), it only shows the value at the moment the code runs. This means it won't automatically update as the user interacts with the dropdown.

It will capture the initial value (in this case, `"breakfast"`) and won't reflect any changes the user makes afterward.

Step 39:

Now that you have finished testing the value of `entryDropdown.value`, you can remove the `console.log` statement.

Your next step is to allow users to add entries to the calorie counter. Declare an empty function `addEntry`. This function should not take any parameters.

Step 40:

You'll need to know which category the entry goes in. Thankfully, you added a dropdown for the user to select a category.

Remember that you queried that dropdown earlier in your JavaScript and assigned it to the `entryDropdown` variable. You can use the value property to get the value of the selected option.

Use concatenation to add a `#` to the beginning of the `value` property of `entryDropdown`, and assign that result to a `targetId` variable.

Step 41:

Now you need to target the `.input-container` element within the element that has your `targetId`. Declare a new `targetInputContainer` variable, and assign it the value of `document.querySelector()`. Use concatenation to separate `targetId` and `'.input-container'` with a space, and pass that string to `querySelector()`.

Step 42:

JavaScript has a feature called template literals, which allow you to interpolate variables directly within a string. Template literals are denoted with backticks ``, as opposed to single or double quotes. Variables can be passed in to a template literal by surrounding the variable with `${}` – the value of the variable will be inserted into the string.

For example:

Example Code

```
const name = "Naomi";

const templateLiteral = `Hello, my name is ${name}~!`;

console.log(templateLiteral);
```

The console will show the string "Hello, my name is Naomi~!".

Replace your concatenated string in the `querySelector` with a template literal – be sure to keep the space between your `targetId` variable and `.input-container`.

Step 43:

Thanks to template literals, you actually don't need the `targetId` variable at all. Remove that variable, and update your template

literal to replace `targetId` with `entryDropdown.value` – remember to add `#` before that, in the string.

Step 44:

You will want to number the entries a user adds. To get all of the number inputs, you can use the querySelectorAll() method.

The `querySelectorAll()` method returns a `NodeList` of all the elements that match the selector. A `NodeList` is an array-like object, so you can access the elements using bracket notation.

Declare an `entryNumber` variable and give it the value of `targetInputContainer.querySelectorAll()`. You do not need to pass an argument to the query selector yet.

Step 45:

Each entry will have a text input for the entry's name, and a number input for the calories. To get a count of the number of entries, you can query by text inputs.

Pass the string `input[type="text"]` to the `querySelectorAll()` method. Remember that you will need to use single quotes for your string, so that you can use double quotes within.

This will return a `NodeList` of all the text inputs in the form. You can then access the `length` property of the `NodeList` to get the number of entries. Do this on the same line.

Step 46:

Now you need to build your dynamic HTML string to add to the webpage. Declare a new `HTMLString` variable, and assign it an empty template literal string.

Step 47:

Inside your template literal, create a `label` element and give it the text `Entry # Name`. Using your template literal syntax, replace `#` with the value of `entryNumber`.

Step 48:

Give your `label` element a `for` attribute with the value `X-#-name`, where `X` is the `value` of the `entryDropdown` element and `#` is the value of `entryNumber`. Remember that HTML attributes should be wrapped in double quotes.

Step 49:

After your `label` element, and on a new line in your template string, create an `input` element. Give it a `type` attribute set to `text`, a `placeholder` attribute set to `Name`, and an `id` attribute that matches the `for` attribute of your `label` element.

Step 50:

Create another `label` element (on a new line) at the end of your `HTMLString`. This `label` should have the text `Entry # Calories`, using your template literal syntax to replace `#` with the value of `entryNumber`, and the `for` attribute set to `X-#-calories`, where `X` is the `value` of `entryDropdown` and `#` is the value of `entryNumber`.

Step 51:

Finally, on a new line after your second `label`, create another `input` element. Give this one a `type` attribute set to `number`, a `min` attribute set to `0` (to ensure negative calories cannot be added), a `placeholder` attribute set to `Calories`, and an `id` attribute that matches the `for` attribute of your second `label` element.

Step 52:

To see your new HTML content for the `targetInputContainer`, you will need to use the innerHTML property.

The `innerHTML` property sets or returns the HTML content inside an element.

Here is a `form` element with a `label` and `input` element nested inside.

Example Code:

```
<form id="form">

  <label for="first-name">First name</label>

  <input id="first-name" type="text">

</form>
```

If you want to add another `label` and `input` element inside the form, then you can use the `innerHTML` property as shown below:

Example Code:

```
const formElement = document.getElementById("form");

const formContent = `

  <label for="last-name">Last name</label>

  <input id="last-name" type="text">
```

```
`;

formElement.innerHTML += formContent;
```

Use the addition assignment operator `+=` to append your `HTMLString` variable to `targetInputContainer.innerHTML`.

Step 53:

In the Role Playing Game project, you learned how to set a button's behavior by editing its `onclick` property. You can also edit an element's behavior by adding an event listener.

The following example uses the `addEventListener` method to add a click event to a button. When the button is clicked, the `printName` function is called.

Example Code:

```
<button class="btn">Print name</button>
```

Example Code:

```
const button = document.querySelector('.btn');

function printName() {

  console.log("Jessica");

}

button.addEventListener('click', printName);
```

The addEventListener method takes two arguments. The first is the event to listen to. (Ex. 'click') The second is the callback function, or the function that runs when the event is triggered.

Call the .addEventListener() method on the addEntryButton. Pass in the string "click" for the first argument and the addEntry function for the second argument.

Note that you should not *call* addEntry, but pass the variable (or function reference) directly.

Step 54:

Try adding a couple of entries to the Breakfast category, and you may notice some bugs! The first thing we need to fix is the entry counts – the first entry should have a count of 1, not 0.

This bug occurs because you are querying for input[type="text"] elements *before* adding the new entry to the page. To fix this, update your entryNumber variable to be the value of the length of the query plus 1. Add this on your declaration line, not in your template strings.

Step 55:

Your other bug occurs if you add a Breakfast entry, fill it in, then add a second Breakfast entry. You'll see that the values you added disappeared.

This is because you are updating innerHTML directly, which does not preserve your input content. Change your innerHTML assignment to use the insertAdjacentHTML() method of targetInputContainer instead. Do not pass any arguments yet.

Step 56:

The `insertAdjacentHtml` method takes two arguments. The first argument is a string that specifies the position of the inserted element. The second argument is a string containing the HTML to be inserted.

For the first argument, pass the string `"beforeend"` to insert the new element as the last child of `targetInputContainer`.

For the second argument, pass your `HTMLString` variable.

Step 57:

Great! Now you can add entries without losing your previous inputs.

Your next step is to write a function that will get the calorie counts from the user's entries.

Declare a `getCaloriesFromInputs` function, and give it a parameter called `list`.

Step 58:

In your new function, declare a `calories` variable and assign it the value `0`. Use `let` to declare it, since you will be reassigning it later.

Step 59:

The `list` parameter is going to be the result of a query selector, which will return a `NodeList`. A `NodeList` is a list of elements like an array. It contains the elements that match the query selector. You will need to loop through these elements in the list.

In previous steps, you learned how to loop through an array using a `for` loop. You can also use a for...of loop to loop through an array and a `NodeList`.

A `for...of` loop is used to iterate over elements in an iterable object like an array. The variable declared in the loop represents the current element being iterated over.

Example Code:

```
for (const element of elementArray) {

  console.log(element);

}
```

Create a `for...of` loop that loops through the `list`. For the loop's variable name, use `const` to declare a variable called `item`.

Step 60:

The `NodeList` values you will pass to `list` will consist of `input` elements. So you will want to look at the `value` attribute of each element.

Assign `item.value` to a `const` variable called `currVal`.

Step 61:

Remember that you wrote a function earlier to clean the user's input? You'll need to use that function here.

Update your `currVal` declaration to be the result of calling `cleanInputString` with `item.value`.

Step 62:

You also need to confirm the input is valid. Declare an `invalidInputMatch` variable, and assign it the result of calling your `isInvalidInput` function with `currVal` as the argument.

Step 63:

Remember that your `isInvalidInput` function returns `String.match`, which is an array of matches or `null` if no matches are found.

In JavaScript, values can either be truthy or falsy. A value is truthy if it evaluates to `true` when converted to a Boolean. A value is falsy if it evaluates to `false` when converted to a Boolean. `null` is an example of a falsy value.

You need to check if `invalidInputMatch` is truthy – you can do this by passing the variable directly to your `if` condition (without a comparison operator). Here's an example of checking the truthiness of `helloWorld`.

Example Code:

```
if (helloWorld) {

}
```

Add an `if` statement that checks if `invalidInputMatch` is truthy.

Step 64:

Browsers have a built in `alert()` function, which you can use to display a pop-up message to the user. The message to display is passed as the argument to the `alert()` function.

Using a template literal, in your `if` block, call the `alert()` function to tell the user `"Invalid Input: "`, followed by the first value in the `invalidInputMatch` array.

Step 65:

In programming, `null` is meant to represent the absence of a value. In this case, if the user enters an invalid input, you want to alert them and then return `null` to indicate that the function has failed.

Still within your `if` block, set `isError` to `true` and return `null`.

Step 66:

Remember that `return` ends the execution of a function. After your `if` block, you need to handle the logic for when the input is valid. Because your `if` statement returns a value, you do not need an `else` statement.

Use the addition assignment operator to add `currVal` to your `calories` total. You'll need to use the `Number` constructor to convert `currVal` to a number.

The `Number` constructor is a function that converts a value to a number. If the value cannot be converted, it returns `NaN` which stands for "Not a Number".

Here is an example:

Example Code:

```
Number('10'); // returns the number 10

Number('abc'); // returns NaN
```

Step 67:

After your `for` loop has completed, return the `calories` value.

Step 68:

Now it's time to start putting it all together. Declare an empty `calculateCalories` function, which takes a parameter named `e`. This function will be another event listener, so the first argument passed will be the browser event – `e` is a common name for this parameter.

Step 69:

You will be attaching this function to the `submit` event of the form. The `submit` event is triggered when the form is submitted. The default action of the `submit` event is to reload the page. You need to prevent this default action using the `preventDefault()` method of your `e` parameter.

Add a line to your `calculateCalories` function that calls the `preventDefault()` method on the `e` parameter. Then, reset your global error flag `isError` to `false`.

Step 70:

Your function needs to get the values from the entries the user has added.

Declare a `breakfastNumberInputs` variable, and give it the value of calling `document.querySelectorAll()` with the selector `#breakfast input[type='number']`. This will return any `number` inputs that are in the `#breakfast` element.

Step 71:

Using that same syntax, query your number inputs in the #lunch element and assign them to lunchNumberInputs.

Step 72:

Following the same pattern, query for your `number` inputs in the `#dinner`, `#snacks`, and `#exercise` elements. Assign them to variables following the naming scheme of the previous two.

Step 73:

Now that you have your lists of elements, you can pass them to your `getCaloriesFromInputs` function to extract the calorie total.

Declare a `breakfastCalories` variable, and assign it the result of calling `getCaloriesFromInputs` with `breakfastNumberInputs` as the argument.

Step 74:

Now declare a `lunchCalories` variable, and give it the value of calling `getCaloriesFromInputs` with your `lunchNumberInputs`.

Step 75:

Following this same pattern, declare variables for the `number` inputs in the `#dinner`, `#snacks`, and `#exercise` elements. Assign them the appropriate `getCaloriesFromInputs` calls.

Step 76:

You also need to get the value of your `#budget` input. You already queried this at the top of your code, and set it to the `budgetNumberInput` variable. However, you used `getElementById`, which returns an `Element`, not a `NodeList`.

A `NodeList` is an array-like object, which means you can iterate through it and it shares some common methods with an array. For your `getCaloriesFromInputs` function, an array will work for the argument just as well as a `NodeList` does.

Declare a `budgetCalories` variable and set it to the result of calling `getCaloriesFromInputs` – pass an array containing your `budgetNumberInput` as the argument.

Step 77:

Your `getCaloriesFromInputs` function will set the global error flag `isError` to `true` if an invalid input is detected. Add an `if` statement to your `calculateCalories` function that checks the truthiness of your global error flag, and if it is truthy then use `return` to end the function execution.

Step 78:

It is time to start preparing your calculations. Start by declaring a `consumedCalories` variable, and assign it the sum of `breakfastCalories`, `lunchCalories`, `dinnerCalories`, and `snacksCalories` (note that order matters for the tests). Be sure to do this after your `if` statement.

Step 79:

Now declare a `remainingCalories` variable, and give it the value of subtracting `consumedCalories` from `budgetCalories` and adding `exerciseCalories`.

Step 80:

You need to know if the user is in a caloric surplus or deficit. A caloric surplus is when you consume more calories than you burn, and a caloric deficit is when you burn more calories than you consume. Burning as many calories as you consume is called maintenance, and can be thought of as a surplus or deficit of `0`, depending on your goals.

Declare a `surplusOrDeficit` variable. Then use a ternary operator to set `surplusOrDeficit` to the string `"Surplus"` or `"Deficit"` depending on whether `remainingCalories` is less than `0`. If it is less than `0`, then `surplusOrDeficit` should be `"Surplus"`. Otherwise, it should be `"Deficit"`.

Step 81:

You need to construct the HTML string that will be displayed in the `output` element. Start by assigning an empty template literal to the `innerHTML` property of the `output` element on a new line at the end of the function.

Step 82:

When you need to lower case a string, you can use the toLowerCase() method. This method returns the calling string value converted to lower case.

Example Code:

```
const firstName = 'JESSICA';

console.log(firstName.toLowerCase()); // Output: jessica
```

Your `output.innerHTML` string will need a `span` element. Create that, and give it a `class` attribute set to the `surplusOrDeficit` variable. Your `surplusOrDeficit` variable should be converted to lower case using the `toLowerCase()` method.

Do not give your span any text yet.

Step 83:

Give your span the text remainingCalories Calorie surplusOrDeficit, using interpolation to replace remainingCalories and surplusOrDeficit with the appropriate variables.

Step 84:

When the user has a calorie surplus, the remainingCalories value will be negative. You don't want to display a negative number in the result string.

Math.abs() is a built-in JavaScript method that will return the absolute value of a number.

Example Code:

```
const num = -5;

Math.abs(num); // 5
```

In your span text, wrap your remainingCalories reference in Math.abs() to ensure that the value is positive.

Step 85:

After your span element, add an hr element to create a horizontal line.

To keep your code clean and readable, you should add this on a new line in the template literal.

Step 86:

Now create a `p` element with the text `budgetCalories Calories Budgeted`, using interpolation to replace `budgetCalories` with the appropriate variable.

This should come after your `hr` element.

Step 87:

Using the same interpolation syntax, add a second `p` element with the text `consumedCalories Calories Consumed` and a third with the text `exerciseCalories Calories Burned`. Remember to replace your `consumedCalories` and `exerciseCalories` variables with the appropriate values.

Step 88:

Finally, you need to make the `#output` element visible so the user can see your text. Your `output` variable is an Element, which has a `classList` property. This property has a `.remove()` method, which accepts a string representing the class to remove from the element.

Example Code:

```
const paragraphElement = document.getElementById('paragraph');

paragraphElement.classList.remove('hide');
```

Use the `.remove()` method of the `output` variable's `classList` property to remove the `hide` class. Don't forget to place the word `hide` inside quotes.

Step 89:

If you click on your `Calculate Remaining Calories` button, you'll see that nothing happens. You still need to mount the event listener.

Add an event listener to your `calorieCounter` element. The event type should be `submit`, and the callback function should be `calculateCalories`.

Step 90:

Your final feature to add is the ability for a user to clear the form. Start by declaring an empty function called `clearForm` – it should not take any arguments.

Step 91:

You need to get all of the input containers. Declare an `inputContainers` variable, and assign it to the value of querying the document for all elements with the class `input-container`.

Step 92:

Remember that `document.querySelectorAll` returns a `NodeList`, which is array-like but is not an array. However, the `Array` object has a `.from()` method that accepts an array-like and returns an array. This is helpful when you want access to more robust array methods, which you will learn about in a future project.

The following example takes a `NodeList` of `li` elements and converts it to an array of `li` elements:

Example Code:

```
<ul>

  <li>List 1</li>
```

```
  <li>List 2</li>
  <li>List 3</li>
</ul>
```

Example Code:

```
const listItemsArray = Array.from(document.querySelectorAll('li'));

console.log(listItemsArray); //Output: (3) [li, li, li]
```

Wrap your `inputContainers` query selector in `Array.from()`. Do this on the same line as your declaration.

Step 93:

It is time for another loop. Create a `for...of` loop with a variable called `container` to iterate through the `inputContainers` array.

Inside the loop, set the `innerHTML` property of the `container` to an empty string. This will clear all of the contents of that input container.

Step 94:

After your loop completes, you need to clear the `budgetNumberInput`. Set the `value` property of `budgetNumberInput` to an empty string.

Step 95:

You also need to clear the `output` element's text. You can do this by setting the `innerText` property to an empty string.

The difference between `innerText` and `innerHTML` is that `innerText` will not render HTML elements, but will display the tags and content as raw text.

Step 96:

To finish off this function, you need to restore the `hide` class to the `output` element. The `classList` property has an `.add()` method which is the opposite of the `.remove()` method. It accepts a string representing the class to add to the element.

Add the `hide` class to your `output`.

Step 97:

To complete this project, add an event listener to the `clearButton` button. When the button is clicked, it should call the `clearForm` function.

Congratulations! Your project is complete.