Lesson Thursday

Introduction to Programming (/introduction-to-programming) / JavaScript and Web Browsers (/introduction-to-programming/javascript-and-web-browsers) / Calculator with Forms and Branching

Text | Cheat sheet

The last time we added to our calculator project, we were using `window` methods to prompt and alert information from and to the user. We also carefully separated our business and user interface logics. The functionality of our calculator was limited to adding, and in the following practice prompt, you expanded the functionality with your pair by adding subtraction, multiplication, and division.

Now that we are familiar with new form input types, event listeners, and branching, it's time to update our calculator app to modern standards. In this lesson, we'll refactor our calculator app to implement the following:

- A single form that uses basic inputs for the numbers, and radio buttons with options to pick between adding, subtracting, multiplying, and division.
- Event listeners to handle events, both when our webpage finishes loading all resources and when the user submits the form.

- Branching to determine which math function to run based on the user's selections in the form.
- Separation of business and user interface logics.

This example will be more complex than any of the previous examples we've worked with! Take your time and discuss any points of confusion that come up. You are welcome to code along with this lesson, or just read through it. In the next practice prompt, you'll have the opportunity to recreate the calculator project.

Consolidating Multiple Options into One Form

Our new, all-inclusive form should look something like this:

calculator.html

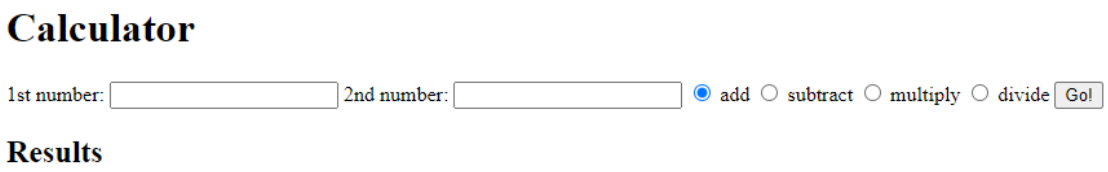
```
<!DOCTYPE html>
<html lang="en-US">
<head>
  <title>Calculator</title>
  <link href="css/styles.css" rel="stylesheet" type="text/c
ss">
  <script src="js/scripts.js"></script>
</head>
<body>
  <h1>Calculator</h1>
  <form id="calculator">
    <label for="input1">1st number:</label>
    <input id="input1" type="text">
    <label for="input2">2nd number:</label>
    <input id="input2" type="text">

    <label>
      <input type="radio" name="operator" value="add">
      add
    </label>
    <label>
      <input type="radio" name="operator" value="subtract">
      subtract
    </label>
    <label>
      <input type="radio" name="operator" value="multiply">
      multiply
    </label>
    <label>
      <input type="radio" name="operator" value="divide">
      divide
    </label>

    <button type="submit" class="btn">Go!</button>
  </form>

  <h2>Results</h2>
  <p id="output"></p>
</body>
</html>
```

In the above HTML, we've given the form an id of `calculator` and the first two inputs have the ids `input1` and `input2`. This is where we'll get the user's 1st and 2nd numbers.

We've also added radio buttons to allow users to choose what operation to perform on the two numbers they provide.

Below the form, there's a "Results" section with an empty P tag. This is where we'll print the results from the calculation.

Now, if we launch our HTML page in the browser we should see our two inputs and radio buttons:



Our website isn't very pretty or well-formatted right now. If you want, you can add styling later.

Updating Scripts

Now that we've finished the changes to our HTML, we need to update our scripts. Let's start with a basic update to set up event handlers for the window load event and the form submission event.

js/scripts.js

```
// Business Logic
function add(num1, num2) {
  return num1 + num2;
}

function subtract(num1, num2) {
  return num1 - num2;
}

function multiply(num1, num2) {
  return num1 * num2;
}

function divide(num1, num2) {
  return num1 / num2;
}

// User Interface Logic
function handleCalculation(event) {
  event.preventDefault();
  // the code to get and process form values will go here!
}

window.addEventListener("load", function() {
  const form = document.getElementById("calculator");
  form.addEventListener("submit", handleCalculation);
});
```

Notice that we've used an event listener with an anonymous function expression as the callback for the window's load event, and an event listener with a function declaration (`handleCalculation()`) as the callback for the form's submission event. The `handleCalculation()` reacts to the form submission event, so we need to include an `event` parameter and call `event.preventDefault();` within it.

Our scripts are still separated between business and user interface logic. User interface logic is any code that handles accessing, manipulating, or updating the DOM. Business logic is any JavaScript

code that handles processing data. Business logic never accesses, manipulates or updates the DOM.

Next up, let's add functionality to the `handleCalculation()` function to get the form values. We'll just look at the `handleCalculation()` function for the next few code snippets, because that's the only code we'll be updating. The ellipses `...` indicate omitted code.

js/scripts.js

```
...

// User Interface Logic
function handleCalculation(event) {
  event.preventDefault();
  const number1 = parseInt(document.querySelector("input#in
put1").value);
  const number2 = parseInt(document.querySelector("input#in
put2").value);
  const operator = document.querySelector("input[name='oper
ator']:checked").value;
}

...
```

With the three new lines of code we've added, we now have the two user-inputted numbers saved in the `number1` and `number2` variables, as well as the selected operator from the radio buttons saved in the `operator` variable. Notice that we've made sure to `parseInt()` the numbers from the form.

Testing and Debugging with `console.log()`

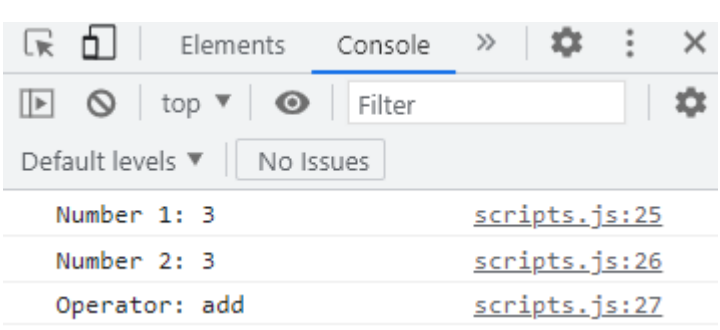
Before we continue on to adding more logic, now is a good time to test our code and verify that we're correctly grabbing the form values. We can easily do this with `console.log()`. Let's add the

following instances of `console.log()` to the `handleCalculation()` function:

js/scripts.js

```
...

// User Interface Logic
function handleCalculation(event) {
  event.preventDefault();
  const number1 = parseInt(document.querySelector("input#in
put1").value);
  const number2 = parseInt(document.querySelector("input#in
put2").value);
  const operator = document.querySelector("input[name='oper
ator']:checked").value;
  console.log("Number 1: " + number1);
  console.log("Number 2: " + number2);
  console.log("Operator:", operator);
}

...
```

Notice that we've added a description to each variable that we are logging. We can combine strings and variables with a plus sign `+` or a comma `,`. If we run this project, we'll get positive confirmation that we're correctly grabbing the form values, just like in the image below. Note that if you are coding along with this lesson, your logged values may be different based on the numbers and operation you've inputted.



Using `console.log()` in this manner is a great debugging approach. If our form were not functioning correctly, or if our calculator was providing us odd, unexpected results (such as `NaN`) we could use `console.log()` to double-check that our values are being retrieved. And, if they weren't, we could pinpoint *which* value is causing the issue, and double-check that we're retrieving this value with the correct HTML id.

Make sure to remove all `console.log()` statements from your code after you are done debugging. Leaving behind `console.log()` statements makes your code look unpolished and buggy. If you leave in `console.log()` statements in your independent projects, we will ask to your resubmit and remove them.

Next up, we'll need to add branching to our project.

Implementing Branching and Printing the Result

Now that we've confirmed we're successfully collecting form input values, let's add branching to call the appropriate function based on the user's selected radio button:

js/scripts.js

```
...

// User Interface Logic
function handleCalculation(event) {
  event.preventDefault();
  const number1 = parseInt(document.querySelector("input#in
put1").value);
  const number2 = parseInt(document.querySelector("input#in
put2").value);
  const operator = document.querySelector("input[name='oper
ator']:checked").value;

  let result;
  if (operator === "add") {
    result = add(number1, number2);
  } else if (operator === "subtract") {
    result = subtract(number1, number2);
  } else if (operator === "multiply") {
    result = multiply(number1, number2);
  } else if (operator === "divide") {
    result = divide(number1, number2);
  }

  document.getElementById("output").innerText = result;
}

...
```

Here, we've simply added an `if...else` statement that calls different methods depending on what radio button the user has selected. For instance, if they select "divide", we run our `divide()` function; if they select "subtract", we run our `subtract()` function, etc. Regardless of which operation they choose to perform, we still insert the answer onto our page with the same line, `document.getElementById("output").innerText = result;`.

Notice in the code above that we declared the variable `result` *outside* the `if...else` statement, but we did not immediately give it a value. We simply told JavaScript that it *exists*. It will have the value

of `undefined` until we define its value depending on which branch we're following.

Then, after the `if...else` statement, we print the value of `result` as the text of the output P tag. Notice that we're making use of JavaScript's implicit data type coercion: `result` is a number and the `innerText` property expects a string as an argument. In this case, we don't need to explicitly convert the `result` variable into a string first with `result.toString()`. But we can if we prefer to. Relying on JavaScript's data type coercion can lead to bugs, however, it is common to see out there, so it's important to become familiar with it.

By the way, you might be wondering why there is no `else` condition in the code above. Remember that anything that our other statements don't cover will end up in the `else` condition. If you don't have an `else` condition, it just means the code will move on. We could use the `else` condition for error handling, returning something like `"Error – unrecognized operator"`. However since the radio buttons don't allow the room for user error, this error handling would be unnecessary.

Keep in mind, though, that we *wouldn't* want to use an `else` statement for something like division, though. `else` statements should be used to handle errors and inputs our code doesn't recognize.

Completed Scripts

To see the completed JS for the Calculator website, visit the cheat sheet for this lesson.

More on Business and UI Logic

Organizing our code into user interface (UI) and business logic is a standard practice in writing code that's reusable and well organized. We've seen this at play in the refactor we just completed!

Because all of our business logic was already separated into their own functions, when it came time to update our scripts to use event handlers and branching, all we had to do was focus on our UI logic.

It may be hard to see the advantage of separating code between UI and business logic, given that our calculator functionality is so simple. However, this separation is truly foundational to writing code that's easy to read, reuse, and refactor as your applications get larger.

Another Configuration for our Scripts' UI and Business Logics

Let's look at another way we can organize our UI and Business Logics. Before you read through the following updated scripts, take note that the following separation of logic isn't superior to the previous example. It's just another way to organize your scripts, and it's meant to give you ideas of how you can create functions to organize your code. Ultimately, you should choose the separation of UI and business logic that is most comfortable for you right now. There will be a lot of opportunity to practice code organization in the coming weeks.

```
// Business Logic
function add(num1, num2) {
  return num1 + num2;
}

function subtract(num1, num2) {
  return num1 - num2;
}

function multiply(num1, num2) {
  return num1 * num2;
}

function divide(num1, num2) {
  return num1 / num2;
}

function calculate(num1, num2, operatorParam) {
  if (operatorParam === "add") {
    return add(num1, num2);
  } else if (operatorParam === "subtract") {
    return subtract(num1, num2);
  } else if (operatorParam === "multiply") {
    return multiply(num1, num2);
  } else if (operatorParam === "divide") {
    return divide(num1, num2);
  }
}

// User Interface Logic
function handleSubmission(event) {
  event.preventDefault();
  const number1 = parseInt(document.querySelector("input#in
put1").value);
  const number2 = parseInt(document.querySelector("input#in
put2").value);
  const operator = document.querySelector("input[name='oper
ator']:checked").value;

  let result = calculate(number1, number2, operator);
```

```
  document.getElementById("output").innerText = result;
}

window.addEventListener("load", function() {
  const form = document.getElementById("calculator");
  form.addEventListener("submit", handleSubmission);
});
```

With the new organization, we've made the following changes:

- We've moved the branching into its own function called `calculate()`. Because the branching doesn't handle updating or accessing the DOM, we can separate it into a separate function in the business logic section of our scripts.
- `calculate()` has 3 parameters, one for each number (`num1` and `num2`), and one for the operator (`operatorParam`).
- In each condition block within `calculate()`, instead of updating the value of the `result` variable, we simply return the value of calling the appropriate add/subtract/etc function. For example: `return add(num1, num2);`. This same code could be rewritten on two lines like so:

```
function calculate(n1, n2, op) {
  if (operatorParam === "add") {
    let result = add(n1, n2);
    return result;
  }
  ...
}
```

- We've also updated the handler function for the form's submission event (originally `handleCalculation()`) to use a more generic name: `handleSubmission()`. Now this method calls on the new `calculate()` function in the line of code `let result = calculate(number1, number2, operator);`, and prints the result to the DOM just like before.

Again, this organization of user interface and business logics is not better than the previous example. What this refactored example shows us is how we can use functions to group code together. If some of this new code organization is hard to wrap your head around, it's completely okay to not understand it or feel uncomfortable organizing your code in this way. In another two weeks of examples and practice, this sort of organization *will* begin to feel comfortable. So for now, talk to your pair and instructor about your thoughts and questions, and do your best to include a basic separation of UI and business logics in your scripts.
Separating UI and business logic will not be required on this section's independent project.

Previous (/introduction-to-programming/javascript-and-web-browsers/form-input-types)
Next (/introduction-to-programming/javascript-and-web-browsers/practice-calculator-and-more)