JavaScript Labs

Contents

Quick Lab 13 - JavaScript Types

Objectives

- To understand how types work in JavaScript

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/13_JavaScriptTypes/starter/** folder.
2. Add code to declare a number and a console.log to output it:

```
let numTest = 45.324568;
console.log(numTest);
```

Press **F12** to see the developer tools and choose the **Console** tab. You should see the value of the **numTest** variable displayed.

You have created a Number data type object in the program stack, this is a 64bit number. Next, we will explore the Number type a little more with some of its methods.

We will create a new variable called **twoDecimalPoints** and use a method to truncate the number.

3. Under the last code you wrote in **index.js**, add the following code:

```
let twoDecimalPoints = numTest.toFixed(2);
console.log(twoDecimalPoints);
```

4. Save the file and your browser should automatically refresh to display the value of **twoDecimalPoints** as **45.32**. Notice that this number is now BLACK not BLUE? The toFixed function converts the number to a string!
5. Under your last line of code in **index.js**, create a **stringTest** variable, as shown in the code segment below.
   (Please be sure to add the text exactly as it appears; otherwise, the notes will not match up to what you will see in the console.)

```
let stringTest = `I am the very model of a modern major general`;
let indexOfM = stringTest.indexOf(`m`);
console.log(indexOfM);
```

6. Save the code and observe the browser console.

You will see a value of **3**, examine the string and you will see that the **m** is the *fourth* character, so there are *three* characters before the first **m**.

7. Change the **m** within the **indexOf** method call to a capital **M**, save and observe the output in the browser again.

This time, the **console.log** will return a **-1** value. The **-1** value is telling us that there is *no match within the string* at all, proving that string searches are case sensitive.

What if we convert the string to upper case?

8. Before the **indexOfM** line, add the following code:

```
stringTest = stringTest.toUpperCase();
```

9. Save and observe the output in the browser again.

The output will, once again, give a value of **3**. Behind the scenes, the string is an indexed collection of characters and the search function is making its way through the letters character by character until it makes a match.

With that concept in mind, we will use the principals to learn how to slice a string.

10. Add the following code under the last line, then capture **start** and **end** in a **console.log**, save and observe the output.

```
let start = stringTest.indexOf("MODEL");
let end = stringTest.lastIndexOf('MAJOR');
```

This time, we have matched based upon words, but you could search for file paths or extensions; for instance, if we were reading from a form.

The two integer values held can be used to create a substring from the longer one using string's substring method.

11. Add the following lines of code to the end of your code, save and observe the output.

```
let subStr = stringTest.substring(start, end);
console.log(subStr);
```

The console should now return a value of **"MODEL OF A MODERN"**.

Let's finish up this exercise by writing this content to the browser window using the **document.write** method (we will look at the document object in more depth later on).

12. Add the following lines of code to the end of your code, save and observe the output.

```
document.write("<p>" + subStr + "</p>");
```

We have used an operator here: the + sign, which we have used to concatenate the string together and mix our string value with some hard-coded HTML, to create new content for the page. With that done, let's learn some more about operators.

This is the end of Quick Lab 13

Quick Lab 14 - JavaScript Operators

Objectives

- To understand how operators work in JavaScript.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/14_JavaScriptOperators/solution/** folder.

The browser window will open, but there will be nothing to view!

2. Examine the following operations and write the result you expect before you look at any code output!

Arithmetic operators

Operation	**Result**
console.log(5 + 5);	
console.log(5 * 10);	
console.log(10 % 3);	
console.log(5 + 10 / 2 * 5 - 10);	
console.log((6 + 10) / 2 * 5 - 10);	

3. Uncomment the Arithmetic Operators section of the code - highlight the required lines to uncomment and press **CTRL + /** (or **CMD + /** on MacOS).
Save the file and check the console output against what you expected.

4. Repeat for assignment operators, assuming x is initialised as 0 and the statements are processed sequentially.

Assignment operators

Operation	**Result**
console.log(x = x + 1);	
console.log(x+= 1);	
console.log(x++);	
console.log(++x);	

5. Now, we will move on to relational operators. Every expression will evaluate as either true or false.

Relational operators

Operation	**Result**
console.log(5 > 3);	
console.log(3 != 3);	
console.log(3 <= 2 && 5 >2);	
console.log(!5>3);	

6. Finally, we will explore what happens with mismatched types.

Mismatched types

Operation	**Result**
console.log(5 + "5");	
console.log(5 + true);	
console.log(5 * "5");	
console.log(1 == true);	
console.log(1 === true);	

This is the end of Quick Lab 14

Quick Lab 15a - JavaScript Conditionals

Objectives

- To investigate the JavaScript flow of conditional statements.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/15a_JavaScriptConditionals/starter** folder.

The browser window will open, but there will be nothing to view.

2. Declare a variable called **age** and initialise it to be **15**.
3. Enter the following code to create the if below the variable declaration:

```
if (age <= 17) {
            console.log("Underage");
} else {
            console.log("18 or over");
}
```

4. Save the page and observe the output on the console of the developer tools (F12).
5. Change the value to **42** and check that the output has updated.

We now have a simple **if** statement in place. Now it's time to take our code a step further and add an **else if** caveat into our code. The **else if** is a further, logical check delivering a Boolean value. Additional checks are examined in order; when a **true** is evaluated, the program leaves the **if** statement, so the order is important.

6. The code from the previous part has been slightly amended and now needs to be included under the comment for Part 2.

```
if (age <= 17) {
            console.log("Underage");
} else if (/*Remove me and insert your code here*/) {
            console.log("Insurable");
} else {
            console.log("out of range");
}
```

The **else if** statement is going to check if the age variable is between **18** and **65,** so:

do we need to use an **and** statement to achieve this, or will a simple check for **<=65** suffice?

7. Replace the code in the **else if** statement to achieve the desired result. Refer back to your notes or ask your instructor for help if you get stuck.
8. Save the page and observe the output in the browser.

9. Test that your code works by setting age to the following values:

- 10
- 50
- 80

10. Use a ternary statement to achieve the same results that were gained in parts 6-8.

This is the end of Quick Lab 15a

Quick Lab 15b - JavaScript Loops

Objectives

- To investigate JavaScript looping statements.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/15b_JavaScriptLoops/starter/** folder.

The browser window will open, but there will be nothing to view.

The **for** loop has three arguments: the **counter**, the **condition**, and the **iterator**. You are going to code in a simple **for** loop where the following properties need to be set:

parameter	value
counter	Variable name **i** set to **1**
condition	**i** is less than **10**
iterator	Each loop must *add* **1** to the value of **i**

2. Enter the following code, amended appropriately to achieve this:

```
for (counter; condition; iterator) {
            console.log(i);
}
```

3. How many times do you expect the loop to execute? _____
4. Save the file and observe the output in the browser to check your assumptions.
5. Write a while loop that has the following rules:

parameter	value
initial conditions	Variable name **x** set to **2** and **loopCounter** set to **0**
condition	**x** is less than **10000**
iterator	Each loop must **square** the value of **x** and *add* **1** to **loopCounter**
action	Each loop must log out the value of **x** and **loopCounter**

6. Once you have resolved this, save the code and observe the output in the browser.

Further activities

Only attempt the further activities if there is enough time remaining. Your instructor can help you with this if you need it, but we hope you can start ranging out on your own.

7. Alter the for loop you created in Step 8 of the exercise to count down rather than up.

This is the end of Quick Lab 15b

Quick Lab 16 - JavaScript Arrays

Objectives

- To investigate JavaScript arrays and their functions.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/16_JavaScriptArrays/starter** folder.

The browser window will open, but there will be nothing to view.

2. Declare an array called **quote** that contains four *strings*, "**I**", "**am**" "**your**" and "**friend**".
3. Log the array to the console.
4. Save the file and observe the browser to check the output - *you should see details of an array and expanding it will show the values and their indexes.*
5. Access the index of the array that contains the string "your" and log the array element to the console.

```
console.log(quote[2]);
```

6. Save the file and observe the browser to check the output.
7. Using the **pop** function, remove the string "**friend**" from the end of the array.
8. Using the **push** function, add the string "**father**" to the end of the array.
9. Log the array to the console again.
10. Save the file and observe the browser to check the output.
11. Use the **unshift** function to add the string "**Luke**" to the start of the array.
12. Log the array to the console again.
13. Save the file and observe the browser to check the output.

There are two things wrong with the output. The first is that it the string is concatenated by commas, and the second is that the 'quote' is actually a misquote!

We're going to generate an output in a different way by looping through the array and creating a new string. We're also going to fix the misquote by detecting the erroneous word in the array and replacing it with the correct word! Let's do the latter first:

To do this, we are going to detect if indeed the erroneous word is in the array. If it is, we are going to find the index that the word is at, then use this information to replace that index with the correct word.

14. Declare a variable called **erroneousWord** and set it to a string with the misquoted word from the array (it's **Luke** if you didn't know).
15. Set a variable called **lukeIsHere** using the **find()** function to see if the quote array contains the **erroneousWord**. The code is:

```
let lukeIsHere = quote.find(n => { return n === erroneousWord});
```

The syntax inside the find function will feel a little alien at the moment, but go with it as it's explained later in the course.

16. Declare a variable called **lukeIsAt** without assigning it.
17. If **lukeIsHere** has been set to **true**, find the index that the **erroneousWord** sits at using the **findIndex()** function and set **lukeIsAt** to the value of the **index**. The code is:

```
lukeIsAt = quote.findIndex(n => { return n === erroneousWord});
```

18. Still inside the **if** block, use the value of **lukeIsAt** to set that index in the quote array to the string **"No"**.

```
if (lukeIsHere) {
    lukeIsAt = quote.findIndex(n => {
        return n === erroneousWord
    });
    quote[lukeIsAt] = "No";
}
```

19. Log out of the array and ensure that the expected result is outputted in the browser.

Extension

To sort out the display of the quote, you need to create a new string, then loop through the array. You need to add a space into the string after each word, apart from the first word, (to which you'll append a comma and a space) and the final word, (to which you'll append an exclamation mark).

20. Declare a variable called **output** and set it to be an empty string.
21. Create a **for** loop that:

- Loops through the quote array.
- Executes when the loop counter is less than the length of the array.
- Adds an exclamation mark to the **output** string, if we are at the last element in the array.
- Adds a comma and a space to the **output** string, if the current element is 'No'.
- Otherwise adds a space to the **output** string.

```
for (let i = 0, j = quote.length; i < j; i++) {
    if (i === j - 1) {
        output += quote[i] + '!';
    } else if (quote[i] === 'No') {
        output += quote[i] + ', ';
    } else {
        output += quote[i] + ' '
    }
}
```

22. Log out the **output** string.

23. Save the file and then check your browser output to ensure that the correct quote is displayed.

```
No, I am your father!
```

This is the end of Quick Lab 16

Quick Lab 17 - JavaScript Functions

Objectives

- To investigate JavaScript functions.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/17_JavaScriptFunctions/starter** folder.

The browser window will open, but there will be nothing to view!

Part 1 – Defining and using functions and understanding scope

In this part of the exercise, you will practise writing a function to perform some actions and investigate the scope of the variables within it.

In addition, you will look at the scope of variables - notice that the script already contains an array of objects, that contain some details about different films.

2. Under the comment for Part 1, declare a function called **findMovie** that takes an argument called **movieTitle**.
3. In the *body of the function,* create a **for...of** loop of the **movies** array:
4. The loop body should:

- Check to see if the current *movie title* is the same as the **movieTitle** passed into the function and *if it is*, log out details of the movie in a suitable string;
- Log out the value of **movie** before the loop's closing brace

5. The value of **movie** should be logged before the function closes
6. Call the **findMovie** function with an argument of **"Star Wars"**.
7. Log out the value of **movie**.
8. At this point, save your file and check the output.

The expected outcome is that there is a Reference Error - but which console.log is, or console.logs are, causing it/them?

9. Comment out the offending **console.log(s)** and check your output.

You should see all five movies logged, with the string you wrote for a found movie being outputted before the movie object - for it's logged itself.

Two of the **console.log** statements added produced a Reference Error. This is because of the scope of the variable **movie**. As it's declared as part of the **for...of** loop, its scope is limited to inside the body of this block (i.e., between the { } that immediately follows the for). As long as execution remains inside this loop, the variable **movie** is in scope.

Once the loop finishes and execution returns to the level above (i.e., back to the body of the function) and the variable **movie** is no longer in scope - referring to it in the code will cause the Reference Error.

It follows that if **movie** is not available here, it will also not be available after the line that calls the function has completed execution - again causing a Reference Error.

Note: Because **movieTitle** is part of the function block, it's accessible throughout the execution of the function, including inside any blocks that are used within the function body (i.e., in the **for** and **if** blocks).

Note: Because the const **movies** is declared at script level (i.e., inside the script tag) and at the top of it, it's available to all blocks of code that live inside this script tag.

10. Under the last line, define a variable called **movie** set to the value of **"Thor: Ragnorok"**.
11. *Uncomment* the **console.logs** of movies and *add another log* of the value of **movie** under the declaration of the variable from Step 9.
12. Observe the results.

What you should see this time is two undefined values.

13. In **index.html**, comment out the first script tag and uncomment the second.
14. Save and reload the page.

You will notice that Reference Errors as before are present.

15. Change the declaration of **movie** to have the **var** keyword in front of it (rather than **let**) and make sure that all **console.logs** are *uncommented*.
16. Observe the results.

What you should see this time is two undefined values again. The differences are all to do with concepts called hoisting and 'temporal dead zones'. More details of which can be found, with a good explanation of **let** at:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

This error was not shown when working in the module due to the scope rules creating a new scope for each module.

17. Swap the comments around the script tags in index.html again.
18. Back in index.js, add a call to the **findMovie** with the argument set to **movie**, save and observe the output.

There is no output from the function call for findMovie - using the defined movie variable. This is because there is no movie in the movies array with the title **"Thor: Ragnorok"** and therefore, the loop completes without ever entering the **if** condition.

Extension

Part 2 – Creating functions that return data

It's rare for functions to just execute code and for the program to continue. Usually, a function will manipulate some data then return some data, which can then be used further. This part of the exercise will allow you to experiment with returning data from a function.

19. Comment out the section of code for Part 1, *leaving the* **movies** *array intact* and work under the comment for Part 2.
20. Declare a function called **returnMovie** that takes **movieTitle** as an argument and has a function body that:
21. Uses a **for...of** loop on the **movies** array with a loop body that:
22. Checks to see if the **title** property of the current **movie** matches the **movieTitle** supplied to the function;
23. *If it does,* it should simply **return** the current **movie**;
24. Logs out the current value of **movie**;
25. Logs out **"Any text, any text at all"**.
26. In the body of the script, declare a variable called **myMovie** and set it to the result of calling **returnMovie** with an argument of **"Avengers: Infinity War"**.
27. Log out the value of **myMovie**, save and observe the output.

If you have created your **returnMovie** function you should observe the following:

- Each of the movies that appear BEFORE the selected movie are logged out - as the loop has executed for each of these movies
- The movies that are AFTER the selected movie are not logged out - because the presence of the return statement stops the execution of the loop and indeed the function (so that "Any text, any text at all" is also not shown)
- The execution 'returns' to its call point with the value of whatever is returned

28. Access the properties of **myMovie** to *produce and log a string* as a sentence with them in it, saving and observing your output.

What happens if we try to pass a movie title that doesn't exist in the movies array into returnMovie? Let's find out!

29. Declare a variable **myOtherMovie** and set its value to a call to **returnMovie** with an argument of **"Thor: Ragnorok"**.
30. Log out the value of **myOtherMovie** and observe the output.

The first thing that we notice is that the whole of the **movies** array has been logged out and the text **"Any text, any text at all"**. This is because the title was not found and the function completed its execution fully and never returned a value... or did it?

The next thing that we notice is that the console.log of **myOtherMovie** has outputted **undefined**. It looks like we've never set the value of **myOtherMovie** - because we haven't! Let's fix that...

31. Comment out the logging of **"Any text..."** and add a line that **returns** the string **Movie not found**.

32. Save and observe the output.

The logging of **myOtherMovie** now outputs **"Movie not found"**.

The code is still not very reusable as if I want to log out the details of a movie, I have to supply the string inside a **console.log**. Also, what happens if it's already a string (because it's a movie not in the array)? Our output would be very messy! Step up another function!

33. Create a function called **myMovieDetails** that takes a variable **anyMovie** as an argument.
34. Check that the typeof **anyMovie** is an 'object' and **return** a suitable string if it is; simply **return anyMovie** if it isn't.
35. Inside a **console.log**, call **myMovieDetails** with an argument of **myOtherMovie**.
36. Observe the results.

It should output: **Movie not found**.

Can we use a function as the argument to another function? Yes, we can!

37. Repeat the last instruction instead passing in **returnMovie** with an argument of **"Jaws"** as the argument to the **myMovieDetails** function.
38. Observe the results.

It should output the details for Jaws in your defined string.

This is the end of Quick Lab 17

Quick Lab 18 - JavaScript Maps

Objectives

- To create a Map in JavaScript

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/18_JavaScriptMaps/starter** folder.

The browser window will open, but there will be nothing to view.

For clarity of instructions, the steps to save and observe the browser have been omitted after each instruction that affects the output.

2. Create a new **Map** object called **hanSolo** and add the following *key/value* pairs to it:

- **vehicle** - **Millenium Falcon**;
- **bff** - **Chebacca**;
- **sweetheart** - **Leia**.

3. Access the properties that you have just declared by logging out the following details:

- The **size** of the map **hanSolo**;
- Han Solo's **vehicle** name (***HINT*** - use the **Map.get()** method);
- If Han Solo has a **sweetheart** (***HINT***: use the **Map.has()** method);
- If Han Solo is a (***has***) **Jedi**.

4. Add another *key/value* pair to **hanSolo** that sets a key **son** to **Ben** and log this new property to the console.
5. Iterate over **hanSolo** using a **for...of** loop that uses both the *key* and the *value* of each pair, logging out each pair.
6. Manipulate the map by:

- *Changing* the value of **bff** to **Luke** and log out **hanSolo**;
- *Deleting* the key/value pair **son** and log out **hanSolo**;
- *Clearing* the Map and logging it out.

This is the end of Quick Lab 18

Quick Lab 19 - JavaScript Objects

Objectives

- To understand how to declare and destructure objects

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/19_JavaScriptObjects/starter** folder.

The browser window will open, but there will be nothing to view.

For clarity of instructions, the steps to save and observe the browser have been omitted after each instruction that affects the output.

2. Create a new **Object** called **darthVader** and add the following *key/value* pairs to it:

- **allegiance** - **Empire**;
- **weapon** - **lightsabre**;
- **sith** - **true** (Boolean value).

3. Access the properties that you have just declared by logging out the following details:

- DarthVader's **allegiance**;
- Darth Vader's **weapon**;
- If Darth Vader is a **sith**;
- The value of **Jedi** from Darth Vader; (even though it'sn't defined in the object)
- The number of properties Darth Vader has (see the line of code below for this)

```
console.log(Object.keys(darthVader).length);
```

Quick explanation - Object.keys is a function that takes an object and returns an array of the keys in it. By appending .length to it, we return the number of keys in the object.

4. Add *key/value* pairs to **darthVader** that:

- Set a key of **children** to **2**
- Set a key of **childNames** to the array **['Luke', 'Leia']**;
  then log the **children** property and the *value of the first element* in the **childNames** array.

5. Iterate over **darthVader** using a **for...in** loop that uses both the *key* and the *value* of each pair, logging out each pair.
6. Manipulate the object by:

- *Changing* the value of **allegiance** to **The light side** and log out **darthVader**
- *Deleting* the key/value pair **children** and log out **darthVader**

Hint: use the code below:

```
delete darthVader.children;
```

- *Destructuring* the object, setting a variable for each of the keys in the object to the corresponding value in the object:

```
let{allegiance, weapon, sith, childNames} = darthVader;
```

7. Console each individual variable out to ensure that they have been set.
8. *Clearing* the object and logging it out.

This is the end of Quick Lab 19

Quick Lab 20 - JavaScript and the DOM

Objectives

- To be able to manipulate the DOM by creating and adding content.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/20_JavaScriptDOM/starter** folder.

The browser window will open, but there will be nothing to view.

2. Under the array declaration, create a function called **buildP** that takes *1 parameter* called **placeholder.**

The parameter placeholder will hold a reference to the DOM element which we want to append new content to. The DOM programming mechanism allows us to create new elements by type using the **createElement** function. Any new elements have to be added to the DOM or it will not display on the page, and it will not have any content, unless we give it some! When DOM programming, it's important to work in the correct order:

3. Create the new element
4. Create any text nodes that need to be attached to the new element;
5. Attach the text nodes to the new element;
6. Attach the new element to the appropriate existing DOM element;

7. Inside the function, declare a variable called **p** and set it to a **new element** of type **p** using the **createElement** function:

```
document.createElement('p')
```

8. Create a *text node* to attach to the new paragraph called **text** with *any string you want in it,* using the line of code below (the text given is an example):

```
document.createTextNode(`Have you tried turning it off and back on
again?`)
```

9. Append the *text node* to the paragraph **p** using the **appendChild** function.
10. Append the paragraph **p** to the **placeholder** that was passed into the function.

The function is now complete and ready to go. All we need to do is call it and pass in the correct parameters. The new **<p>** element we are going to create needs to be appended to a container in the page. This could be another element on the page or even the document root itself. We will add it to the ** ** found in **index.html.**

11. Under the function definition, call the function setting the placeholder argument with the element found by using:

```
document.querySelector('#placeholder')
```

> **When we created the buildP function, we added a second parameter to the function that we have not used, or even set in the function so far - this is one of JavaScript's nice features. You can either add none, the first, or all parameters.**

12. Save your code and refer to your browser.

You should see that the text and paragraph you created with JavaScript are now part of the page. Verify this by inspecting the elements in the Developer Tools.

Extension

A second parameter of the **buildP** function is going to be used to create and append multiple paragraph elements. We will always want *at least 1 paragraph* but maybe more - time to use the **do...while** loop.

13. Add a *second parameter* called **num** to the arguments for **buildP**.
14. At the top of the **buildP** function, declare a variable **i** initialised to **0**.
15. Surround the rest of the code in the **buildP** function with a **do...while** loop that exits when **i** reaches the **number** supplied to the function (remember to increment **i**).
16. Amend the call to the **buildP** function to add a second argument of any number you choose.
17. Save and refer to your browser, checking the output is as expected.
18. Try a few different values and observe the page.

The final piece in this part of the exercise is to randomise the colours used to display our text in. To do this, we will use the array of colours already defined and a randomly generated value in the range of the indexes of the array.

Math.random gives us a value somewhere between 0 and 1. Indexes of arrays are integer values as are their length. The parseInt function will be used to return the nearest integer to a value which we are going to calculate, using the randomly generated number and the length of the colours array.

19. Before the line that appends the text node to the paragraph, add the following line:

```
p.style.color = colours[parseInt(Math.random() * colours.length)]
```

20. Save and refer to the output of the browser.

> **This is the end of Quick Lab 20**

Quick Lab 21 - JavaScript Events Styles

Objectives

- To be able to add and remove JavaScript events.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/21_JavaScriptEvents/starter** folder.

The browser window will open, but there will be nothing to view except the provided code.

2. Under the comment, add event listeners to the elements, using the functions shown:

ELEMENT	EVENT	HANDLER
BUTTON WITH ID OF TEXTCOLOUR	click	blueToRed
BUTTON WITH ID OF BGCOLOUR	click	greenToPink
BUTTON WITH ID OF FONTS	click	tnrToArial

3. Save the file and make sure that the page functions as it did at the end of the last Quick Lab.
4. Add an event listener to **tnrParagraph** that has an event of **mouseover** and a handler of **mouseOver**.
5. Add **mouseOver** as an arrow function that takes an argument of **event**:

```
const mouseOver = event => {
     // function body here
}
```

6. Populate the function body:
7. Change the *background colour* of the **event target** to **limegreen**;

```
event.target.backgroundColor = 'limegreen';
```

8. Check to see if the **textContent** of the **event target** contains the word **background**

- If it doesn't, add **I have had my background colour changed on mouse over** to it
- If it does, replace the word **out** with **over** in it

9. Save the page and check to see if the new mouse over functionality works.

You should find that it doesn't. The explanation for this is in JavaScript's hoisting rules. Normal variables and functions are hoisted by JavaScript, meaning that it knows about them and their implementation no matter where they are declared. Arrow functions (and Classes - when using Object Oriented JavaScript), are not. Therefore, the page fails silently.

10. Move the registering of the event listener to beneath the declaration of the arrow function.
11. Save the file and you should see that the event now fires correctly.
12. Add another event handling function to **tnrParagraph**, using the previous steps as a template, to change the *background colour* to **yellow** when the *mouse leaves* the element, and *add/change* the text accordingly. Don't forget to register the event on **tnrParagraph**!

The way that these new event handlers have been written means that they are reusable across any elements as they are not tightly bound to the element. When we pass in the **event**, **event.target** gives the reference to the element which fired the event.

13. Add an arrow function called **elementClick**. It should:
    - Receive **event** as an argument;
    - Set the *background colour* of the *element that raised the event* to **white**;
    - Change the **text content** of the *element that raised the event* to **I have no event listeners attached to me now**;
    - **Remove** the **click** event listener on the **button** with **id** of **fonts**;
    - **Remove** the **click**, **mouseover** and **mouseout** event listeners from the *element that raised the event*;
    - Check the **id** of the *element that raised the event* and if it was **tnrParagraph**:
    - Change the **textContent** of **blueParagraph** to **Event listeners enabled**;
    - Add event listeners to **blueParagraph** for **click**, **mouseover** and **mouseout**.
14. If the *element that raised the event didn't* have an **id** of **tnrParagraph** it should:
    - Change the **textContent** of **tnrParagraph** to **Event listeners enabled**;
    - Add event listeners to **tnrParagraph** for **click**, **mouseover** and **mouseout**.
    - *Register* the **click** event and handling function with **tnrParagraph**.
15. Save the file and check that all of the events are fired at the appropriate time. Use the developer tools and add console.logs if you wish to examine further.

This is the end of Quick Lab 21

Quick Lab 22 - JavaScript Form Validation

Objectives

- To be able to use JavaScript to validate fields on a form and intercept form submission.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/22_JavaScriptAndFormValidation/starter** folder and the file **index.html** from the **QuickLabs/13a_JavaScriptAndFormValidation/starter**
2. The browser window will open - click **Submit** and notice the change in the URL.
3. Enter some values and then click **Submit** again - notice that your form data is now part of the URL.
4. In **index.html**, remove **novalidate** from **<form>**.
5. Add the property **required** to the elements named **name**, **email**, and **gender**.
6. Save the file, refresh, and try to **Submit** an empty form.
7. Add a *name* to the input for **Name** and then click **Submit** again.
8. Add a *name* to the input for **Email *(not an email address)*** and click **Submit**.
9. Add **name@email.com** to the input for **Email** and click **Submit**.
10. Choose a *value* from the list for **Gender** and then click **Submit**

Default validation is applied to all fields that have the required property. However, this validation may not be exactly as we want. The form generates a GET request when the Submit button is clicked and this behaviour is often intercepted.

11. In **index.js**, write an arrow function called **formSubmit** that:

- Takes an argument of **event**
- *Stops the generation of the GET request*
- Pops up an **alert** box to say that the *form has been submitted*

12. Register a **submit** event with the **form** element using a handler of **formSubmit**.
13. Save the file and enter data onto the form, then click **Submit**.

This code should have stopped the page from refreshing with a new URL.

14. Add a loop to the **formSubmit** function that outputs the *first 3* **values** of the *array* included in the **event target** generated by the submit.

```
for(let i = 0; i < 3;i++) {
      console.log(event.target[i].value);
}
```

Extension

15. Declare a constant called **nameInput** and set it to be the *element with a name attribute of name*. (Good practice - define all variables and constants at the top of the code).
16. Add an **arrow function** called **validateNameLength** that:

- Receives an **event** as an argument
- Checks the length of the value to see if it's at least 2:

```
if(event.target.value.length < 2) { }
```

- Alerts Name not long enough if it's not and
- Puts the focus back on the input.

```
nameInput.focus();
```

17. Register a **change** event with **nameInput**, calling the function you have just defined.
18. Save the file and enter a single character in the name field, then move on to the Email field. You should get an alert message.
19. Verify that the alert message does not show when the name is at least 2 characters long.

This is the end of Quick Lab 22

Quick Lab 23 - JavaScript Modules

Objectives

- To be able to use JavaScript Modules in development.

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/23_JavaScriptModules/starter/** folder.
2. Install the dependencies using the command:

```
npm i
```

3. Once the installation is finished, run the application by using the command:

```
npm start
```

4. The browser window will open showing the provided form.
5. Create a new file in the **src** folder:
   Circle.js
6. Place any appropriate functions and variables in the files from **index.js**
7. Remember to include **import** statements, where they need to be used, but are not in the file.
8. Remember to include **export** statements, where they are needed externally, to the file.
9. Save all files and check that the validation still functions as before.

This is the end of Quick Lab 23

Quick Lab 24a - Asynchronous JavaScript - JSON

Objectives

- To be able to create a properly formed JSON file.
- To be able to install and run json-server

Activity

1. In **VSCode**, open the file **index.js** from the **Labs/24_AsynchronousJavaScript/starter** folder.

   Install the dependencies using the command:

```
npm i
```

2. Once the installation is finished, run the application by using the command:

```
npm start
```

3. In the **src** folder, create a new file called **reactrangers.json**.
4. Start the file with an opening and closing set of curly-braces:

```
{


}
```

5. Add a key of results with the value of an empty array:

```
{
  "results": [


  ]
}
```

6. Inside this array, add at least 2 objects (separated by a comma) that have 5 key/value pairs (**id** should increment with each). The object should look like the example below:

```
{
  "id": 1,
  "home": "React Rangers",
  "away": "Angular Athletic",
  "homeScore": 2,
  "awayScore": 0
}
```

7. Save the file.

8. In **VSCode**'s terminal window, initialise another terminal by clicking the ✚ button.
9. Install **json-server** globally using:

```
npm i json-server -g
```

10. Ensure that the terminal is pointing to the **src** folder for this exercise, then spin up **json-server** using the command:

```
npx json-server reactrangers.json
```

11. Open your browser at:

```
http://localhost:3000/results
```

12. You should see the data from the file presented on the screen.

This is the end of Quick Lab 24a

Quick Lab 24b - Asynchronous JavaScript - Promises

Objectives

- To understand how Promises work

Activity

1. In **VSCode**, create a new file **promises.js** in the **Labs/24_AsynchronousJavaScript/starter/src** folder.
2. Create a function called **runPromise()** that is **exported** by **default**.
3. Inside the function, declare a variable called **aPromise** that is a new **Promise** whose **constructor** has an *arrow function* that:
   a) Takes **resolve** and **reject** as arguments

```
let aPromise = new Promise((resolve, reject) => {

}
```

   b) Has a function body that:
      i) Declares a variable called **delayedFunc** that is set as follows:

```
…
  let delayedFunc = setTimeout(() => {
    //whether it resolves or rejects is unknown
      let randomNumber = Math.random();
      (randomNumber < 0.5) ? resolve(randomNumber) :
        reject(randomNumber);
  }, Math.random() * 5000); //function will return sometime: 0-5s
…
```

The fact that we have used **setTimeout** here and the final argument **Math.random() * 5000** (which generates a random number between 0 and 1 and multiplies it by 5000) means that the *arrow function* will execute somewhere between 0ms and 5000ms. The arrow function itself generates a random number between 0 and 1 -and the **Promise** is resolved if the number is less than 0.5 and rejects otherwise.

4. Call **aPromise** with a **.then** chain and set **data** to be the *resolved value* and log this out.
5. Add a **catch** block and set **error** to be the *rejected value* and log this out.
6. Save the file and open **index.js**.
7. **import runPromise** and then call it.
8. Save the file and refer to the console of your browser - don't forget that after each refresh you will need to wait up to 5 seconds for the result to show. Refresh the browser several times to satisfy that the Promise resolves and rejects randomly.

This is the end of Quick Lab 24b

Quick Lab 24c - Asynchronous JavaScript - Fetch

Objectives

- To be able to use the Fetch API to be able to send and receive data.

Activity

1. Open the files **getResultsUtils.js** and **formUtils.js** - you will see that some functions have been provided here to allow you to concentrate on the **Fetch** part of the application.
2. In the **src** folder, create a new file called **constants.js** and declare a **const** with the name of **resultsURL** with the *address of your results on* **json-server** as a *string value*. Ensure that this is **exported**.
3. If you look at the page in the browser, you should notice that there is a placeholder for the results but there is nothing displayed.
4. Populate the **getResults** arrow function in the **getResultsUtils.js** file with code that:
   a) **Returns** a **fetch** call to **resultsURL** that in the *first*.**then** block:
      i) Takes an argument of **results** for an arrow function;
      ii) Checks to see if **results.ok** is **true**, returning:

```
results=results.json();
```

5. Otherwise **throws** a **new Error** object with the **message Data not fetched**;
6. In the *second* **.then** block:
   - Takes an argument of **results** for an arrow function;
   - Sets a variable called **reactRangersResults** to **results**;
   - Calls the function **populateResults** with an argument of **reactRangersResults**.
7. In the **.catch** block:
   - Takes an argument of **error** for an arrow function;
   - Logs out the **error message**.
8. Save your work and open **index.js**.
9. Make a call to **getResults**, making sure you **import** it.
10. Save this file and check your browser window - the results should appear, perhaps after a short delay.

To make the form submit the data, another **Fetch** call is needed.

11. Open **formUtils.js** and locate **submitResult**.
12. The function body should:
    i) Declare a variable called **resultToSubmit** and set it to the *stringified version* of **results**:

```
let resultToSubmit = JSON.stringify(result);
```

13. Return a **fetch** call to **resultsURL** (remember to **import** it!) with *a configuration object* that sets:

- **method** to **POST**;
- **body** to **resultToSubmit**;
- **mode** to **cors**;
- **headers** to:

```
"headers": {
  "Content-Type": "application/json"
}
```

14. In the first **.then** block, take **response** as the argument to the arrow function.
15. If **response.ok** is **true**, **alert** that **Data submitted successfully**.
16. Otherwise, **throw** a **new Error** with the **message Something went wrong, please try again**.
17. In the **.catch** block:
18. Take **error** as the argument to the arrow function;
19. **Alerts** the **error message**.
20. Save the file and open **index.js**.
21. Make a call to the function **registerEventListeners** which should be **imported** from **formUtils.js**.
22. Save the file and enter data on the form. You should notice that unless React Rangers (in any case) is one of the teams, the form does not submit. If you successfully enter data and submit, the page refreshes and the new result is displayed along with the others.
23. Check the **reactrangers.json** file you created before - the new result(s) is stored in this file and an id has automatically been added.

This is the end of Quick Lab 24c

Quick Lab 24d - Asynchronous JavaScript - async/await

Objectives

- To be able to use async/await to be able to send and receive data.

Activity

A little bit of preparation first:

- Comment out the functions directly under the Fetch comment in **getResultsUtils.js** and **formUtils.js**.

1. In **getResultsUtils**, immediately under the function you have just commented out, **export** a new **async** function called **getResults** that:
   a) Sets a variable called **results** to **await** a **fetch** call to **resultsURL**
   b) Sets a variable called **reactRangersResults** to **await results.json()**
   c) *Returns* **reactRangersResults**
2. In **index.js**, add a **.then** clause to the call to **getResults** that:
   a) Takes **results** as an *arrow function argument*
   b) Passes **results** to a call to **populateResults** in the function body (don't forget to **import populateResults**!)
3. Add a **.catch** clause to the call to **getResults** that:
   a) Takes **error** as an *arrow function argument*
   b) *Logs out* the **error message**
4. Save all files and check that the results still display as before.
5. In **formUtils.js**, immediately under the function that has been commented out, declare an **async** function called **submitResult** that takes an argument of **result**.
6. The function body should:
   a) Set **resultToSubmit** to the *stringified* version of **result**
   b) Set **response** to **await** the **fetch** call to **resultsURL** with the *same config object* as before
   c) Checks to see if the **response** was **ok** and *alerts* if it was, alerting the *contrary* if not.
   d) *Returns* **response**
7. Save this file and then check that the form still submits correctly.

This is the end of Quick Lab 24d