**Programming with JavaScript Notes**

**Week 3**

**Advanced JavaScript Features**

For Loops and Objects

It’s important to note that a for loop cannot work on an object directly, since an object is not iterable. For example:

const car = {

speed: 100,

color: "blue"

}

for(prop of car) {

console.log(prop)

}

This code will result in an error being thrown.

Uncaught TypeError: car is not iterable

Contrary to objects, arrays are iterable.

const colors = ['red','orange','yellow']

for (var color of colors) {

console.log(color);

}

This time, the output will be:  
  
red  
orange  
yellow

Fortunately, for loops can be ran on arrays to loop over objects.

In order to do this, you must have an understanding of the methods: Object.keys(), Object.values(), and Object.entries().

Object.keys()

The Object.keys() method receives an object as its parameter. Remember, this object is the object you want to loop over. Focus on the returned array of properties when you call the Object.keys() method.

Here’s an example of running the Object.keys() method on a brand new car2 object:

const car2 = {

speed: 200,

color: "red"

}

console.log(Object.keys(car2)); // ['speed','color']

When the Object.keys() method is ran and passed to the car2 object, the returned value is an array of strings, where each string is a property key of the properties contained in the car2 object.

Object.values()

Another useful method is Object.values(), it returns the values of the properties

const car3 = {

speed: 300,

color: "yellow"

}

console.log(Object.values(car3)); // [300, 'yellow']

Object.entries()  
Object.entries() is another useful method, it returns an array listing both the keys and the values.

const car4 = {

speed: 400,

color: 'magenta'

}

console.log(Object.entries(car4));

Output:

[ ['speed', 400], ['color', 'magenta'] ]

This time, the values that are returned are 2-member arrays nested inside an array. In other words, you get an array of arrays, where each array item has two members. The first being a property’s key and the second a property’s value.

[

[propertyKey, propertyVal],

[propertyKey, propertyVal],

...etc

]

To summarize, you can loop over arrays using the for of loop. You can extract object keys, values, or both, using the Object.keys(), Object.values(), and Object.entries() syntax.

With all those ingredients you can loop over any object’s own property keys and values.

Here’s an example:

var clothingItem = {

price: 50,

color: 'beige',

material: 'cotton',

season: 'autumn'

}

for( key of Object.keys(clothingItem) ) {

console.log(keys, ":", clothingItem[key])

}

The trickiest part to understanding this syntax is the clothingItem[key].

Recall learning how to access an object’s member using the brackets notation and hat you can dynamically access a property name.

To revisit this concept and show a practical example of how it works, I’ll code a function declaration that randomly assigns either the string speed or the string color to a variable name, and then build an object that has only two keys: a speed key and a color key.

After this setup, you should be able to dynamically access either one of those properties on a brand new drone object, using the brackets notation.

function testBracketsDynamicAccess() {

var dynamicKey;

if(Math.random() > 0.5) {

dynamicKey = "speed";

}else{

dynamicKey = "color";

}

var drone = {

speed: 15,

color: "orange"

}

console.log(drone[dynamicKey]);

}

testBracketsDynamicAccess();

This example may seem convoluted, but its purpose is to demo that I get either one or the other value from an object’s key, based on the string that got assigned to the dynamicKey variable, and accessed without issues, using the brackets notation.

Template Literals Examples

Template literals are an alternative way of working with strings, which was introduced in the ES6 addition to the JS language.

Up until ES6, the only way to build strings in JS was to delimit them in either single or double quotes:

"Hello, World!"

ES6 also introduced the use of backtick characters as delimeters:

`Hello, World!`

The above snippet is an example of a template string, also known as a temperate literal.

Differences between a template and a regular string:

There are several ways they are different,

First, it allows for variable interpolation:

let greet = "Hello";

let place = "World";

console.log(`${greet} ${place} !`) //display both variables using template literals

Output:

Hello World !

Essentially, using temperate literals allows programmers to embed variables directly in between the backticks, without the need to use the + operator and the single or double quotes to delimit string literals from variables. In other words, in ES5, the above code would have to be written as follows:

var greet = "Hello";

var place = "World";

console.log(greet + " " + place + "!"); //display both variables without using template literals

Besides variable interpolation, template strings can span multiple lines:

`Hello,

World

!

`

Notice that this can’t be done with string literals (that is, strings delimited in single or double quotes):

"Hello,

World"

The above code will result in a syntax error.

Put simply, temperate literals allow for multi-line strings – something that simply isn’t possible with string literals.

Additionally, the reason it’s possible to interpolate variables in template literals is because this syntax actually allows for expression evaluation.

In other words, this:

//it's possible to perform arithmetic operation inside a template literal expression

console.log(`${1 + 1 + 1 + 1 + 1} stars!`)

This will console log the string: “5 stars!”.

This opens up a host of possibilities. For example, it’s possible to evaluate a ternary expression inside a template literal.

Some additional use cases of template literals are nested template literals and tagged templates.

Data Structure Examples

Working with arrays in JavaScript

Let’s explore three specific built-in methods

1. forEach
2. filter
3. map

forEach()

Arrays in JavaScript come with a handy method that allows you to loop over each of their members:

const fruits = ['kiwi','mango','apple','pear'];

function appendIndex(fruit, index) {

console.log(`${index}. ${fruit}`)

}

fruits.forEach(appendIndex);

Output:

0. kiwi

1. mango

2. apple

3. pear

The forEach() method accepts a function that will work on each array item. That function’s first parameter is the current arrau item itself, and the second (optional) parameter is the index.

Very often, the function that the forEach() method needs to use is passed in directly into the method call, like this:

const veggies = ['onion', 'garlic', 'potato'];

veggies.forEach( function(veggie, index) {

console.log(`${index}. ${fruit}`);

});

This makes for more compact code, but perhaps somewhat harder to read. To increase readability, sometimes arrow functions are used. You can find out more about them in the additional reading.

filter()

This method filters your arrays based on a specific test. The array items that pass the test are returned:

const nums = [0,10,20,30,40,50];

nums.filter( function(num) {

return num > 20;

})

Returned array values:

[30,40,50]

Similar to the forEach() method, the filter() method also accepts a function and that function performs some work on each of the items in the array.

Map

This method is used to map each array item over to another array’s item, based on whatever work is performed inside the function that is passed-in to the map as a parameter:

[0,10,20,30,40,50].map( function(num) {

return num / 10

})

Return value:

[0,1,2,3,4,5]

This proves, choosing a proper data structure affects the very code that you can write. This is because the data structure itself comes with some built-in functionality that makes it easier to perform certain tasks or makes it harder or even impossible without converting the code to a proper data structure.

Working with Object in JS

The example below demonstrates how to use the object data structure to complete a specific task. This task is to convert an object to an array:

const result = [];

const drone = {

speed: 100,

color: 'yellow'

}

const droneKeys = Object.keys(drone);

droneKeys.forEach( function(key) {

result.push(key, drone[key])

})

console.log(result)

Output:

['speed',100,'color','yellow']

Although it’s possible, having to do something like this might mean you haven’t chosen the correct data structure to work on with your code.

Working with Maps in JS

To make a new map, use the map constructor:

new Map();

A map can feel similar to an object in JS.

However, it doesn’t have inheritance or prototypes, making it useful for data storage:

let bestBoxers = new Map();

bestBoxers.set(1, "The Champion");

bestBoxers.set(2, "The Runner-up");

bestBoxers.set(3, "The third place");

console.log(bestBoxers);

Console output:

Map(3) {1 => 'The Champion', 2 => 'The Runner-up', 3 => 'The third place'}

To get a specific value, you need the get() method:

bestBoxers.get(1); // 'The Champion'

Working with Sets in JS

A set is a collection of unique values.

To build a new set, use the Set constructor:

new Set();

The Set constructor can accept any array.

This means it can be used to quickly filter an array for unique members:

const repetitiveFruits = ['apple','pear','apple','pear','plum', 'apple'];

const uniqueFruits = new Set(repetitiveFruits);

console.log(uniqueFruits);

Output:

{'apple', 'pear', 'plum'}

Other Data Structures in JS

Besides the already built-in data structures in JS, it’s possible to build non-native, custom data structures.

These data structures come built-in natively in some other programming languages or even those other programming languages don’t support them natively.

Some more advanced data structures that I didn’t cover include:

* Queues
* Linked lists (singly-linked and doubly-linked)
* Trees
* Graphs

For resources regarding these, refer to the additional reading.

Using Spread and Rest

Here, I review:

* Adding new members to arrays without using the push() method.
* Convert a string to an array
* Copy either an object or an array into a separate object

Join arrays, objects using the rest operator.

Using the spread operator, it’s easy to concatenate arrays:

const fruits = ['apple', 'pear', 'plum']

const berries = ['blueberry', 'strawberry']

const fruitsAndBerries = [...fruits, ...berries] // concatenate

console.log(fruitsAndBerries); // outputs a single array

Result:

['apple', 'pear', 'plum', 'blueberry', 'strawberry']

It’s also easy to join objects:

const flying = { wings: 2 }

const car = { wheels: 4 }

const flyingCar = {...flying, ...car}

console.log(flyingCar) // {wings: 2, wheels: 4}

Add new members to arrays without using the push() method

Here’s how to use the spread operator to add one or more members to an existing array:

let veggies = ['onion', 'parsley'];

veggies = [...veggies, 'carrot', 'beetroot'];

console.log(veggies);

Output:

['onion', 'parsley', 'carrot', 'beetroot']

Convert a string to an array using the spread operator

Given a string, it’s easy to spread it out into separate array items:

const greeting = "Hello";

const arrayOfChars = [...greeting];

console.log(arrayOfChars); //  ['H', 'e', 'l', 'l', 'o']

Copy either an object or an array into a separate one

Here’s how to copy an object into a completely separate object, using the spread operator:

const car1 = {

speed: 200,

color: 'yellow'

}

const car 2 = {...car1}

car1.speed = 201

console.log(car1.speed, car2.speed)

The output is:

201, 200

You can also copy an array into a completely separate array, using the spread operator:

const fruits1 = ['apples', 'pears']

const fruits2 = [...fruits]

fruits1.pop()

console.log(fruits1, "not", fruits2)

Output:

['apples'] 'not' ['apples','pears']

Note the spread operator only performs a shallow copy of the source array or object, for more information visit the additional resources.

There are many more things the spread operator can do, especially in React.

Additional Resources

Template literals

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals>

Arrow functions

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions>

Spread syntax

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax>

Rest parameters

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters>

JavaScript data structures

<https://data-flair.training/blogs/javascript-data-structures/>