Back-end Web Development

March 3rd, 2020

Dorset College Dublin

JavaScript

What we covered last week...

- **Objects**
  - key/value
  - Literal notation
  - constructor notation
  - Accessing an object and dot notation
  - Updating an object
  - Many objects
  - Object instances
  - Create and access constructor notation
  - Adding and removing properties
  - .this

JavaScript

What we will cover today...

- **Objects**
  - Native and Host objects
  - Serializing Objects
- print() method

- **Iterators**
  - forEach()
  - every()
  - some()
  - map()
  - filter()
  - reduce()

- Revision

Native and Host objects

- A **native object** is an object or a class of objects defined by the ECMAScript specification
  - Arrays, functions, dates and regular expressions
- A **host object** is an object defined by the host environment (such as a web browser) within which JS interpreter is embedded
  - The HTMLElement objects that represent the structure of a web page in a client-side JS are host objects

Serializing Objects

- Object serialization is the process of **converting an object's state** to a string from which it can later be restored.
- JS provides native functions **JSON.stringify()** and **JSON.parse()** to serialize and restore JavaScript objects.
- These functions use the **JSON data** interchange format.
- JSON stands for *"JavaScript Object Notation,"* and its syntax is very similar to that of JavaScript object and array literals.

Example

```
let book = {
    title: 'JavaScript',
    'sub-title': 'The good parts',
    author: {
        firstname: 'Douglas',
        surname: 'Crockford'
    }
};

//Produces a string version of the object
let s = JSON.stringify(book);

// Converts the string to an object
let sObj = JSON.parse(s);

console.log(s);
console.log(sObj);
```

print() method

```
let book = {
    title: 'JavaScript',
    'sub-title': 'The good parts',
    author: {
        firstname: 'Douglas',
        surname: 'Crockford'
    },
    print: function(){
        console.log('I am a book!');
    }
};

book.print();
```

print() accessing object properties

```
let book = {
    title: 'JavaScript',
    'sub-title': 'The good parts',
    author: {
        firstname: 'Douglas',
        surname: 'Crockford'
    },
    print: function(){
        console.log(`The ${this.title} book was written by
        ${this.author.firstname} ${this.author.surname}`);
    }
};

book.print();
```

Iterators

Iterators are **methods** that were introduced to JS to simplify looping over arrays.

In 2009 the iterator methods that were introduced to ES5:

- forEach()
- every()
- some()
- map()
- filter()
- reduce()

forEach()

```
const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];

// iterate through all array elements
donuts.forEach(function(donutElement){
    console.log(donutElement);
});
```

- Iterates over the elements of an array:
- Note how this is used:
  arrayName.forEach(callback)
- The callback takes the following parameters:
  function(element, index, originalArray)
- You don’t have to supply all the parameters
- The methods on the following slides use the same syntax

forEach() with string interpolation

```
const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];

donuts.forEach(function(donutElement, i, donutArray){
    console.log(`Donut option ${i + 1} is ${donutArray[i]}`);
});
```

Activity

Use forEach() to take the array:
[‘Thinking in JS’, ‘JS Patterns’, ‘JS: The Good Parts’, ‘ES6 and Beyond’]

```
I need to read Thinking in JS
I need to read JS Patterns
I need to read JS: The Good Parts
I need to read ES6 and Beyond
```

Solution

```
const books = ['Thinking in JS',
               'JS Patterns',
               'JS: The Good Parts',
               'ES6 and Beyond'];

books.forEach(book => console.log(`I need to read ${book}`));
```

For loops

```
const prices = [10, 5, 6, 4, 10, 170];

for(let i = 0; i < prices.length; i++){
    if(prices[i] > 100){
        doSomethinf();
        // Break the loop and continue to execute the code following
        break;
    }
}
```

One reason to still use **for loops** is for when you need to **break out** of the loop early as shown in the example above

every()

```
const words = ['speciality', 'sleepy', 'funny'];
const words2 = ['walk', 'misty', 'happy'];

words.every(function(string){
    console.log(string[string.length - 1] === 'y');
});

words2.every(function(string){
    console.log(string[string.length - 1] === 'y');
});
```

This method returns **true** if the callback returns **true** for every element.

every()

```
const words = ['speciality', 'sleepy', 'funny'];
const words2 = ['walk', 'misty', 'happy'];

const endsInY = function(string){
    console.log(string[string.length - 1] === 'y');
};

words.every(endsInY);
words2.every(endsInY);
```

Refactored so that the callback can be written once and reused.

every() Refactored to an arrow function

```
const words = ['speciality', 'sleepy', 'funny'];
const words2 = ['walk', 'misty', 'happy'];

const endsInY = string => console.log(string[string.length - 1] === 'y');

words.every(endsInY);
words2.every(endsInY);
```

Activity

Use every to check if all elements in an array are divisible by 5.

```
const numbers = [5, 10, 15, 30]; //true
const numbers2 = [6, 10, 15, 30]; //false
```

Solution

```
const numbers = [5, 10, 15, 30]; //true
const numbers2 = [6, 10, 15, 30]; //false

const divide = num => num % 5 === 0;

console.log(numbers.every(divide));
console.log(numbers2.every(divide))
```

some()

```
const words = ['speciality', 'sleepy', 'funny'];
const words2 = ['walk', 'misty', 'happy'];

const endsInY = string => console.log(string[string.length - 1] === 'y');

words.every(endsInY); //true
words2.every(endsInY); //false
```

This method returns **true** if the callback returns **true** for at least one element.

Activity

Use **some()** to check if any of the elements in an array have more than 5 characters.

```
const names = ['Monica', 'Mathew', 'Alexandria'];
const names2 = ['Tom', 'Will', 'Alex'];
```

Solution

```
const names = ['Monica', 'Mathew', 'Alexandria'];
const names2 = ['Tom', 'Will', 'Alex'];

const long = string => string.length > 5;

console.log(names.some(long));
console.log(names2.some(long));
```

map()

```
const x = [1,2,3,4,5];

const y = x.map(function(value){
    return value * value;
});

console.log(y);
```

Applies a callback to **each element** of the array and **stores the results** in a new output array.

map() Refactored to an arrow function

```
const x = [1,2,3,4,5];

const y = x.map(value =>value * value);
console.log(y);
```

map() Another example

```
const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];

const donuts2 = donuts.map(function(donutElement){
    return donutElement + ' tasty donut';
});

console.log(donuts2);
```

map() Previous example refactored

```
const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];

const donuts2 = donuts.map(donutElement => donutElement + ' tasty donut');

console.log(donuts2);
```

Activity

- Use **map()** to take the array

[‘chocolate’, ‘red velvet’, ‘custard’, ‘jam’, ‘lemon’]

- And produce an array containing

[‘Chocolate’, ‘Red velvet’, ‘Custard’, ‘Jam’, ‘Lemon’

```
[ 'Chocolate', 'Red velvet', 'Custard', 'Jam', 'Lemon' ]
```

Solution

```
const donuts = ['chocolate', 'red velvet', 'custard', 'jam', 'lemon'];

const formatDonuts = donuts.map(function(donutElement){
    return donutElement[0].toUpperCase() + donutElement.slice(1);
});

console.log(formatDonuts);
```

filter()

```
const dictionary = ['outrageous', 'crazy', 'absurd', 'flabbergasted'];

const isLongWord = string => string.length > 6;

const longWords = dictionary.filter(isLongWord);

console.log(longWords);
```

The output array contains only those input elements for which callback returns **true**.

Activity

- Use **filter()** to parse an array and create a version containing only the positive values in the array

```
const posNum = [1, -5, -3, 2, 5, 8, -12];
// expected output [1, 2, 5, 8]
```

Solution

```
const posNum = [1, -5, -3, 2, 5, 8, -12];

const isPositive = num => num > -1;

const onlyPos = posNum.filter(isPositive);

console.log(onlyPos);
```

reduce()

```
const scoredGoals = [1,2,4,0,5,2];

const totalGoals = scoredGoals.reduce(function(sum, value){
    return sum += value;
});

console.log(totalGoals);
```

Applies a callback against an accumulator and each element in the array (from left to right) to reduce it to a single value.

reduce() Refactored to an arrow function

```
const scoredGoals = [1,2,4,0,5,2];

const totalGoals = scoredGoals.reduce((sum, value) => sum += value);

console.log(totalGoals);
```

reduce()

- The syntax differs to the previous methods: **arrayName.reduce(callback [, intialValue)**

- The initial value for the accumulator is optional. If not supplied it's set equal to the first element in the array

- The callback takes the following parameters: **function(accumulator, element, index, originalArray)**

Activity

- Use **reduce()** to iterate over this array and **sum** the length of all the elements that are longer than 6 characters

['outrageous', 'crazy', 'absurd', 'flabbergasted']

- Output should be 23

Solution

```
const dictionary = ['outrageous', 'crazy', 'absurd', 'flabbergasted'];

const longLength = dictionary.reduce((sum, value) => {
    if(value.length > 6){
        sum += value.length;
    }
    return sum;
}, 0);

console.log(longLength);
```

MapReduce

- The map() and reduce() operations are the basis of a very popular technique for processing big data

- They are often used in distributed computing environments eg. Hadoop

Further reading:

https://www.edureka.co/blog/mapreduce-tutorial/
https://www.simplilearn.com/what-is-mapreduce-and-why-it-is-important-article
https://www.ibm.com/analytics/hadoop/mapreduce

Lets Code!

Your lab for today is to use JS objects and functions integrated with HTML and CSS.

Take the example code provided in class and expand from it. *Eg. add any extra information such as whether there is gym in the hotel, is breakfast included etc.*