sofia

LAST UPDATED
January 2023

WRITTEN BY
Sofia Franek

JavaScript Fundamentals

Written by Sofia Franek

WEBSITE
www.sofiafranek.com

LINKEDIN
sofia-franek

MEDIUM
@sofiafranek_

Index

		Pages
1.0	JavaScript Overview	1
1.1	Primitive Data Types	2-3
1.2	Non-Primitive Data Types	4-6
1.3	Variables	7-9
1.4	Operators	10-13
1.5	Conditionals	14-16
1.6	Arrays	17-19
1.7	Objects	20-23
1.8	Functions	24-27
1.9	Loops	28-30
2.0	Common Methods	31-37
2.1	DOM Manipulation	38-41
2.2	Fundamentals of JavaScript Summary	42

1.0 JavaScript Overview

What is JavaScript?

Javascript is a dynamic computer programming language, that allows you to implement complex features on web pages. JavaScript is executed after the HTML and CSS source code is parsed and constructed for the web page. When JavaScript is executed it will trigger events or variables within its files and the Document Object Model (DOM) will be updated and the JavaScript will be rendered in the browser.

JavaScript is both an imperative and a declarative type of language. It contains a standard library of objects and a core set of language elements.

JavaScript can be used for client-side and server-side development. Client-side JavaScript extends the core language by giving objects to control a browser and its DOM (Document Object Model). Server-side JavaScript extends the core language by giving objects relevant to running JavaScript on a server.

What is ECMAScript?

EMCAScript (ES) is a scripting language that standardises the use of JavaScript code. Since June 2022 we are now on JavaScript ES13. Different editions of ECMAScript are released when the language is updated with new features, bug fixes and tweaks.

1.1 Primitive data types

In JavaScript, we declare variables (a way to store data) with a value. that has a data type. This tells you what kind of data is being stored in the variable and there are two types, primitive and non-primitive data types. A data type defines how we will use and represent data in our code.

JavaScript is known as a loosely typed and dynamic language. This means you do not have to specify the data type in advance because JavaScript will figure this out for you.

Primitive data types

In JavaScript, we refer to the most basic data types as primitives. Primitive data types are immutable, they cannot be changed. There are seven primitive data types: string, number, bigint, boolean, symbol, null and undefined. These data types are provided for us by JavaScript.

String

The string data type in JavaScript represents a collection of characters wrapped inside single or double quotes.

```
// Single Quotes
'Hello world!'

// Double Quotes
"Hello world again!"
```

Number

The number data type in JavaScript can be used to hold whole numbers or decimal values also known as floating-point values. We can use positive and negative numbers.

```
// Whole Number
10

// Positive Number
1

// Negative Number
-1

// Decimal Number
0.1
```

BigInt

The bigint data type is useful for large numerical values, usually over 15 digits long. To make a bigint you place an n at the end of a literal number or use the BigInt constructor and pass a number into the parameter.

```
// Placing an n at the end of the integer
12345678910n

// Using BigInt constructor
BigInt(12345678910)

// These will both give the same result --> 12345678910n
```

Boolean

The boolean data type can accept two values either true or false.

```
true
false
```

Symbol

The symbol data type was introduced in the ES6 version of JavaScript. They can be created using the function Symbol() and are unique identifiers that cannot be mutated.

```
let symbolOne = Symbol()
let symbolTwo = Symbol()

// Returns --> false
console.log(symbolOne === symbolTwo)
```

Null

The null data type can only have one value, which is null. This is most often intentionally set and it simply represents nothing or a nonvalue. You can create a variable and set the value to null.

```
let emptyVarible = null
```

Undefined

The undefined data type simply means there isn't a value assigned. This is different to null. Variables that are defined with no value or have not been declared will be given the value of undefined.

```
let noValueAssigned

// Returns --> undefined
console.log(noValueAssigned)
```

1.2 Non-primitive data types

Non-primitive data types are also known as mutable data types. This means we can change the value after creation. There are non-primitive three types: object, array and regular expressions.

Object

An object in JavaScript allows you to store multiple items in a JavaScript variable and only allows you to store one element. An object is something defined using attributes and methods and in JavaScript, everything is an object!

```
// Creating an empty object using a constructor
let objOne = new Object()

// Creating an empty object using literal notation
let objTwo = {}
```

Array

An array in JavaScript allows you to store more than one element under a single variable.

```
// Creating an empty array using a constructor
let arrOne = new Array()

// Creating an empty array using literal notation
let arrTwo = []
```

Regular Expressions (RegExp)

The RegExp object in JavaScript compares a string to a particular pattern. We can find if a string matches a regular expression pattern by using a comparison function of the RegExp object.

```
// Creating a text variable with a string
let text = 'testing'

// Declaring a regex pattern
let regExp = /testing/i

// Using method match() to see if the text variable matches the regExp pattern
let result = text.match(regExp)

// Returns --> testing
console.log(result)
```

1.3 Variables

In JavaScript, a variable is a way to store data.

Naming variables

When naming variables in Javascript you should be concise and descriptive with your names. The better the name the clearer the code is to follow. E.g. using the variable name usersEmail instead of e.

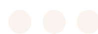
It would help if you always stayed away from abbreviations like x, y or z and short words. Unless your intentions are clear and you know what you're doing.

Variable names are case sensitive, for example — example and EXAMPLE are two different variables. Common practice is to use lowercase wording for variable names however, if you have multiple words we use camelCase, e.g. thisIsAnExample.

JavaScript has a list of reserved words you cannot use to name variables. This is because the terms already have a use in the language. E.g. let, return and function are reserved.

Limitations on variable naming

- Name must contain only letters, digits or the symbols $ or _
- The first character of the variable name must not be a digit



Creating variables

To create a variable in JavaScript we can use any of the three keywords: var, let or const.

Var

The keyword var when used to declare a variable allows the value to be accessible anywhere in the code if it is not declared inside a function. If it's declared inside a function it's only accessible in that function.

Let

The keyword let when used to declare a variable allows you to access the value stored only in the block you've declared it in.

Constants

The const keyword, which stands for constant, works the same as the var keyword but you cannot reassign the value of the variable. When you are sure that a variable will never change, you can declare it with const .

Coercion

When we convert a value from one type to another this is called coercion. Coercion is possible in JavaScript because it is dynamically typed, which means that the types of values can change at runtime.

```
// Value is a string
let number = 'One'

// Value changed to a number
number = 1
```

Variable scope

A variable can be declared either in the global scope or local scope.

Global scope

Variables that are declared out of any function are called global variables. This means they can be accessed anywhere in your code, even inside any function.

```
let myVariable = "I am a global variable!"

let myFunction = () => {
  // Result --> I am a global variable!
  console.log(myVariable)
}
```

Local scope

Variables declared inside a function are called local variables. This means they can only be accessed in the function where they were declared. If you try and use them outside of this function you will get an error.

```
let myFunction = () => {
  let myVariable = "I am a local variable!"

  // Result --> I am a local variable!
  console.log(myVariable)
}

// Result --> error: can't access local variable
// Variable is outside of the functions scope
console.log(myVariable)
```

1.4 Operators

Operators in JavaScript are special symbols used to perform operations on values and variables.

Assignment operators

Assignment operators are used to assigning values to variables. See the list below of commonly used assignment operators in JavaScript.

```
// Using the '=' operator is used to assign the value 10 to valueOne
const valueOne = 10
```

Operator	Name	Example
=	Assignment operator	a = 7; // 7
+=	Addition assignment	a += 7; // a = a + 7
-=	Subtraction assignment	a -= 7; // a = a - 7
*=	Multiplication assignment	a *= 7; // a = a * 7
/=	Division assignment	a /= 7; // a = a /= 7
%=	Remainder assignment	a %= 7; // a = a % 7
**=	Exponentiation assignment	a **= 7; // a = a**2

Comparison operators

In JavaScript, you can use comparison operators to compare two values and return a boolean based on whether the comparison is true or false.

```
// Using operator '>' to compare whether valueOne is greater than valueTwo
const valueOne = 10
const valueTwo = 11

// Result --> false
console.log(valueOne > valueTwo)
```

Operator	Name	Example
==	Equal to: will returns true if values are equal	a == b
!=	Not equal to: returns true if values are NOT equal	a != b
===	Strict equal to: returns true if values are equal or the same type	a === b
!==	Strict not equal to: returns true if values are equal but are different type or not equal at all	a !== b
>	Greater than: returns true if left value is greater than the right	a > b
>=	Greater than or equal to: returns true if left value is greater or equal to the right	a >= b
<	Less than: returns true if left value is less than the right	a < b
<=	Less than or equal to: returns true is left value is less than or equal to the right	a <= b

Arithmetic operators

Arithmetic operators in JavaScript are used to perform mathematical calculations.

```
// Result --> 20
const valueOne = 10 + 10

// Result --> 40
const valueTwo = 10 * 4

// Result --> 60
const valueThree = 120 / 2

// Result --> 80
const valueFour = 90 - 10
```

Operator	Name	Example
+	Addition	a + b
-	Subtration	a - b
*	Multiplication	a * b
/	Division	a / b
%	Remainder	a % b
++	Increment (increments by 1)	++a or a++
--	Decrement (decrements by 1)	--a or a--
**	Exponentiation (power)	a ** b

Logical operators

Logical operators in JavaScript perform logical operators that return a boolean value.

```
// Using '&&' returns true if both values are true else returns false
const valueOne = 5
const valueTwo = 10

// Result --> true
(valueOne < 6) && (valueTwo < 11);
```

<table>
<tr><th>Operator</th><th>Name</th><th>Example</th></tr>
<tr><td>&&</td><td>Logical AND: if both values are true, returns true else returns false</td><td>a && b</td></tr>
<tr><td>||</td><td>Logical OR: if either values are true, returns true else returns false</td><td>a || b</td></tr>
<tr><td>!</td><td>Logical NOT: returns true if value is false and vice-versa</td><td>!x</td></tr>
</table>

String operators

In JavaScript, you can use the +operator to join two or more strings together.

```
// Result --> hello world
console.log('hello' + 'world')
```

Other operators

There are other operators available in JavaScript, but the ones I've mentioned are the most commonly used in everyday programming.

1.5 Conditionals

In JavaScript, you'll find sometimes you need to make decisions. We call these conditionals statements, they control behaviour in JavaScript and determine whether a particular group of code runs.

If-else statements

An if statement is used when a condition is true. An else statement is used after an if or else if statement when the condition/s have both been false. You can use a number of else-if statements after the first condition is false in a statement.

```
if (conditionOne) {
  // If (conditionOne === true
} else if (conditionTwo) {
  // If (conditionOne === false) and (conditionTwo === true)
} else {
  // If (conditionOne === false) and (conditionTwo === false)
}
```

Conditional operators

In JavaScript, we sometimes just want to create a small condition. You can use conditional operators in this case. The ? represents an if and the : can be read as the else.

```
// Syntax of a conditional operator
condition ? val1 : val2

// Example of conditional operator in use
let number = 19;
let example = (number >= 18) ? 'over 18' : 'under 18';
```

Switch statements

A switch statement in JavaScript can be used to replace an if-else statement if you have multiple checks. A switch statement has one or more case blocks and an optional default.

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    break;

  case 'value2':  // if (x === 'value2')
    break;

  default:
    break;
}
```

1.6 Arrays

Arrays in JavaScript are a special type of object used to store a sequence of data.

Creating an array

In JavaScript, there are two ways to create an array.

Using an array literal

```
const arrayOne = ["hello", "world"]
```

Using the new keyword

```
const arrayTwo = new Array("hello", "world")
```

The recommended method is to use an array literal to create an array. You can store other data types inside an array — see my chapter about Primitive and Non-Primitive Data Types.

Accessing elements in an array

To access elements in an array you can use the indices.

```
const arrayThree = [1, 2, 3, 4, 5]

// Result --> 1
console.log(arrayThree[0])

// Result --> 5
console.log(arrayThree[4])
```

In array's the index starts at 0 and not 1.

Adding an element to an array

In JavaScript, there are built-in methods that allow you to add elements to an array.

push()

The push() method allows you to add an element at the end of the array.

```
const arrayFour = ["hello", "to", "the"]

// Add an element to the end of array
arrayFour.push("world")

// Result --> ["hello", "to", "the", "world"]
console.log(arrayFour)
```

unshift()

The unshift() method allows you to add an element at the beginning of the array.

```
const arrayFive = ["to", "the", "world"]

// Add an element to the start of array
arrayFive.unshift("hello")

// Result --> ["hello", "to", "the", "world"]
console.log(arrayFive)
```

Changing elements in an array

You are able to change or add elements in an array by accessing the index values.

```
const arraySix = ["hello", "world"]

// This will add a new element at index 1
arraySix[1] = "the"

// Result --> ["hello", "the", "world"]
console.log(arraySix)
```

Removing an element in an array

In JavaScript, there are built-in methods that allow you to remove elements in an array.

pop()

The pop() method allows you to remove the last element of an array and also returns the returned value.

```
const arraySeven = ["hello", "to", "the", "world"]

// Remove the last element of array
const removedElement = arraySeven.pop()

// Result --> ["hello", "to", "the"]
console.log(arraySeven)

// Returned value from pop() method
// Result --> "world"
console.log(removedElement)
```

shift()

The shift() methods allow you to remove the first element of an array and also return the removed element.

```
const arrayEight = ["hello", "to", "the", "world"]

// Remove the first element of array
const removedElement = arraySeven.shift()

// Result --> ["to", "the", "world"]
console.log(arrayEight)

// Returned value from shift() method
// Result --> "hello"
console.log(removedElement)
```

Array methods

These are the most commonly used array methods in JavaScript, including the ones mentioned above.

Method	Descrption
concat()	Joins two or more arrays and returns result
indexOf()	Searches as element of an array and returns its position
forEach()	Calls a function for each element
includes()	Checks if an array contains a specific value
sort()	Sorts the elements alphabetically in strings and in ascending order
slice()	Selects the part of an array and returns the new array
splice()	Removes or replaces existing elements and/or adds new elements

1.7 Objects

In JavaScript, an object is an unordered collection of related data in key-value pairs. You can create an object by using the object literal notion.

```
// Object created using object literal notion

let exampleObject = {
  name: ["Sofia", "Franek"],
  gender: "female"
}
```

Accessing properties

To access the properties of an object, you can use either the dot or bracket notation.

Dot notation (.)

Using the exampleObject above I can access the name property using dot notation.

```
// objectName.keyName = value

exampleObject.name = ["Sofia", "Franek"]
```

Bracket notation ([])

```
// objectName["keyName"] = value

exampleObject['name'] = ["Sofia", "Franek"]
```

Adding a new property to an object

In JavaScript, you can add a property to an object even after you’ve created the object.

```
let exampleObject = {
  name: ["Sofia", "Franek"],
  gender: "female"
}

exampleObject.age = 24

/*
Result -->
  let exampleObject = {
    name: ["Sofia", "Franek"],
    gender: "female",
    age: 24
  }
*/
console.log(exampleObject)
```

Modifying the value of a property

You can use the assignment operator (=) to change a property's value in an object.

```
let exampleObject = {
  name: ["Sofia", "Franek"],
  gender: "female"
}

name.exampleObject = ["Sofia"]

/*
Result -->
  let exampleObject = {
    name: ["Sofia"],
    gender: "female",
    age: 24
  }
*/
console.log(exampleObject)
```

Checking if a property exists

If you want to check if a property exists in your object, you can use the in operator. If the propertyName exists in the objectName, the in operator will return true, else it will return false.

```
// propertyName in objectName

let exampleObject = {
  name: ["Sofia"],
  gender: "female",
  age: 24
}

// Result --> false
console.log("Sofia" in exampleObject)

// Result --> true
console.log(age in exampleObject)
```

Deleting a property

To delete a property of an object, you can use the delete operator.

```
// delete objectName.propertyName

delete exampleObject.gender

/*
Result -->
  let exampleObject = {
    name: ["Sofia"],
    age: 24
  }
*/
console.log(exampleObject)
```

Object methods

There are many methods in JavaScript to manipulate, add, and delete data from objects. These are the most common ones used every day in JavaScript.

```
let exampleObject = {
  name: ["Sofia"],
  age: 24
}

// Result --> ["name", "Sofia"]
Object.entries(exampleObject)[0]

// Result --> ["Sofia", "24"]
Object.values(exampleObject)

// Result --> ["name", "age"]
Object.keys(exampleObject)

/*
Result -->
  let exampleObject = {
    name: ["Sofia"],
    age: 24,
    gender: "female"
  }
*/
Object.assign(exampleObject, {gender: "female"})
```

1.8 Functions

In JavaScript, if you want to do a task you use a function.

Declaring a function

To declare a function you use the function keyword. When you name a function you want to be as descriptive as possible with the name, similar to when naming a variable in JavaScript.

```
function myFunction() {
  console.log("Hello world")
}
```

Function expressions

Functions can also be defined as expressions, you do this by assigning the function to a variable.

```
let myVariable = myFunction() {console.log("Hello world")}
```

Function parameters

You can also declare parameters to be used by a function. A parameter is a value that is passed into a function when it's declared.

```
function myFunction(name) {
  console.log("Hello my name is", name)
}

let result = myFunction("Sofia")

// Result --> "Hello my name is Sofia"
console.info(result)
```

Arrow functions

When ES2015 came out, we can now declare functions in a shorter syntax which is called an arrow function. Arrow functions can be written inline, it saves space in your code compared to regular functions.

```
// Regular function
const myFunction = () => {
  console.log("Hello world")
}

// Inline function
const myFunction = () => {console.log("Hello world")}
```

Calling a function

After you've declared a function you need to call it if you want to use it.

```
// Function decalred
function myFunction() {
  console.log("Hello world")
}

// Function called
myFunction();
```

Return statement

The return statement in a function can be used to return the value to a function call. If nothing is returned from the function, the function will return undefined.

```
// Function decalred
function myFunction(a, b) {
  return a + b
}

// Function called
let result = myFunction(1, 2);

// Result --> 3
console.log(result)
```

Callback functions

A callback function is a function that receives another function as an argument. The received function is called the callback function.

```
// Arrow function
const toAdd = (a, b) => {
  return a + b
}

// Callback function
const callbackFunction = (num1, num2, fn) => fn(num1, num1)

// Result --> 3
callbackFunction(1, 2, toAdd)
```

Promises

With callback functions, you can run into a problem if you are not careful. Can you imagine a callback function containing another callback function, and then that callback function had another callback function? This would be chaotic! To avoid this pattern we use promises in JavaScript.

Promises allow us to postpone the execution of a function until an asynchronous operation is completed. Promises have three states; pending, fulfilled and rejected.

Pending: the initial state of a promise
Fulfilled: the specified operation was completed
Rejected: the operation did not complete

```
let promiseFunction = new Promise((resolve, reject) => {
  let thisIsWorking = true

  if(thisIsWorking) {
    resolve("Yes!")
  } else {
    reject("No")
  }
}

promiseFunction.then((fromResolve) => {
  console.log("Did it work?", fromResolve)
}).catch((fromReject) => {
  console.log("Did it work?", fromReject)
})
```

When the promise is resolved the .then method will be fired but when the promise is rejected, the .catch method will be fired.

Resolve: this means the function did what it said it was going to do
Reject: this means the function didn't do what it was going to do

Async/await

Using async/await in JavaScript, means you are waiting for one promise to resolve before moving to the next line of code.

In the function, you use the keyword async. Any async function returns a promise, and the resolved value of the promise will be what's returned from the function. The keyword await means the function will wait until whatever the await is waiting for returns a value, and then it will resume to the next line of code.

```
let myFunction = async () => {
  await getData()
  return "All done!"
}

// This function will wait for getData() to be called
// Until function receives getData() it won't return "All done!"
myFunction()
```

1.9 Loops

In JavaScript, loops are used to iterate over a piece of code. There are four types of loops in JavaScript; for loop, while loop, do/while loop, and for/in loop.

For loop

The for loop iterates over elements for a fixed number of times. To use this loop you need to know the number of iterations you want.

```
// for(initialization; condition; increment) {}

for(i=1; i <= 5; i++) {
  console.log(i);
}

/* Result -->
  1
  2
  3
  4
  5
*/
```

While loop

The while loop iterates over elements for an infinite number of times. To use this loop you don't need to know the number of iterations.

```
// while(condition) {}

let i = 5;

while(i <= 10) {
  console.log(i);
  i++;
}

/* Result -->
  5
  6
  7
  8
  9
  10
*/
```

Do/while loop

The do/while loop iterates over elements for an infinite number of times, like the while loop. However, with a do-while loop, the code is executed at least once whether the condition results in true or false.

```
// do(condition) {} while(condition)

let i = 10;

do(i++) {
  console.log(i);
} while(i <= 15)

/* Result -->
  10
  11
  12
  13
  14
  15
*/
```

For/in loop

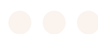
The for/in loop iterates over an object and returns every value.

```
// for(const item in myObject) {}

let myObject = {
  a: 1,
  b: 2,
  c: 3
}

for(const item in myObject) {
  console.log(myObject[item])
}

/* Result -->
  1
  2
  3
*/
```



Break and continue

In loops, the break statement can be used to jump out of the loop. If a certain condition returns true, then you can stop the loop.

```
for(i=1; i <= 5; i++) {
  if(i === 3) { break;
    console.log(i);
  }
}

/*
  1
  2
*/
```

In the continue statement allows you to skip an iteration if a certain condition returns true.

```
for(i=1; i <= 5; i++) {
  if(i === 3) { continue;
    console.log(i);
  }
}

/*
  1
  2
  4
  5
*/
```

2.0 Common methods

In JavaScript, there are a number of built-in methods that we can use and they are accessible anywhere in our JavaScript files.

Methods for strings

.concat()

This method joins two or more strings and returns a new joined string.

```
let firstString = "Hello";
let secondString = "World";

// Result --> "Hello World"
secondString.concat(firstString);
```

.includes()

This method checks whether a string contains any specific string/characters you have stated in the parentheses of the method. Be aware it is case-sensitive!

```
let myString = "Hello to the world";

// Result --> true
myString.includes('Hello');

// Result --> false
myString.includes('hello');
```

.slice()

This method extracts a part of a string and then returns a new string. The first argument is the first index of the new array. The second argument is the last index which will be removed.

```
// Result --> "ello"
"Hello world".slide(1, 5)
```

.split()

This method splits a string into an array of substrings.

```
// Result --> ["Hello", "to", "the", "world"]
"Hello to the world".split(' ')
```

.toLowerCase()

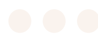
This method converts a string to lowercase letters.

```
// Result --> "hello world"
"HELLO WORLD".toLowerCase()
```

.toUpperCase()

This method converts a string to uppercase letters.

```
// Result --> "HELLLO WORLD"
"hello world".toLowerCase()
```



Methods for arrays

.concat()

This method joins two or more arrays and returns a copy of the joined arrays.

```
let firstArray = [1, 2, 3];
let secondArray = [4, 5, 6];

// Result --> [1, 2, 3, 4, 5, 6]
firstArray.concat(secondArray)
```

.filter()

This method creates a new array with every element in an array that passes a test.

```
const test = (value) => value >= 5;

let arr = [1, 2, 3, 5, 10, 15, 20];
let filteredArr = arr.filter(test)

// Result --> [5, 10, 15, 20]
console.log(filteredArr)
```

.forEach()

This method calls a function for each element in the array.

```
let array = [1, 2, 3];

// Result --> 2, 4, 6
array.forEach(element => console.log(element * 2));
```

.join()

This method joins all the elements of an array into a string.

```
// Result --> "123"
[1, 2, 3].join('');

// Result --> "1,2,3"
[1, 2, 3].join();
```

.map()

This method creates a new array with the result of a function called by each element in the array.

```
const arr = [1, 2, 3];
const mappedArr = arr.map(x => x * 2);

// Result --> [2, 3, 6]
console.log(mappedArr);
```

.pop()

This method removes the last element of an array and returns that element.

```
const nums = [1, 2, 3, 4, 5, 6];
const lastNum = nums.pop();

// Result --> [1, 2, 3, 3, 5]
console.log(nums);

// Result --> 6
console.log(lastNum);
```

.push()

This method adds new element(s) to the end of an array and returns the new length.

```
const nums = [1, 2, 3, 4, 5, 6];
const newNums = nums.push(7);

// Result --> [1, 2, 3, 3, 5, 6, 4]
console.log(nums);

// Result --> 7 (The length of the new array)
console.log(newNums);
```

.reduce()

This method reduces the value of an array to a single value.

```
const nums = [1, 2, 3];
const totalOfNums = nums.reduce((cur, acc) => cur + acc);

// Result --> 6
console.log(totalOfNums);
```

.reverse()

This method reverses the order of the elements within the array.

```
// Result --> [3, 2, 1]
[1, 2, 3].reverse()
```

.shift()

This method removes the first element on an array and returns that element.

```
const nums = [1, 2, 3];
const firstNum = nums.shift();

// Result --> [2, 3]
console.log(nums);

// Result --> 1
console.log(firstNum);
```

.slice()

This method selects a part of an array and then returns the new array. The original array doesn't get modified.

```
const nums = [1, 2, 3, 2, 1];
const newNums = nums.slice();

// Result --> [2, 3]
console.log(nums);

// Result --> 1
console.log(newNums);
```

.sort()

This method sorts the elements of an array.

```
// Result --> [1, 1, 2, 2, 3]
[1, 2, 3, 2, 1].sort();
```

.splice()

This method adds/removes an element(s) from an array.

```
const nums = [1, 2, 3, 2, 1];
nums.splice(2, 0, "new");

// Result --> [1, 2, "new", 3, 2, 1]
console.log(nums);
```

.unshift()

This method adds a new element to the beginning of an array and returns the new length.

```
const nums = [3, 4, 5];
const newNumsLength = nums.unshift(1, 2)

// Result --> [1, 2, 3, 4, 5]
console.log(nums);

// Result --> 5
console.log(newNumsLength);
```

.toString()

This method converts an array to a string and then returns the result.

```
// Result --> "Hello, World"
["Hello", "World"].toString();
```

The Math object

The Math object in JavaScript, allows us to do mathematical calculations on numbers very easily.

Math.round()

This method returns the passed value to its nearest integer (whole number) either down or up.

```
// Result --> 1
Math.round(1.2)

// Result --> 2
Math.round(1.5)

// Result --> 2
Math.round(1.6)
```

Math.min() and Math.max()

The .min() method finds the lowest value in a list of arguments, whereas the .max() method finds the highest.

```
// Result --> 1
Math.min(1, 10, 100, 5, 50, 500);

// Result --> 500
Math.max(1, 10, 100, 5, 50, 500);
```

Math.ceil()

This method returns the passed value up to its nearest integer (whole number).

```
// Result --> -1
Math.ceil(-1);

// Result --> 4
Math.ceil(4.11)
```

Math.floor()

This method returns the passed value down to its nearest integer (whole number).

```
// Result --> 1
Math.floor(1.7)

// Result --> -2
Math.floor(-1.2)
```

Math.random()

This method returns a random value.

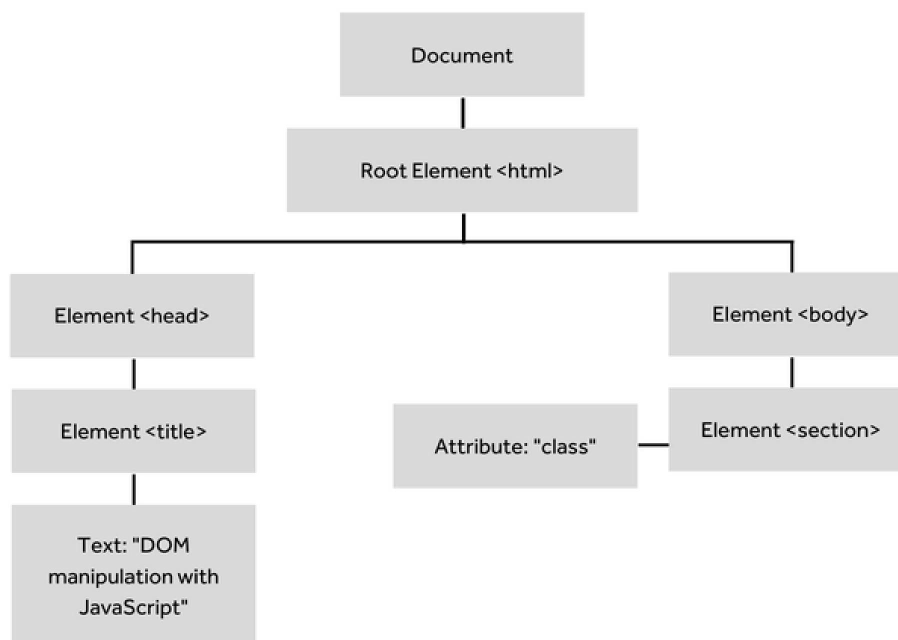
```
// Result --> Returns a random value between 0 and 1
Math.random();

// Result --> Returns a random value between 0 and 100
Math.random() * 100;
```

2.1 DOM Manipulation

The DOM which stands for Document Object Model is a tree-like structure showing the relationship between different HTML elements and their hierarchy.

Understanding the DOM



Above is a diagram which is a structure of a typical HTML page.

With JavaScript, we can easily create, modify and remove HTML elements and attributes, change styles, and respond to all HTML events on the page in the DOM.

The DOM is created when the webpage is loaded. The document is the root node which is accessible in the window object and has access to all other nodes as it's on top of the tree - (Shown in the diagram above).

Adding an element to the DOM

document.createElement()

This method allows you to create HTML elements with JavaScript. You do this by using tag names which are for example — section, div, h1, p, etc.

```
<!-- document.createElement("tagname") -->

let createDiv = document.createElement("div")
```

parentNode.appendChild()

This method appends a node as the last child of a node. This means if you want to create a div with a paragraph in it, you can do this using this method.

```
<!-- parentNode.appendChild(node) -->

let div = document.createElement("div")
let paragraph = document.createElement("p")

<!-- <div> <p> </p> </div> -->
div.appendChild(paragraph)
```

Accessing elements in the DOM

In JavaScript, you can use methods to manipulate the content of HTML elements and you can use a property to access the property of HTML elements.

document.getElementById()

This method returns the element in an object whose id matches the string inputted to the methods.

```
<!-- document.getElementById("id") -->

<p id="test"></p>

const test = document.getElementById("test")
```

document.getElementByClassName()

This method returns a node list which represents the elements whose class name matches the string inputted to the methods.

```
<!-- document.getElementsByClassName("class") -->

<p class="test"></p>
<p class="test"></p>
<p class="test"></p>

const test = document.getElementByClassName("test")
```

document.querySelector()

This method returns the first element within the document that matches a selector or group of selectors you've specified.

```
<!-- document.querySelector("#id, .class or element") -->

<h1 id="title">Test 1</h1>
<h2 class="heading">Test 2</h2>
<p>Test 3</p>

const firstElement = document.querySelector("#title")
const secondElement = document.querySelector(".heading")
const thirdElement = document.querySelector("p")
```

element.getAttribute()

This method gets the value of the specified attribute.

```
<!-- element.getAttribute(attributeName) -->

<p class="test"></p>

let testParagraph = document.querySelector("p")

<!-- returns test -->
testParagraph.getAttribute("class")
```

Removing an element from the DOM

To remove an element from the DOM you can use the remove()method. You first need to use a method that selects the element you want to remove, like in the example below.

```
<!-- element.remove() -->

<p class="test"></p>

const deleteElement = document.querySelector("p")

deleteElement.remove()
```

2.2 Fundamentals of JavaScript Summary

In this eBook, we have covered the following topics which I believe to be the fundamentals of learning JavaScript.

- JavaScript Overview
- Primitive Data Types
- Non-Primitive Data Types
- Variables
- Operators
- Conditionals
- Arrays
- Objects
- Functions
- Loops
- Common Methods
- DOM Manipulation

Recommend resources

Eloquent JavaScript 3rd Edition (Book) - https://eloquentjavascript.net/

Beginning JavaScript (TeamTreeHouse) - https://teamtreehouse.com/tracks/beginning-javascript

Starting with JavaScript (Coding Ninjas) - https://www.codingninjas.com/free-content/full-stack-web-development/content/starting-with-javascript

Medium - https://medium.com/

MDN Web Docs - https://developer.mozilla.org/en-US/docs/Web/JavaScript

Thank you for reading my eBook.

Happy coding!