This file contains all my JavaScript learnings from the Web Development Bootcamp course from Udemy

[Browser Rendering Behind the scenes](https://blog.logrocket.com/how-browser-rendering-works-behind-scenes/)

Flow: The data is first received by the browser engine through server or local storage in the form of bytes. The data is then converted into characters. The HTML content is then used to create nodes (which consists of properties of HTML elements) which are then used to create DOM (Document object model).

Which reading the HTML file, Two things may occur

1. if the browser engine encounters inline a link to CSS file, it then simultaneously builds CSSOM tree just like DOM (DOM construction is not halted).
2. If the browser engine encounters js script, DOM construction is Halted until the js script execution is complete and so THE LOCATION of the script matters in the document. And Note that js script execution waits till CSSOM is constructed.

This is because JavaScript can alter both the DOM and CSSOM. Because the browser isn’t sure what this particular JavaScript will do, it takes precautions by halting the entire DOM construction altogether.

Once the DOM and CSSOM is ready, the browser engine constructs a **render tree** which is the combination of DOM and CSSOM and then displays the content of the document in the browser.

(Read about CSSOM)

**Chapter 142:**

Primitive types in js: Number, String, Boolean, Null, Undefined

Technically there are two others also: BigInt and Symbol

**NOTE: Strings are primitive unlike in java. This means ‘Hi’ === ‘Hi’ returns true whereas in java it returns false.**

**NOTE: JS is a pass by value language.**

[**is-string-a-primitive-type-or-object-in-javascript (VIP)**](https://stackoverflow.com/questions/7675127/is-string-a-primitive-type-or-object-in-javascript) **as string also has methods like toUpperCase, or properties like length etc, So what happens when** “string”.length **is executed? So** “string” **is of primitive type and for a very short period of time,** “string” **is wrapped with the object, then the property of the object is used (in this case: length) and then the object gets disposed immediately. This is called as Coertion (Read more about this)**

Type Coercion refers to the process of automatic or implicit conversion of values from one data type to another. This includes conversion from Number to String, String to Number, Boolean to Number etc. when different types of operators are applied to the values. ([Coertion in JS](https://www.geeksforgeeks.org/what-is-type-coercion-in-javascript))

**Coertion works both ways**

> String("hello ") + String("world")

"hello world"

> Number(2) + 3

5

**The objects are down-casted to their primitive versions in order to accomplish the operation.**

if(typeof(s) == "string" || s instanceof String){

//s is a string (literal or object)

}

And don't forget "string" instanceof String is false, whereas new String() instanceof String is true. So to check whether some variable s is a string, you'd have to check whether typeof(s) == "string" || s instanceof String

**Chapter 143: Numbers**

JS has only one number type which consists of decimal, positive, negative numbers etc.

**Chapter 144: NaN (Not a Number)**

Eg: 0/0, 1 + NaN

Note that typeof NaN is number

**Chapter 146: Declaring Variables**

Variables are declared using “let” or “var” or “const”

**Difference between let, var and const**

[Difference between let, var and const](https://www.educative.io/edpresso/difference-between-var-let-and-const-keyword-in-javascript)

1. Scope: Variables declared using “var” are function scoped. “let” and “const” variables are block scoped. (They will be accessible inside the block only)
2. Hoisting: Hoisting means we can define a variable before its declaration. “var” variables follow hoisting while “let” and “const” do not follow hoisting.

x = 8;

console.log(x);

var x;

for the above code, we will not get any error as hoisting is allowed for var. But if we had “let” or “const” instead of “var”, then we would have got an error.

1. Reassigning: “var” and “let” variables can be reassigned but “const” variables cannot be reassigned.
2. Redeclaration: “var” variables can be redeclared but “let” and “const” variables cannot be redeclared.

var v1 = 1;

var v1 = 30;

console.log(v1);

The above code would not give any error. But instead of “var”, if we used “let” or “const”, then we would get error

1. **NOTE: Variables created without a declaration keyword (var, let, or const) are always global, even if they are created inside a function.**

**Chapter 153:** Strings are indexed just like arrays and strings are immutable in js.

**Chapter 156: Template literals**. Template literals are strings which allow to embed expressions which will be evaluated and then turned into a literal string.

`I counted ${3 + 4} sheep` // “I counted 7 sheep”

Note: Use Back-ticks and not single quotes or double quotes. And the expression to be evaluated needs to be inside the curly braces and there should be a $ sign before the starting of curly braces.

**Chapter 157: Null and Undefined**

**Null:** Intentional absence of any value. Must be assigned

**Undefined:** Variables that do not have an assigned value are undefined.

**NOTE: Any expression involving undefined, null or NaN results in NaN**

(Read more)

**Chapter 161: Comparison Operators**

Difference between === and ==

== stands for equality

=== stands for strict equality

== Return true only if the two operands are equal while === returns true only if both values and data types are the same for the two variables.

Eg: let x = 5

x == 5 returns true and x == “5” returns true but x === “5” returns false

some weird results using ==

**5 == “5” //true**

**null == undefined; //false**

**0 == false; //true**

**Always try to use === instead of == and !== instead of !=**

**Chapter 163: Console, alert and prompt**

**Chapter 164: Connect JS File with HTML**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <script src = "app.js"></script>

    <title>Document</title>

</head>

<body>

</body>

</html>

Connecting app.js file with html file

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

//ALL HTML CODE

    <script src = "app.js"></script>

</body>

</html>

**Chapter 169: Truthy and Falsey Values**

false, 0, null, “” (empty string), NaN, undefined are falsey values.

Rest all are truthy values

Eg:

let a = null;

if(a) console.log("TRUTHY");

else console.log("FALSEY");

let a = '';

if(a !== '') is same as if(a)

**Chapter 175: Introducing Arrays**

In JS, we can have mixed array

For eg:

let a = [null, undefined, 2, "Hi", NaN, 4.0] //it is a valid array

**NOTE:**

**There is no index-out-of-bound exception in JavaScript. If an index greater than the array length is specified when retrieving an array element, the "undefined" value will be returned. Same case with Strings**

**Chapter 177 and 178: push, pop, shift and unshift**

push and pop: adding and removing elements in the end (O(1))

unshift and shift: adding and removing elements in the start (O(n))

**Chapter 179 and 180: Concat, indexOf, includes, reverse, slice and splice (Common Array Methods)**

Concat: Merges 2 arrays and returns a new merged array, does not change existing array

indexOf: returns the idx to the given value, returns -1 if value is not found.

Includes: returns true if given value is present in the array, else returns false

Reverse: reverses the array (Overwrites the original array)

Slice: exactly like string substring method, returns a new array (Does not change the original array)

Splice: inserts new element(s) and deletes given number of elements (rewrites original array)

Splice syntax:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.splice(2, 1, "Lemon", "Kiwi"); // insert 2 elements "Lemon" and "Kiwi" starting from 2nd idx and remove one element

//[“Banana”, “Orange”, “Lemon”, “Kiwi”, “Mango”]

fruits.splice(2, 2); //At pos 2, remove 2 elements

**Note that arrays.sort(nums) does not work as intended. It sorts an array lexicographically even though the contents are numbers. Use Comparators for sorting**

**Chapter 182: Arrays + const**

const arr = [1, 2]; //This means arr cannot be reassigned but elements can be

added or removed

arr.push(3); //NO ERROR as we are not reassigning

const arr2 = [1];

arr = arr2 //ERROR

It is generally a good idea to use constant objects

Read this:

[why-does-a-string-index-in-an-array-not-increase-the-length](https://stackoverflow.com/questions/9526860/why-does-a-string-index-in-an-array-not-increase-the-length)

Javascript arrays cannot have "string indexes". A Javascript Array is exclusively numerically indexed. When you set a "string index", you're setting a property of the object with key as the string.

These are equivalent:

array.a = 'foo';

array['a'] = 'foo';

[JS Run Time of Array Functions](https://stackoverflow.com/questions/22614237/javascript-runtime-complexity-of-array-functions)

**Chapter 185: Object Literals**

1. Objects are collection of properties
2. Properties are a key value pair
3. Rather than accessing data using an index, we use custom keys

//objects syntax

const fitbitData = {

    totalSteps: 123,

    avgCalorieBurn: '32KCal',

    avgGoodSleep: null

}

**Chapter 187: Accessing data from Objects**

1. For accessing the data, we can either use dot or square brackets

//accessing the data

fitbitData['totalSteps'] //is same as fitbitData.totalSteps

**NOTE: ALL VALID KEYS ARE CONVERTED TO STRINGS EXCEPT FOR SYMBOLS**

const years = {

    1999: "GOOD",

    2020: "TOO GOOD",

    2021: "BONK!"

};

years["1999"] //is same as years[1999] because the key is converted into a string

**This means we can have null, undefined as keys too because EVERY KEY IS CONVERTED TO STRING**

const years = {

    null: "GOOD",

    undefined: "TOO GOOD",

};

years[null] //is same as years["null"] or years.null because the key is converted into a string

console.log("string1", "string2"); //prints string1 string2

**Chapter 199: For Of loops**

//for of loops

let arr = ["I", "am", "trying", "to", "learn", "JS"];

for(let str of arr)console.log(str);

//is same as for(String s: arr) in java

**Chapter 200: Iterating over objects**

**ACTUAL OBJECT LITERALS WITH CURLY BRACES ARE NOT ITERABLE LIKE ARRAYS**

const testScores = {

    keenan: 80,

    damon: 67,

    vonnie: 60

};

for(let pair of testScores)console.log(pair);

**VM3008:1 Uncaught TypeError: testScores is not iterable**

**at <anonymous>:1:17**

**These are objects, But these are NOT Iterable objects**

**To iterate over an object, we use for…in instead of for…of**

for(let key in testScores)console.log(key); //prints the keys in the object

//We can also extract the keys and values using Object.keys(testScores)

//and Object.values(testScores) which returns an array of keys and values respectively

//Object.entries(testScores) gives us a nested array of key-value pairs

**Chapter 202: DOUBT: WHERE TO PUT THE SCRIPT IN HTML FILE FOR OPTIMAL RESULTS**

**Chapter 205: Functions**

**Reference:** [**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions)

[**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Function**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function)

**NOTE: Functions are objects in JS**

function myFunction(){

}

**NOTE: If we try to call our function before declaring, it will execute without any error due to hoisting (Just like var has the hoisting property which lets the variable to be defined before declaration)**

**Chapter 210 and 211: Function and block scope**

**READ THIS FOR SCOPES:** [**https://stackoverflow.com/questions/17279437/lexical-scope-closures-in-javascript**](https://stackoverflow.com/questions/17279437/lexical-scope-closures-in-javascript)

let animal = "Giant Pacific Octopus";

function observe(){

    let animal = "Pajama Squid";

    console.log(animal); //prints Pajama Squid

}

observe();

//The above block of code will run without any error.

//This first animal has global scope and second animal has function scope. So As they are in 2 different scopes, redeclaration results in no error

let animal = "Giant Pacific Octopus";

function observe(){

    console.log(animal); //prints "Giant Pacific Octopus"

}

observe();

let x = 10;

// Here x is 10

{

    let x = 2;

    // Here x is 2

}

// Here x is 10

let x = 5;

{

    let x = 10;

    {

        let x = 15;

        console.log(x);

    }

    console.log(x);

}

console.log(x);

//The above code will run without any error. As all the x's have different scopes

const creature = "Common Sea Dragon";

function scubaDive(){

    const creature = "Spanish Dancer"; //A type of sea slug

    console.log(creature); //prints Spanish Dancer

}

scubaDive();

//This block of code also will not result any error because the creature variable has two scopes. The first creature has global scope and second creature has function scope

// So As they are in 2 different scopes, redeclaration results in no error

**Chapter 212 Lexical Scope:**

Again, Refer this: [**https://stackoverflow.com/questions/17279437/lexical-scope-closures-in-javascript**](https://stackoverflow.com/questions/17279437/lexical-scope-closures-in-javascript)

Javascript closures: <https://dmitripavlutin.com/simple-explanation-of-javascript-closures/>

Some questions: <https://dmitripavlutin.com/javascript-closures-interview-questions/>

Lexical Scope allows functions to access variables from outer scopes

Closure is just a function which has access to all the variables in its lexical scope

**Chapter 213: Function expression**

**References:**

[**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function)

[**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/callee**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/callee)

The main difference between a function expression and a function declaration is the function name, which can be omitted in function expressions to create anonymous functions.

**NOTE: function hoisting only works with function declarations—not with function expressions.**

square(); //does not give any error because of hoisting

function square(num){

    return num \* num

}

square(); //Error because of function expressions do not support hoisting.

const square = (num) => num \* num;

**Anonymous Functions: Functions without any name**

function(num){

    return num \* num;

} //will give a error because this function cannot be called

const square = function(num){

    return num \* num;

} //will not give any error because this function object is stored in square variable

**COMMON USES OF ANONYMOUS FUNCTIONS**

1. Anonymous functions can be used as an argument to other functions (callback functions)

button.addEventListener('click', function(event) {

    console.log('button is clicked!')

});

1. Anonymous Functions can be used as **Immediately Invoked Function expression (IIFE)**

(function() {

    console.log('Code runs!')

})();

// or

!function() {

  console.log('Code runs!')

}();

When do we need name function expressions?

Consider the below code:

function factorial (n) {

    return !(n > 1) ? 1 : factorial(n - 1) \* n;

}

[1, 2, 3, 4, 5].map(factorial);

But

[1, 2, 3, 4, 5].map(function(n) {

    return !(n > 1) ? 1 : /\* what goes here? \*/ (n - 1) \* n;

});

To solve this we can either use named function expression or use [arguments.callee](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/callee) method which returns the current function

Every function has a “name” property (Remember Functions are objects in JS)

The name doesn't change if it's assigned to a different variable. If function name is omitted, it will be the variable name (implicit name). If function name is present, it will be the function name (explicit name). This also applies to arrow functions (arrows don't have a name so you can only give the variable an implicit name).

var foo = function() {}

foo.name // "foo"

var foo2 = foo

foo2.name // "foo"

var bar = function baz() {}

bar.name // "baz"

console.log(foo === foo2); // true

console.log(typeof baz); // undefined

console.log(bar === baz); // false (errors because baz == undefined)

**Chapter 216: Methods**

**What is the difference between function and methods? Methods are just functions which belong to an object**

**Every method is a function and vice versa is not true**

Short hand for methods

const math = {

    PI: 3.14,

    add: (x, y) => x + y,

    multiply: (x, y) => x \* y

};

//is same as

const math2 = {

    PI: 3.14,

    add(x, y){

        return x + y;

    },

    multiply(x, y){

        return x \* y;

    }

}

//math.add(10, 50) is same as math2.add(10, 50)

Array Methods and callbacks (Chapter 219 to 230)

const arr = [1, 2, 3, 4, 5, 6];

arr.forEach((num) => console.log(num \* 2));

//forEach method does not return any new array, it is used to traverse the array

//NOTE: array methods can also be written in the below way

arr['forEach']((num) => console.log(num)) //NOT advised to write like this...use dot notation

arr.filter((num) => num % 2 === 0);

//returns a new array with all even numbers

arr.map((num) => num \*\* 2);

//returns a new array with all squares of the original numbers

arr.some((num) => num % 2 == 0); //returns true

//retuns a boolean value

//checks if any number is divisible by 2

arr.every((num) => num % 2 === 0); //returns false;

//returns boolean value

//checks if every number is divisible by 2

arr.reduce((prevValue, currValue) => currValue + prevValue, initialValueOfPrevValue);

//reduce method returns a single value

//in this case returns initialValueOfPrevValue + sum(arr);

setTimeout(func, timeOut);

//executes the function "func" after a min of timeout milliseconds

setInterval(func, timeout);

//executes the function "func" every timeout milliseconds

//to stop the setInterval function, use clearInterval function

//setInterval returns an unique id which should be used in the clearInterval function

const id = setInterval(func, timeout);

clearInterval(id);

**Chapter 232: Default Parameters**

//Default Parameters

const multiply = (a, b = 1) => a \* b;

multiply(4, 2); //returns 8

multiply(4); //returns 4;

//Earlier way of doing it

const multiply = (a, b) => {

    if(b === undefined)b = 1;

    return a \* b;

}

**Chapter 233, 234 and 235: Spread**

**Spread syntax allows an iterable such as an array to be expanded in places where zero or more arguments are expected OR an object expression to be expanded in places where zero or more key-value pairs are expected.**

//Spread

//Spread with function calls

const nums = [1, 2, 3, 4, 5];

//...nums mean 1, 2, 3, 4, 5

//So this means

console.log(1, 2, 3, 4, 5)

//is same as

console.log(...nums);

console.log(Math.max(...nums)) //is same as Math.max(1, 2, 3, 4, 5);

//and [...nums] is [1, 2, 3, 4, 5]

//Note that we cannot assign the spread to another variable

const x = ...arr //ERROR

const cats = ['Blue', 'Scoutt'];

const dogs = ['Rusty', 'Wyatt'];

const allPets = [...cats, ...dogs]; //['Blue', 'Scoutt', 'Rusty', 'Wyatt'];

//NOTE that strings are also iterables

console.log(..."HELLO"); //'H', 'E', 'L', 'L', 'O'

//Spread in objects

const obj1 = {

    key1: 1,

    key2: 2

};

const obj2 = {

    ...obj1,

    key3: 3

};

// obj2 = {

//     key1: 1,

//     key2: 2,

//     key3: 3

// }

//in case of conflict with keys

const obj1 = {

    key1: 1

};

const obj2 = {

    key1: 2

};

const obj3 = {

    ...obj1,

    ...obj2

};

//The key in obj1 will be overwritten by obj2

// obj3 = {

//     key1: 2

// }

//But if

const obj3 = {

    ...obj2,

    ...obj1

}

//then

// obj3 = {

//     key1: 1

// };

//the keys that come last are the winners

const arr = [1, 2, 3, 4];

console.log({...arr});

//Here the indices are used as keys

// {

//     0: 1,

//     1: 2,

//     2: 3,

//     3: 4

// }

//same case with strings

console.log({..."HELLO"});

// {

//     0: 'H',

//     1: 'E',

//     2: 'L',

//     3: 'L',

//     4: 'O'

// }

**Chapter 236: Rest Params:**

//The arguments object

function sumAll() {

    let total = 0;

    for(let i = 0; i < arguments.length; i++)total += arguments[i];

    return total;

}

sumAll(8, 4, 3, 2); //17

sumAll(2, 3); //5

**NOTES ABOUT arguments object**

1. **Available inside every function but not function expression**
2. **arguments is not an array. It is an array like object. Has length property. Does not have push and pop methods**
3. **contains all the arguments passed into the function**

**Rest Params: Collects all the remaining arguments in an array**

sumAll(8, 4, 3, 2); //17

sumAll(2, 3); //5

function sumAll(...nums) {

    //nums is an array

    let total = 0;

    for(let num of nums)total += num;

    return total;

}

**NOTE that Rest parameter must be last formal parameter otherwise it will result in an error**

**Chapter 237: Destructuring arrays**

const nums = [1, 2, 3, 4, 5];

const val1 = nums[0];

const val2 = nums[1];

//But there is a shorter way

const [val1, val2] = nums;

//val1 = 1, val2 = 2

const [val1, val2, ...restValues] = nums;

//val1 = 1, val2 = 2, restValues = [3, 4, 5]

**Chapter 238: Destructuring objects**

const user = {

    id: 23,

    email: 'user@gmail.com',

    firstName: 'fake',

    lastName: 'user'

}

const {lastName, firstName, id} = user; //note that these elements must have the same name as keys

console.log(id, firstName, lastName); //23, fake, user

//lets say we want the id with a different name userID

const {id: userId} = user;

//Now if we access id, we will get an error

console.log(userID); //23

//lets say we want to extract the property "birthYear" from the user object

const {birthYear} = user;

console.log(birthYear) //undefined

//we can set default values for the keys that are not present in the user object

const {birthYear = 'N/A'} = user;

const {birthYear: yearOfBirth = 'N/A'} = user;

console.log(yearOfBirth); //N/A

**Chapter 241: Introducing the DOM**

**Src:** [**https://developer.mozilla.org/en-US/docs/Web/API/Document\_Object\_Model/Introduction**](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

**The DOM is the object representation of the web page (document). It allows programming languages to manipulate the document (HTML).**

**The DOM was designed to be independent of any particular programming language, making the structural representation of the document available from a single, consistent API.**

1. The DOM is the Representation of a web page as objects.
2. Its your JS window into the contents of a web page
3. Its just a bunch of objects that you can interact with JS

**Chapter 242: The Document**

When the browser loads our webpage (Document), part of its process is involved in creating the document object (DOM) (which consists the object representations of the content in the web page) from HTML and CSS.

DOM is represented in the form of a tree data structure.

**Chapter 243, 244: getElementById, getElementsByClassName and getElementsByTagName**

We can select Elements in Document using getElementById, getElementsByTagName and getElementsByClassName etc.

//select html element using id

const element = document.getElementById('elementID');

//select multiple elements which has the same class name

const elements = document.getElementsByClassName('className');

//select all elements with tag name div

const divElements = document.getElementsByTagName('div');

**Chapter 245: query selector**

Easier way of selecting HTML elements. All-in-one method to select an element

//finds first h1 element

document.querySelector('h1');

//finds first element with the ID of red

document.querySelector('#red');

//finds first element with class of big

document.querySelector('.big');

**querySelectorAll** method gives us all the elements instead of first element

(Go thru video 245 again to understand nested selections after css videos)

**Chapter 246: innerHTML, textContent and innerText**

A picture containing website

Description automatically generated

Src: [https://stackoverflow.com/questions/35213147/difference-between-textcontent-vs-innertext](%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20https://stackoverflow.com/questions/35213147/difference-between-textcontent-vs-innertext)

<https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent>

Both innerText and textContent gives the content of an element but there are important differences:

1. **textContent gets the content of all elements, including <script> and <style> elements. In contrast, innerText only shows "human-readable" elements.**
2. **textContent returns every element in the node. In contrast, innerText is aware of styling and won't return the text of "hidden" elements (whose display is set to none).**
3. **Moreover, since innerText takes CSS styles into account, reading the value of innerText triggers a reflow to ensure up-to-date computed styles. (Reflows can be computationally expensive, and thus should be avoided when possible.)**
4. **Both textContent and innerText remove child nodes when altered, but altering innerText in Internet Explorer (version 11 and below) also permanently destroys all descendant text nodes. It is impossible to insert the nodes again into any other element or into the same element after doing so.**

When you are trying to alter the text, textContent is usually the property you are looking for.

When you are trying to grab text from some element, innerText approximates the text the user would get if they highlighted the contents of the element with the cursor and then copied to the clipboard. And textContent gives you everything, visible or hidden, including <script> and <style> elements.

**NOTE: We CANNOT use innerText or textContent for adding new HTML elements because the HTML elements are treated as strings. For this, we have to use innerHTML property**

**Chapter 247: Attributes**

We can access and set the attributes of a HTML element using dot notation, getAttribute() method or setAttribute() method

const a = document.querySelector('a');

//get attribute

const href1 = a.href;

const href2 = a.getAttribute('href');

//set attribute

a.href = 'https://www.google.com';

a.setAttribute('href', 'https://www.google.com');

**Chapter 248: Changing Styles**

We cannot access the styles which are defined in CSS files using DOM (Because DOM is the tree of nodes where each node specifies the properties of the HTML element and so CSSOM is the tree like DOM which contains CSS properties for a particular class or id or tag etc…. DOM and CSSOM are independent and that is why CSS styles are absent in DOM. But we can access inline styles and add inline styles because these are defined in the document only.)

const h1 = document.querySelector('h1');

console.log(h1.style.color) //undefined if there is no inline style defined even though the color is defined in css

h1.style.color = 'magenta' //add inline style in the document

**Chapter 249: ClassLists**

//add styles of specific classes to a HTML element

const h2 = document.querySelector('h2');

const h2Class = 'h2Class';

//1st way

h2.class = h2Class;

//But what if we want to add another class

const h2Class2 = 'h2Class2';

h2.class = h2.class == null? h2Class2: h2Class + ' ' + h2Class2;

//or

h2.setAttribute('class', `${h2.getAttribute('class') == null? h2Class2: h2.getAttribute('class') + ' ' + h2Class2}`);

//or we can use a property called classList

h2.classList.add(h2Class2);

//toggle a class (enable or disable the styles of that class) using classList

h2.classList.toggle(h2Class2); //if h2Class2 is already there in the classList, then it disables the styles of h2Class2 or of h2Class2 was

//not there previously, then it adds this class

h2.classList.toggle(h2Class2); //h2Class2 was disabled previously, now h2CLass2 gets enabled

**Chapter 250: Traverse parent, children and sibling**

//access parent node

const b = document.querySelector('b');

b.parentElement; //parent of b

b.parentElement.parentElement; //grandparent of b

b.children //children of b

b.nextElementSibling //next sibling

b.previousElementSibling; //prev sibling

**Chapter 251: Append and AppendChild**

//create a html element

const imgElement = document.createElement('img');

img.setAttribute('src', 'dummyLink');

//append it as the LAST ELEMENT of a html element

document.querySelector('body').appendChild(img); //img is the body's last child

//append method is more flexible than appendChild method (Note that append method is not supported in Internet Explorer)

//we can pass more than one node in append method

//we can also pass strings in append method

const newP = document.createElement('p');

newP.append("I am a new Paragraph.", "This will not result in an Error unlike appendChild method");

// <p>"I am a new Paragraph. This will not result in an Error unlike appendChild method</p>

const newb1 = document.createElement('b');

const newb2 = document.createElement('b');

newP.append(newb1, newb2);

//to add an element to the start instead of end, use prepend method

The insertAdjacentElement() method of the Element interface inserts a given element node at a given position relative to the element it is invoked upon.

<https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentElement>

See this also for adding an element adjacent to another element:

<https://developer.mozilla.org/en-US/docs/Web/API/Element/after>

<https://developer.mozilla.org/en-US/docs/Web/API/Element/before>

**Chapter 252: remove and removeChild**

//remove and removeChild

<ul>

    <li>First Item</li>

    <li>Second Item</li>

</ul>

//removing first li using remove child

const ul = document.querySelector('ul');

const li = document.querySelector('li');

ul.removeChild(li);

//or

li.parentElement.removeChild(li);

//removing first element using remove (again, remove is not supported in IE)

li.remove();

**Chapter 255: Events**

**A picture containing diagram

Description automatically generated**

**Chapter 256: Inline events**

<body>

    <h1>Events</h1>

    <button onclick="alert('You clicked me!'); console.log('The button is clicked!')">Click Me!</button>

</body>

Inline events are not recommended. Because it becomes more cumbersome.

**Chapter 257: onclick property**

const button = document.querySelector('#button');

button.onclick = () => {

    console.log('You clicked ME!')

}

**Chapter 258: addEventListener**

[**https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener**](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener)

//addEventListener (Recommended)

//specify the event type and a callback function to excecute

const h1 = document.querySelector('h1');

//notice that we omit 'on' from the event in addEventListener

h1.addEventListener('click', () => console.log("You clicked Me!"));

//why is addEventListener better

//because we can add multiple callback functions using addEventListener and

//we cannot do that using properties like onclick etc.,

h1.addEventListener('click', function1);

h1.addEventListener('click', function2);

//now when h1 is clicked, both function1 and function2 would execute

//we can pass another argument options also

h1.addEventListener('click', function3, {once: true});

//now when h1 is clicked, function3 will be executed only once and function3 will then be removed

//from the list of the callback functions

//next time when h1 is clicked, function3 would not be executed

//there are other properties in options, go thru mdn docs

//in the below case only function2 would be executed

h1.onclick = function1;

h1.onclick = function2;

**Chapter 260: Events and Keyword this**

**(Go thru this video again after the concept of this is cleared)**

**Chapter 261: Keyboard Events and event objects**

//events

const input = document.querySelector('input');

const button = document.querySelector('button');

input.addEventListener('keydown', (event) => {

    //event is the Event object which is passed by the

    //addEventListener which consists information about the event like which //key was pressed etc

    console.log(event);

});

button.addEventListener('click', (event) => {

    console.log(event);

})

**Chapter 262: Form events and preventDefault**

Consider the dummy form below

    <form action="random url">

        <input type="text">

        <button>Submit</button>

    </form>

const form = document.querySelector('form');

//whenever the submit button is clicked, we will move to the url

//specified in the action attribute

form.addEventListener('submit', () => {

    console.log("Submitted");

});

//but what if we dont want to move to another url

//for that, we have to use preventDefault method

form.addEventListener('submit', (event) => {

    event.preventDefault();

    console.log("Submitted");

});

**Chapter 265: input and change events**

const input = document.querySelector('input');

//event is change

//this event will fire ONLY when the input is removed from focussed or blurred and input value is changed

input.addEventListener('change', () => {

    console.log(input.value);

});

//event is input

//this event will fire everytime when the input value is changed irrespective of whether

//the input is removed from focus or not

**Chapter 266: Event Bubbling**

//event bubbling

//lets say we have the following html code

<p>

    <button>Click Me</button>

</p>

const p = document.querySelector('p');

const button = document.querySelector('button');

p.addEventListener('click', () => console.log("paragraph is clicked!"));

button.addEventListener('click', () => console.log("button is clicked!"));

//Now, what happens when button is clicked?

//as the button is also the child of the paragraph element,

//first button onclick would be executed and then paragraph onclick also would be executed

//events are executed (bubbled) from bottom to top and hence the name event bubbling

//to prevent that, we have to use an event method called stopPropagation

button.addEventListener('click', (event) => {

    event.stopPropagation(); //now the parent elements' onclick wont be executed

    console.log("Button is clicked!");

});

**Chapter 267: Event Delegation (Go thru again)**

**Chapter 273: The callstack**

**JS IS SINGLE THREADED.**

**WHAT DOES THIS MEAN??**

**AT ANY POINT OF TIME, A SINGLE JS THREAD IS RUNNING ATMOST A SINGLE LINE OF CODE. (IT DOES NOT DO TWO THINGS SIMULTANEOUSLY)**

[**https://medium.com/@Rahulx1/understanding-event-loop-call-stack-event-job-queue-in-javascript-63dcd2c71ecd**](https://medium.com/@Rahulx1/understanding-event-loop-call-stack-event-job-queue-in-javascript-63dcd2c71ecd)

Call stack is where JS Interpreter pushes the single line of synchronous code, the code gets executed and gets popped out and this process is repeated for the next line of code.

If JS Interpreter finds an asynchronous line of code like the **setTimeout, setInterval or requests etc (Which are btw webAPIs),** then the interpreter pushes this code into **Event table** which is responsible to push this code into **callback queue**

The **event loop** continuously checks whether the callstack is empty and whether all the synchronous code is done executing or not. If the callstack is empty and all the synchronous code is executed, then event loop pops the code from the **callback queue** and pushes in the **call stack** and this process is repeated until callback queue is empty.

**Go to the sources tab, add a break point on a function (**we can add a breakpoint by clicking on the number to the left side of the line) **and click (step into next function call button) to see the call stack**

This is basic debugging., more advanced debugging will follow in next lectures.

**Chapter 274: WebAPIs and Single Threaded**

**As we know JS is single threaded. (At any one point of time, JS will be executing only one line of code)**

But what happens when JS interpreter comes across asynchronous code? When JS interpreter comes across asynchronous code, it hands the code over to the browser and ‘tells’ the browser to put it in the callback queue after certain time (Web browsers are written in languages like C++ and not JS because JS is single threaded and synchronous)

**Chapter 275: Callback Hell**

**What is callback? Callbacks are just functions we pass in other functions as an argument**

**What is callback hell?** Lets say we have 4 **async** functions, function4 needs to be executed after function3 is executed and function3 needs to be executed after function2 is executed and function2 needs to be executed after function1 is executed. If we implement this, then we will get sort of nested callback functions

document.body.style.backgroundColor = 'green';

setTimeout(() => {

    document.body.style.backgroundColor = 'red';

    setTimeout(() => {

        document.body.style.backgroundColor = 'yellow';

        setTimeout(() => {

            document.body.style.backgroundColor = 'orange';

        }, 2000)

    }, 2000)

}, 2000);

setTimeout(function f1(){

    console.log("Hello1");

    setTimeout(function f2(){

        console.log("Hello2");

        setTimeout(function f3(){

            console.log("Hello3");

        }, 2000)

    }, 2000)

}, 2000);

setTimeout(function f4(){

    console.log("Hello4");

    setTimeout(function f5(){

        console.log("Hello5");

        setTimeout(function f6(){

            console.log("Hello6");

        }, 2000)

    }, 2000)

}, 1000);

**Tell the output of the above code 😊**

**Chapter 276: Demo: fakeRequest Using Callbacks**

A promise is an **object** representing the eventual completion or failure of an **asynchronous operation.**

**Note when we make a request, we usually pass in 2 callbacks (Success callback and failure callback)**

makeRequest(url,

    successCallback = (response) => console.log(response),

    failureCallback = (err) => console.log(err));

makeRequest(url1,

    successCallback = (response) => {

        console.log(response);

        makeRequest(url2, successCallback = (response) => {

            console.log(response);

            makeRequest(url3, successCallback = (response) => {

                console.log(response);

            }, failureCallback = (err) => console.log(err));

        }, failureCallback = (err) => console.log(err))

    }, failureCallback = (err) => console.log(err));

**Chapter 277: Demo: fakeRequest Using Promises**

To solve the problem of callback hell (caused by asynchronous functions which are dependent on each other), Promises were introduced.

**Definition: A promise is an object representing the success or failure of an asynchronous operation.**

READ THIS BEFORE GOING THROUGH THE BELOW CODE:

[**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Promise**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

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

**NOTE THAT Promises are not supported by IE**

A Promise is in one of these states:

pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

**Please keep in mind the following things while using promises**

1. While creating a Promise using new Promise(), there must be resolve or reject method used, otherwise the promise will always be in pending state.
2. **Promise.resolve(“some data”)** returns a resolved promise object with the data that is passed in it. Same case with **Promise.reject()** method
3. Consider the below code

Promise.resolve(42)

    .then((data) => {

        //console.log(data); 42

        //if we return nothing in this then, this translates to

        //return Promise.resolve(undefined);

        //if we return any data other than a promise, then it translates to

        //return data => return Promise.resolve(data);

        //or return a promise

        //return Promise.reject(420);

    })

    .then((data) => {

    })

1. Consider the below code

Promise.reject(42)

    .then((response) => {

        console.log(response); //this is never executed

        //in this then, there is no handle rejection method, therefore this then returns the rejected promise to next then

    })

    .then((response) => {

        console.log(response); //this is also never executed

    }, (error) => {

        console.log(error); //this is executed

        //and if we return nothing in this then(), then it translates to

        //return Promise.resolve(undefined);

    })

    .then((response) => {

        //the code inside this function is executed

    })

    .catch((error) => {

        //the code inside this function is not executed, because the error was handled in second then

        //also note that catch(rejectionFn) is syntactic sugar for then(undefined, rejectionFn)

    })

const delay = 1000;

const makeRequest = (url) => {

    const statusCode = (Math.random() \* 401) + 200;

    return new Promise(function executor(resolve, reject){

        if(statusCode >= 200 && statusCode < 400){

            setTimeout(() => {

                resolve(`Here is your data from ${url}`);

            }, delay \* 2);

        }

        else setTimeout(() => {

            reject(`Unable to get data from ${url}. SERVER ERROR!`);

        }, delay \* 3);

    });

}

const baseURL = 'www.yelpcamp.com/api/';

const url1 = `${baseURL}page1`;

//consider the below chaining

//in onRejected, we are not returning anything, By default it will be considered as resolve(undefined)

//and onResolved function in the next then function will be called

//So in the below code, if the url1 request fails, we are requesting for url3 in the second then() instead of stopping

makeRequest(url1)

    .then(function onResolved(data){

        console.log(data);

        const url2 = `${baseURL}page2`;

        return makeRequest(url2);

    }, function onRejected(data){

        console.log(data); //url1 request unsuccessful

    })

    .then(function onResolved(data){

        console.log(data);

        const url3 = `${baseURL}page3`;

        return makeRequest(url3);

    }, function onRejected(data){

        console.log(data); //url2 request unsuccessful

    })

    .then(function onResolved(data){

        console.log(data);

    }, function onRejected(data){

        console.log(data); //url3 request unsuccessful

    });

//if we want to stop making requests after a request fails, then we can do something like this

makeRequest(url1)

    .then(function onResolved(data){

        console.log(data);

        const url2 = `${baseURL}page2`;

        return makeRequest(url2);

    }, function onRejected(data){

        console.log(data); //url1 request unsuccessful

    })

    .then(function onResolved(data){

        if(data === undefined)return; //Dont execute the below code if previous onRejected method was executed

        console.log(data);

        const url3 = `${baseURL}page3`;

        return makeRequest(url3);

    }, function onRejected(data){

        console.log(data); //url2 request unsuccessful

    })

    .then(function onResolved(data){

        if(data === undefined)return;

        console.log(data);

    }, function onRejected(data){

        console.log(data); //url3 request unsuccessful

    });

//consider the below chaining, we have ommitted onRejected function in every then method.

//if we get rejected status, then the catch method is called

//NOTE that catch(onRejected) is just the syntactic sugar for then(undefined, onRejected)

//NOTE that the then method after catch is also executed

makeRequest(url1)

    .then(function onResolved(data){

        console.log(data);

        const url2 = `${baseURL}page2`;

        return makeRequest(url2); //suppose, we get a reject status here, then the JS Interpreter will check

        //if there is a onRejected method in next then(), if onRejected method is undefined, then the reject data

        //will be carried to next then(), until it encounters a reject method

    })

    .then(function onResolved(data){

        console.log(data);

        const url3 = `${baseURL}page3`;

        return makeRequest(url3);

    })

    .then(function onResolved(data){

        console.log(data);

    })

    .catch(onRejected = (data) => {

        console.log(data);

    })

    .then(function(){

        console.log("Running then after catch");

    })

//consider the below code

//if we return a promise, then the next then() method waits until the promise is completed or failed.

//but we are not returning a promise and so the 2nd then() is executed before 1st then()'s setTimeout is executed

new Promise((resolve, reject) => {

    resolve('I got resolved');

})

    .then(() => {

        setTimeout(() => {

            console.log('I am first then');

        }, 3000);

    })

    .then(() => {

        console.log('I am second then');

    })

**Chapter 280, 281 and 282: The async keyword, await keyword and error handling with async await**

Cleaner syntax for working with asynchronous code (Syntax sugar for promises)

**NOTE:**

1. Make any functions async using **async** keyword before function declaration or function expression

async function sum(a, b, c){

    return a + b + c;

}

const sum = async(a, b, c) => a + b + c;

1. All Async functions return **Promises.** We don’t need to explicitly return Promises.

async function sum(a, b, c){

    return a + b + c; //same as return Promise.resolve(a + b + c);

}

async function doNothing(){

    //Empty function

    //retuns Promise.resolve(undefined);

}

1. **await** keyword can **ONLY** be used inside async functions.
2. **ONLY Promises** can be awaited till the promise is resolved or rejected

Consider the below code

async function f1(){

    await setTimeout(() =>                                 //this is not a promise. So this line of code will not be awaited

        console.log("The code below this line will be executed when this setTimeout function is executed")

        , 2000);

    return 5;

}

let x = f1();

console.log(x); //Promise.resolve(5) will be returned and after 2000 ms, the statement inside the log method would be printed.

//if we want to return 5 after setTimeout function is executed, wrap the setTimeout function with Promise and then use await

async function f1(){

    await new Promise((resolve, reject) => {

        setTimeout(() => {

            console.log("The code below this line will be executed when this setTimeout function is executed");

        }, 2000)

    });

    return 5;

}

let x = f1();

console.log(x); //x is a pending promise with value undefined since setTimeout takes 2 sec to execute

//after 2 sec, the statement inside console.log would be executed and then Promise.resolve(5) would be returned

1. If the function throws an error, then the promise will be rejected. Use try-catch inside the async function.

const makePromise = () => {

    const random = Math.random();

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            if(random > 0.5)resolve();

            else throw new Error("Oh no!"); //Behind the scenes Promise.reject("Oh no!") is returned

        }, 1000);

    });

 }

//NOTE that in the below code, the catch function will not be executed even though we are throwing an error,

//because behind the scenes it is returning a rejected promise

let x = async () => {

    try{

        await makePromise();

    }

    catch(error){

        console.log(error); //OH, NO

    }

}

x();

//SEE THE BELOW CODE!

const f2 = async() => {

    throw new Error("Oh no!"); //Behind the scenes Promise.reject("Oh no!") is returned

}

try{

    f2();

}

catch(error){

    console.log(error);

}

//the error "Oh no!" will NOT BE CAUGHT.

//BUT WHY?

//REMEMBER THAT these async functions are executed after the main call stack is empty

//so before the execution of f2() is complete, try block is executed, f2() returns a pending promise

//and so no error is caught

//and after the main call stack is empty, f2() is executed and an error "Oh no!" is thrown and there

//is no catch block to execute it in the call stack

**Chapter 284: Intro to AJAX**

Making Requests in JS:

1. XMLHTTP
2. FETCH
3. AXIOS

AJAX: Asynchronous Javascript and XML.

If you go in the network tab in dev tools, you can see the requests made by the sites.

**Chapter 285: Intro to APIs**

In Layman terms, An API is the interface that lets a software1 to make requests to software2 and get response from the software2.

**Chapter 286: WTF is JSON**

In earlier days, to transfer data, XML (extensible markup language) was used. XML (Extensible Markup Language) is a markup language similar to HTML, but without predefined tags to use. Instead, you define your own tags designed specifically for your needs.

Text

Description automatically generated

But Now a days, JSON format is widely used. So when people say AJAX, they usually mean AJAJ (Asynchronous Javascript and JSON). JSON have only **STRING Keys** (DOUBLE QUOTES and NOT SINGLE QUOTES) and values can be **object, array, number, string, Boolean ( true or false ), or null.**

NOTE that JSON cannot have **undefined** value, **NaN** value and the numbers cannot have **leading zeroes**

Homepage of JSON: json.org

NOTE that JSON is not only for javascript but it is also used in many other languages.

To parse JSON string, JSON.parse() method is used

To convert a string into JSON, JSON.stringify() method is used.

READ MORE ABOUT JSON methods here: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON>

**Chapter 287: Postman**

**Postman** is used by many million developers. Used mainly for developing, testing, monitoring apis

Refer this link for more detailed status code: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

200’s: This class of status codes indicates the action requested by the client was received, understood, and accepted

300’s: These status codes are used in [URL redirection](https://en.wikipedia.org/wiki/URL_redirection)

400’s: This class of status code is intended for situations in which the error seems to have been caused by the client

500’s: The server failed to fulfil a request

In the response that we get, we also receive **HEADERS.** Headers are like meta data of the response. It contains information like the time when the request was made, the content-type, the server name, request id etc.

**HTTP Headers:**

If we go to the networks tab in dev tools, we can see **request headers** that has been sent to the serverand **response headers** that we received as a response. Headers tells about information like content-language, the content encoding, data format, date etc.

**Chapter 289: Making XHRs’**

**XHR** stands for **XMLHttpRequest.** Old way of making requests in JS. Does not support **promises .** Its all based on **callbacks.**

const req = new XMLHttpRequest();

req.onload = () => {

    console.log("ALL DONE WITH REQUEST");

    console.log(JSON.parse(this.responseText));

}

req.onerror = () => {

    console.log("ERROR");

}

req.open('GET', 'https://api.cryptonator.com/api/ticker/btc-usd');

req.send();

**Chapter 290: Fetch Api**

1. Newer way of making requests in JS
2. Supports promises
3. NOT SUPPORTED IN IE

fetch('https://api.cryptonator.com/api/ticker/btc-usd');

//this returns a promise

//use then or async await

fetch('https://api.cryptonator.com/api/ticker/btc-usd')

    .then((response) => {

        console.log("RESPONSE", res);

    })

    .catch((error) => {

        console.log("Error", error);

    });

//or

const makeRequest = async(url) => {

    const response = await fetch(url);

    console.log(response);

}

NOW when we make a request using fetch(), as soon as we receive the first bit of headers, it is going to resolve the promise. It is not going to wait for the entire data to be loaded. So to access the data, we use another method json() on response which also returns a promise.

fetch('https://api.cryptonator.com/api/ticker/btc-usd')

    .then((response) => {

        console.log("RESPONSE", res);

        return response.json();

    })

    .then((data) => {

        console.log(data);

    })

    .catch((error) => {

        console.log("Error", error);

    });

//or

const makeRequest = async(url) => {

    try{

        const response = await fetch(url);

        console.log(response);

        const data = await response.json();

        console.log(data);

    }

    catch(error){

        console.log(error);

    }

}

**Chapter 291: Intro to Axios**

1. Built on top of fetch (Makes making requests than fetch)
2. To use axios, include <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script> in your html file (Reference: <https://github.com/axios/axios>)

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

</head>

<body>

    <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script> <!--Make sure to include it before app.js script because app.js script uses axios-->

    <script src="app.js"></script>

</body>

</html>

axios.get(url)

    .then((res) => {

        console.log(res.data);

        //notice that we dont have call json method, res.json() to get the data

    })

    .catch((err) => console.log(err));

**Chapter 292: Setting Headers with Axios**

const getDadJoke = async() => {

    try{

        const config = {headers: {Accept: 'application/json'}}; //note this type of config will change api to api. Always refer the docs

        const res = await axios.get('https://icanhazdadjoke.com');

        console.log(res);

    }

    catch(error){

        console.log(error);

    }

}

**Section 29: Prototypes, classes and OOPS**

**What on earth are prototypes:**

**Prototypes are basically objects whose properties are inherited by the objects which have this prototype.**

Src: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes>

Every object in JavaScript has a built-in property, which is called its prototype. The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain. The chain ends when we reach a prototype that has null for its own prototype.

**Note: The property of an object that points to its prototype is not called prototype. Its name is not standard, but in practice all browsers use \_\_proto\_\_. The standard way to access an object's prototype is the Object.getPrototypeOf() method.**

When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case undefined is returned.

const myObject = {

    greet(){

        console.log("Hello");

    }

}

//accessing the prototype object of myObject

Object.getPrototypeOf(myObject);

//or

myObject.\_\_proto\_\_ //this is generally not preferred

There is an object called Object.prototype, and it is the most basic prototype, that all objects have by default. The prototype of Object.prototype is null, so it's at the end of the prototype chain:

**Overriding properties or Shadowing properties**

const myDate = new Date(1995, 11, 17);

console.log(myDate.getYear()); // 95

myDate.getYear = function() {

  console.log('something else!')

};

console.log(myDate.getYear()); // 'something else!'

This should be predictable, given the description of the prototype chain. When we call getYear() the browser first looks in myDate for a property with that name, and only checks the prototype if myDate does not define it. So when we add getYear() to myDate, then the version in myDate is called. This is called "shadowing" the property.

**If we add or modify any methods of prototype object, then all the objects which has this prototype object will inherit the changes. It is generally not a good idea to override or add methods in object’s prototype.**

**The below code is not recommended.**

Array.prototype.pop = () => {

    console.log("Sorry I want that element, I will not allow to pop");

}

console.log([3, 4, 5].pop());

**Note:**

Object.getPrototypeOf([]) === Array.prototype //true

Object.getPrototypeOf("") === String.prototype //true

**Setting a prototype:**

1. **Using Object.create() method**

const personPrototype = {

    greet() {

      console.log('hello!');

    }

  }

  const carl = Object.create(personPrototype);

  //carl object’s prototype will be set to personPrototype

  carl.greet();  // hello!

1. **Using a constructor**

Have some doubts, will revisit later

**Factory Functions:**

Factory functions are the functions which returns an object containing properties and methods.

//makeColor is a factory function which returns an object which has some properties and methods

function makeColor(r, g, b){

    const color = {};

    color.r = r;

    color.g = g;

    color.b = b;

    color.rgb = function rgb(){

        const {r, g, b} = this;

        return `rgb(${r},${g},${b})`;

    }

    color.hex = function hex() {

        const {r, g, b} = this;

        return "#" + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1);

    }

    return color;

}

const firstColor = makeColor(35, 255, 150);

firstColor.rgb(); //here “this” will refer to firstColor

const secondColor = makeColor(0, 0, 0);

secondColor.hex();

**Constructor Functions:**

**The problem with factory functions:**

//The problem with factory functions

console.log(firstColor.rgb === secondColor.rgb); //returns FALSE

//whats happening in the above case is that, by making factory functions, we are making copies of the methods defined in the function.

//Thus this approach is not efficient.

//consider the example below

console.log("Hello".slice === "World".slice) //returns true

//in the above case, the both slice methods are REFERRING TO THE SAME METHOD.

//Here slice method is not defined on every single string, but it is defined o//n the prototype which this string has

The convention of defining a constructor function is to use capitalize function name to denote that this is a constructor function.

/constructor function

//Note the first letter is capital

function MakeColor(r, g, b){

    this.r = r;

    this.g = g;

    this.b = b;

}

const firstColor = new MakeColor(r, g, b);

// What new does is

// 1. Creates a blank, plain javascipt object. {}

// 2. Links (sets the constructor of) the object to another object.

// 3. Passes the newly created object from step 1 as the "this" context

// 4. Returns "this" if the function does not return its own object

//So behind the scenes, implicitly, this is happening (only if we use the "new" keyword)

function MakeColor(r, g, b){

    const o = {}; //creating a blank, plane javascript object

    //Adds a property to the new object's (\_\_proto\_\_) that links to the constructor function's prototype object

//NOTE THAT MakeColor.prototype is DIFFERENT THAN Object.getPrototypeOf(MakeColor)

    Object.getPrototypeOf(o) === MakeColor.prototype;

    //passes the newly created object from step1 as the "this" context

    o.r = r;

    o.g = g;

    o.b = b;

    //returns this

    return o;

}

**NOTE THAT MakeColor.prototype is DIFFERENT THAN Object.getPrototypeOf(MakeColor). Every Function has a .prototype object whose constructor property refers to the same function**

**MakeColor.prototype.constructor === MakeColor; //true**

More Info: <https://stackoverflow.com/questions/38740610/object-getprototypeof-vs-prototype>

<https://chamikakasun.medium.com/javascript-factory-functions-vs-constructor-functions-585919818afe>

**Difference between factory functions and constructor functions:**

1. Objects using constructor functions are created using “new” keyword, whereas factory functions are those functions which return an object.
2. The use case of this function is that creating similar types of objectswith the new keyword, then later we can do instance checks using the instanceof keyword in JavaScript
3. Using factory functions, we were creating multiple copies of the methods, But with constructor functions, we can avoid that by defining methods on {constructorFunction}.prototype. (For more details, refer 4th point)
4. Objects created using Factory Functions have Object’s prototype

//makeColor is a factory function which returns an object which has some properties and methods

function makeColor(r, g, b){

    const color = {};

    color.r = r;

    color.g = g;

    color.b = b;

    color.rgb = function rgb(){

        const {r, g, b} = this;

        return `rgb(${r},${g},${b})`;

    }

    color.hex = function hex() {

        const {r, g, b} = this;

        return "#" + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1);

    }

    return color;

}

const firstColor = makeColor(35, 255, 150);

Object.getPrototypeOf(firstColor) === Object.prototype; //true

Object.getPrototypeOf(firstColor) === Object.getPrototypeOf({}); //true

**Whereas Objects created using constructor function have there prototype set to constructor function’s prototype object**

//constructor function

//Note the first letter is capital

function MakeColor(r, g, b){

    this.r = r;

    this.g = g;

    this.b = b;

}

const firstColor = new MakeColor(r, g, b);

Object.getPrototypeOf(firstColor) === MakeColor.prototype; //true

//So if we want to add methods in constructor functions without creating multiple copies of the methods UNLIKE Factory functiond

MakeColor.prototype.greet = () => console.log("Hello")

**Javascript classes:**

Using constructor functions, we have been defining properties inside the constructor function and we have been defining methods OUTSIDE on constructor function’s prototype method so that only one copy of method is shared between all objects created using the constructor function.

But as the methods are defined outside the constructor function, it looks disorganized. So to make it organised, CLASSES were introduced.

CLASSES are SYNTACTIC Sugar for the constructor functions.

//NOTE THAT THE CLASS NAME STARTS WITH CAPITAL LETTER

class Color{

    //creating constructor

    constructor(r, g, b, name){

        console.log("INSIDE CONSTRUCTOR");

        this.r = r;

        this.g = g;

        this.b = b;

        this.name = name;

        //behind the scenes, it works the same as the constructor function

        //it creates an empty object, sets the objects prototype to Color.prototype

        //"this" refers to the empty object created

    }

    //the below method is same as writing Color.prototype.greet = function(){return `HELLO FROM ${this.name}`;} OUTSIDE

//Notice the the greet() function does not start with function keyword

    greet(){

        return `HELLO FROM ${this.name}`;

    }

}

//Color.prototype.greet = function(){} we dont have to define methods outside

const c1 = new Color(255, 0, 0, "tomato");

**Extends and super keyword:**

class Pet{

    constructor(name, age){

        this.name = name;

        this.age = age;

    }

    eat(){

        return `${this.name} is eating!`;

    }

}

class Cat extends Pet{

    constructor(name, age, livesLeft = 9){

        super(name, age); //call the constructor of parent class

        this.livesLeft = livesLeft;

    }

    meow(){

        return 'MEOWW';

    }

}

class Dog extends Pet{

    constructor(name, age){

        super(name, age); //call the constructor of parent class

    }

    bark(){

        return 'WOOF';

    }

}

const monty = new Cat('monty', 9);

monty.eat(); //Monty is eating!

**SECTION 30: Mastering the terminal**

**Terminal and shell:**

Terminal is a program which runs a shell and allows us to enter commands, whereas shell is a program that actually processes the commands and outputs results.

Git Bash is an application for windows, which installs bash and git on our windows. Windows powershell is a shell which comes with the windows operating system

**Bash commands:**

1. **ls**: lists out all the directories and files in the current directory.
2. **pwd**: lists out the current directory we are in.
3. **cd**: cd refers to change directory. cd .. goes back, cd {directoryName} moves to the directory, just cd moves to the root directory.
4. **mkdir**: short for make directory, mkdir {directoryName1, directoryName2} creates 2 empty directories in the current path
5. **help**: (man command is not there in got bash),

Eg: ls –help

git help checkout

look for flags in help (We can also combine flags in a command. rm -r -f is same as rm -rf)

1. **touch**: Create files, touch abc.txt creates abc.txt file in current directory.
2. **rm**: remove files, use “rm -r -f” or rm -rf to delete directories which are not empty. (refer help for more information on flags)
3. **rmdir**: used to remove empty directories