Javascript Basics

01

History of Javascript

→ In 1995, A Netscape (browser) programmers named Brandon Eich developed a scripting language in just 10 days

→ At that time, Java was famous programming language. So for marketing purpose 'Livescript' changed into 'Javascript'

Mocha → Livescript → Javascript.

→ In 1997, Microsoft copied Javascript onto its own browser "Internet Explorer" and named it "Jscript"

→ In browser war (Netscape vs Internet Explorer) Ecmascript was born

→ Ecmascript is an international industry association founded in 1996, dedicated to the standardization of information and communication systems.

→ Thus the problem of implementing JS in different browsers was solved.

→ First Ecmascript → 1997
ES2 → 2009 (lots of new features)
ES3 → 2015 (Biggest update in Javascript)

How to execute Javascript

→ Javascript can be executed right inside every browser by just opening the browser console and writing the javascript

→ Another way to execute javascript is a runtime environment like Node.js.

→ Yet another way to execute javascript is by inserting it inside <script> tag of an HTML document

→ And the most important thing is it is Synchronous single-threaded language.

Variables & Interview

Variables

Javascript supports var, let and const

Ex → var a = 5 // a stores 5

* The 'var' keyword was used in all javascript code from 1995 to 2015.

The 'let' and 'const' keywords were used in ES3 (2015)

When to use var, let and const?

1. Always declare 'const' if the values shouldn't be changed
2. Always use 'const' if the type shouldn't be changed
3. Only use 'let' if you can't use 'const'.

Var vs let in Javascript

1. 'Var' is globally scoped while 'let' & const are block scoped
2. 'var' can be updated & redeclared within its scope.
- 'let' can be updated but can't be redeclared in the particular scope it was declared and defined previously
- 'const' can neither be updated nor be re-declared

1. → Before ES6 (2015), Javascript had 'Global Scope' and 'function Scope'

→ ES6 introduced two new Javascript keywords: 'let' & 'const' which are 'block scoped'

- { var a = 2; } // a can be accessed outside this scope
- { let b = 2; } // b can't be accessed outside the scope

→ Redeclaring variables was a problem before ES6

Before	After
var a = 2;	let a = 2;
{var a = 3;}	{let a = 5;}
// here value of a is 3	// here value of a is 2, 5 is limited to the scope it is declared/defined inside.

let Hoisting

Variables defined with 'var' are hoisted to the top and can be initialized at any time. meaning the variable can be used before it is declared.

Ex → carName = "Volvo";
var carName;

But variables defined with 'let' are also hoisted to the top of the block, but can't be initialized.

carName = "Volvo"
let carName = "Audi" // Not allowed.

const keyword

→ It was introduced in 2015 (ES6). It can't be redeclared, reassigned and is block scoped.

Ex → const PI = 3.14159265
PI = 3.14 // Not possible
PI = PI + 10; // Not possible

→ It must be assigned during its declaration

→ Here we can't reassign a value, array or object but can change the element of 'const array' and 'const object'.

Ex → const cars = ["volvo", "audi", "bmw"]
cars[0] = "Toyota"; // can be changed the value
cars.push("Audi") // elements can be added.
cars = ["Toyota", "volvo", "audi"] // Not possible.

Ex → const car = {type: "Audi", model: "Q50", color: "white"};
car.color = "red"; // possible for changing values
car.owner = "Subham"; // possible for adding property.
car = {type: "Audi", model: "Q50", color: "white", owner: "Subham"}
// Not possible.

→ Hoisting of 'const' is same as 'let'.

Datatypes

There exist two types of datatypes :-

I. Primitive

1. String → let color = "Blue";
2. Symbol → let me = Symbol("I am a Good Boy");
3. Number → let N = 16;
4. Null → let a = null;
5. Undefined → let b = undefined; // let b; // Nothing is assigned so undefined
6. Boolean → let x = true;
7. BigInt → let y = BigInt("56789101112 13"); // represent values greater than 253
   ↳ let y = 56789101112 13n;

II. Objects

1. Object → let car = {type: "Audi", Owner: "Subham"};
2. Array → let car = ["Audi", "Subham"];
3. Date →

*Notes

→ Javascript evaluates expression from left to right.

Ex → let x = 16 + "volvo" // output → 16volvo ~~// treated as "16" + volvo~~

- it is treated as "16" + "volvo" as "volvo" is a string and is a the rightmost.

Ex → let x = "volvo" + 16 // output = volvo16

- when concating anything with 'String', JS will convert everything to String.

Ex → let x = 16 + 4 + "volvo" // output → 20volvo

→ Javascript has dynamic types.

Ex → let x;
x = 5; x = "John"; // x is holding a number as well as a string.

→ In JS strings, we can use quote inside it, as long as they don't match the quotes surrounding the string.

→ Extra large or extra small numbers can be written like —
let y = 123e5; // → 12300000
let z = 123e-5; // → 0.00123

→ Number type of javascript is 64-bit floating point

→ Any variable can be emptied by setting the value to undefined. The type will show undefined.

→ For converting string to number, putting (+) before string:
Eg → let str = "9";
console.log (typeof (+str));

→ For converting number to string, add empty string to number
Eg → let num = 10;
console.log (typeof (num + ""));

→ We can push, pull, shift, unshift with the properties and elements in an array and object.

→ Elements in an object can be accessed by two ways:—
(i) console.log (car.type)
(ii) console.log (car['type'])

Operators

1. Arithmetic → (+, -, *, **, /, %, ++, --)
2. Assignment → (=, +=, -=, *=, /=, %=, **=)
3. Comparision → (==, !=, ===, !==, >, >=, <, <=, ? :)
4. Logical → (&&, ||, !)
5. typeof → (typeof, instanceof)
6. String → (+, +=, <, >, =
7. Bitwise → (&, |, !, >>, <<

Date
Page 06

*Notes

→ For comparison operators —

if x = 5 → equality or abstract comparision

↳ x == 8 → false
x == 5 → true
x == "5" → true

↳ x == 5 → true
↳ identity or strict comparision
x == "5" → false
↳ // type doesn't match

↳ x != 5 → false
x != "5" → true // value matches but type doesn't match
x != 8 → true

↳ "2" > "12" → true // 1 < 2

→ For Strings

let x = "Subham", y = "Das"

x + y // SubhamDas

↳ x = "Subham", y = "Das"

z = x + y // SubhamDas

↳ Concatenation operator

Undefined

→ Accessing an uninitialized variable returns undefined
→ Accessing a non existing property of an object returns undefi
→ Accessing a out-of-bound array element returns undefined.

Null

→ Null means 'no value' assign to variable
→ typeof null returns 'object'
→ Null is treated as 'false value'.

Truthy Values

It is a value that is considered true when encountered in a Boolean context

Ex → true, {}, [], 42, "0", "false", new Date(), -42, 12n, 3.14, -3.14, Infinity, -Infinity

Falsy Values

It is a value that is considered false when encountered in a Boolean context.

Ex → Undefined, null, NaN, false, "", 0, -0,

Conditionals

Javascript supports (i) if
(ii) if ... else
(iii) if ... else if ... else
(iv) switch ... case.

Loops

Javascript supports while loop, do...while loop and for loop

For ... in loop

```
Ex → let obj = {
        Subham : 90;
        Rohan : 50;
        Rohit : 65;
        Ritika : 95;
     }
for (let a in obj) {
        Console.log("Marks of " + a + " are " + obj[a]);
}
```

↳ property ↳ values

It basically works through the keys of an object.

For ... of loop

```
for (let b of "Harry") {
    console.log(b)
}
```

→ H a r r y

It basically works through the values of an object.

Functions

```
function fun(x, y) {
    return (x+y);
}
```

- datatype is optional
- Explicit Return

```
fun = (x, y) => {
    return (x+y);
}
```

- Implicit Return

```
fun = (x, y) => x+y
```

↳ while calling fun(3,4)+3
↳ When parenthesis is used no need to use return keyword but return is required curly braces
↳ Parenthesis is required when curly braces

↳ const fun = (x, y) => (x+y);
console.log(fun(4,5)); // 9

↳ To return an object
const fun() => ({name: "Subham"})

*Notes

```
let marks = {
    subham = 65
    debu = 70
    samik = 65
}
```

addOne(5);
function addOne(num) {
return num+1;
}
↑ won't show error when called before definition

addTwo(5);
const addTwo = function(num) {
return num+2;
}
↑ will show error bcz here we are storing a function in a variable

```
for (let i=0; Object.keys(marks).length; i++) {
    console.log("The marks of " + Object.keys(marks)[i] + " are " +
        marks[Object.keys(marks)[i]])
}
```

(Object.keys(marks)[i] → key)

↳ marks of the given key = value

Strings

It can be made by two ways (i) using single quote
(ii) using double quote

⇒ Template literals

Strings can also be made using backtick but it's not a good practise

In order to define a string by putting previously declared string variables, we say this process as Template literals.

Ex → let b="Subham"
let c="programmer"
let e = `${boy2} is a ${c}`
let e = `${b} is a ${c}` // Subham is a programmer

*Notes
→ With template literals, it is possible to use both single as well as double quotes inside a string
→ Inserting variables directly in template literals is called string interpolation. We do this by using ${variable}
→ Escape sequences can be used to nest the quotes.

⇒ String Methods

var str1 = "Subham"; var str2 = "Das"; a="I am a good boy";

1. console.log(str1.charAt(5)); // m
2. console.log(str1.charCodeAt(3)); // 98 → returns UTF-16 code (0-65535)
3. console.log(str1.concat(b)); // Subham Das
4. console.log(String.fromCharCode(97,98,99)); // abc
5. console.log(a.indexOf("word to be searched for from sentence "a")");
6. console.log(a.lastIndexOf("index of char to be found from sentence "a")");
7. console.log(a.replace("good","bad")); // I am a bad boy.
8. console.log(a.search("a")) // 2
9. console.log(a.slice(3,8)) // am a g //
10. console.log(b.split(",") // splits the sentence with words wherever "," is applied.
11. console.log(a.substring(3,10)) // m a goo
12. toLowerCase(String) → converts the whole string into lower case
13. toUpperCase(String) → " " " " " " Upper case.

14. console.log(a.includes("good")) // true
15. console.log(a.endsWith("boy")) // true
16. console.log(a.replaceAll("Microsoft", "Amazon") // replaces all occurences

*Notes

→ If a parameter is -ve, the position is counted from the end of the string.
→ Difference between substring() and slice() is that start and end values less than 0 are treated as 0 in substring()
→ The difference between substr() and slice() is that the second parameter specifies the length of extracted part.
→ In case of substr(), if the 1st parameter is -ve, counting of the position starts from the last index
→ replace() doesn't change the string it is called on. It returns a new/modified string. replace() replaces only the 1st match (from left to right)
→ By default, replace() is case-sensitive
→ To replace case insensitive, "/i" is used.
Ex → str.replace(/WORD/i, "Word to be replaced with");
→ As replace() replaces the first match only, /g could replace all the matches.
Ex → str.replace(/word/g, "Word to be replaced with")
→ String.trim() also support trimStart() and trimEnd(). to produce appropriate records.
→ charCodeAt() method makes the string look like arrays If no character is found, [] returns undefined while charAt() returns an empty string

Arrays

In Javascript, we can store different values in an array.
Ex → let a = [true, "Subham", 6, 'Das']

Notes
→ Arrays are mutable. They can be changed.
→ In Javascript, arrays are objects

⇒ Array Methods

let a = ["Subham", "Prasad", "Das"], b = ["!"]

1. a.push("The Great") // ['Subham', 'Prasad', 'Das', 'The Great']
2. a.pop(); // ['Subham', 'Prasad']
3. a.shift(); // ['Prasad', 'Das']
4. a.unshift("The") // ['The', 'Subham', 'Prasad', 'Das']
5. a.indexOf("Das") // 2
6. a.includes("Das") // true
7. a.concat(b) // ["Subham", "Prasad", "Das", "!"]
8. a.sort() // ["Das", "Prasad", "Subham"]
9. a.toString() // converts an array ~~with numbers in~~ elements to string (Subham, Prasad, Das)
10. a.join("-") // Subham-Prasad-Das
11. a.reverse() // ["Das", "Prasad", "Subham"]
12. a.splice(1, 1, "") // ["Subham", "", "Das"]
    (1 → position to start adding; 1 → no. of elements to remove; "" → elements to replace.)
13. a.slice(1,2) // ["Subham", "Das"] → 2 not included

Notes
→ Javascript variables can be objects. Arrays are special kinds of objects. Because of this, arrays can have variables of different types. It can be objects, it can be functions or an array.

→ typeof operator returns "array" as "object"
→ pop() updates the original array. It returns the value they did their operations on.
→ delete is an operator
→ concat() returns a new array, doesn't change the previous.
→ Last element of the array can be accessed by using negative value.
→ Adding elements with high indexes can create "holes" in array
Ex → let fruits = ["Apple", "Lemon", "Orange"]
fruits[6] = "Mango" // ['Apple', 'Lemon', 'Orange', -, -, -, 'Mango']
→ .isArray() method can be used to distinguish an array.
→ .push() returns the new array length thus updates the array
→ .shift() returns the value that was shifted out.
→ .unshift() returns the new array length.
→ delete make the array remain with undefined holes. so better to use pop() & shift() - It also doesn't affect the array side
→ concat() doesn't change the existing arrays. It returns the new array
→ splice() returns an array with the deleted array item
→ splice() can be used to remove elements from an array.
Ex → const fruits = ["Banana", "Orange", "Apple", "Mango"]
fruits.splice(0, 1) // 0 → new element should be added
1 → no. of elements should be removed
As rest of the parameters are skipped no new elements are added but one item got removed.
→ slice() creates a new array, thus don't change the previous array.

Using loop in Array

using for loop

```
let num = [3, 54, 4, 55, 6];
for (let i = 0; i < num.length; i++) {
    console.log(num[i]);
}
```

using for-each loop

```
let num = [3, 54, 4, 55, 6];
num.forEach((element) => {
    console.log(element * element)
})
```

Notes:-

→ forEach contains a method which will be executed for each and every element of the array.

using for ... of loop

```
let num = [1, 2, 3, 4, 5];
for (let i of num) {
    console.log(i);
}
```

→ 1 2 3 4 5

Notes:-

→ when written console.log(num[i]) → it will print → [2, 3, 4, 5, 1]

using for ... in loop

```
let num = [1, 2, 3, 4, 5]
for (let i in num) {
    console.log(i);
}
```

→ 0 1 2 3 4

Notes:

→ As we know real for... in is basically used in case of objects, where it gives out the keys of the targeted object.

→ But when used with arrays, it gives out the index.

→ when written console.log(arr3[i]) → it gives the elements.

→ for... of will be executed over array & string but not on objects.

→ when for... of is used over string, when the 'iterator' is printed, it results in printing "undefined" for the 'length of the string' times. And when 'iterator along with string' is printed, it results of printing the elements of the string. [Reverse]

→ when for... of is used over array, when the 'iterator' is printed, it will print the elements. when 'iterator along with array' is printed, the array will be printed from index 1 to last & at last it will print 'undefined'.

→ when for... of is used on object, it will give syntax error.

→ when for... in is used on string, when traversed with 'iterator', it will print the index of the string. when printed with 'iterator along with string', elements of the string will be printed.

→ when for... in used on array, when traversed with iterator', the property (index) of the array will be printed when traversed with 'iterator along with array', elements of the array will be printed.

→ when for... in used for objects, when traversed with 'iterator', prints out the properties of the object. when traversed with 'iterator along with object', prints the elements of object

Map()

Returns a new array by performing specified operations on the given array.

Ex → const num = [1, 2, 3]

const newArr = num.map(Math.sqrt);

console.log(newArr);

↳ [1, 4, 9]

Ex → const num = [65, 44, 12, 4]

const newArr = num.map(myFunction);

↳ [650, 440, 120, 40]

function myFunction(numb) {

return numb*10;

}

Notes :-

→ It creates a new array from calling a function for every element of the array.

→ It doesn't execute the function for empty elements.

→ It doesn't change the original array.

→ Difference between forEach() and Map() is forEach() effect the original array but map() creates a new array

→ Syntax : a.map((value, index, array) => {

return value, index;

})

Filter()

→ Returns a new array abide by the condition we specified inside the filter function.

→ Similar methods are .every() [acts as &&], .some() [acts as ||]

Ex → let arr1 = [44, 66, 77, 33, 22];
let arr2 = arr1.filter((value) => {
return (value > 50);
})
console.log(arr2); // [66, 77]

Notes:-
→ let arr1 = [22, 33, 44, 66, 77, 88]
let arr2 = arr1.map((value) => { return value > 50})
↳ ['false', 'false', 'false', 'true', 'true', 'true']
→ forEach returns 'undefined'
→ Neither of forEach, map nor filter affect the original array. forEach returns undefined whereas map & filter return new arr.
→ forEach is a very generic method. We should only try to use it when we wanna perform a specific task on array elements iteratively.
→ map should be used to do a specific operation on the given array without changing it but getting a new array whereas filter holds the same usage but is used to filter out and fill up the new array based on the condition given.

Reduce
It reduces the array elements to a single value according to the condition given and thus returns a single value.
Ex → let arr1 = [1, 2, 3, 4, 5]
let arr2 = arr1.reduce((v1, v2) => {
return v1 + v2;
})
↳ 15

Numbers

→ Numbers can be written with or without decimal

◆ Ex → let x = 3.14, y = 3, z = 123e5 // 12300000
a = 123e-5 // 0.00123

Notes:

→ Numbers are always 64 bit floating point.

→ This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62 and the sign in bit 63.

→ Integers (numbers without a period or exponent notation) are accurate up to 15 digits

→ Floating point arithmetics are not always 100%

→ If two nos are added, result will be a number But if a string is added to itself of a number, result will be a string anyway.

→ let x = 10; let y = 20; ~~Console~~.log("The result is " + x + y);
30 1020

→ let x = 10; let y = 20; let z = "30"; console.log(x + y + z);
1020 30 3030

→ let x = 10; let y = 5;
console.log(x * y) // 50
console.log(x / y) // 2
console.log(x + y) // 105
console.log(x - y) // 5

Thus, Javascript will try to convert strings to numbers in all numeric operation except the contatination operator (+)

→ NaN is a JS reserved keyword indicating that a number shown or in the operation is not going legal.

Ex → let x = 100 / "Apple"; → NaN (Not a Number)

→ isNaN() method is used to find out whether the shown result is number or not. It returns boolean

→ let N = NaN ; M = 5 ; console.log(N+M); // NaN
→ let N = NaN; M = "5"; console.log(N+M); // NaN5
→ typeof (NaN) // Number
→ Infinity or -Infinity is the value in JS, will return if we calculate a number outside the largest possible number.
→ console.log (2/0); // Infinity
console.log (-2/0); // -Infinity
→ typeof (Infinity) // Number
→ Number conversion can be done through —
let num = 32;
num.toString(16) // 20 (Hexadecimal)
num.toString(10) // 32 (Decimal)
num.toString(8) // 40 (Octal)
num.toString(2) // 100000 (Binary)
→ BigInt variables are used to store big integer values that are too big to be represented by normal javascript number. They can be created by appending 'n' to the end of an integer or call BigInt().
e.g. → let x = 99999999999999999n
let y = BigInt(99999999999999999);
→ typeof (BigInt) // bigint
→ Arithmetic operation between BigInt and a number cannot be done formally or can be done by typecasting number.
→ BigInt can't hold decimal values.
→ Number.isInteger() → checks if the parameter is int. or not
Number.isSafeInteger() → checks if the parameter is inside the range of BigInt or not.
→ Safe Integers are all integers from $-(2^{53}-1)$ to $+(2^{53}-1)$

Number Methods

1. toString() Method

It returns a number as a string.

2. toExponential() Method

It returns a string, with a number rounded and written using exponential notation.

A parameter defines the number of characters behind the decimal point.

Ex → let x = 9.656;

x.toExponential(2); // 9.66e+0

x.toExponential(4); // 9.6560e+0

x.toExponential(); // 9.656e+0

3. toFixed() Method

It returns a string, with the number written with a specified number of decimals.

Ex → let x = 9.565;

x.toFixed(0) // 10

x.toFixed(2) // 9.57

x.toFixed(4) // 9.5650

4. toPrecision() Method

It returns a string, with number written as of with parameter

Ex → let x = 9.565;

x.toPrecision(); // 9.565

x.toPrecision(2); // 9.6

x.toPrecision(4); // 9.565

5. Number() Method

It converts Javascript variables to Number

Ex → Number(true); // 1 Number("10.33"); // 10.33

Number(10) // 10 Number("10 33"); // NaN

Number(" 10 ") // 10 Number("Subham") // NaN

6. parseInt() Method

It parses a string and returns a whole Number

Ex → parseInt ("-10") // -10 parseInt ("-10.33") // -10
parseInt ("-10.33") // -10 parseInt (" 10 6") // 10
parseInt ("10 years") // 10 parseInt ("years 10") // NaN

7. parseFloat() Method

→ It parses a string and returns a number.
→ It works as parseInt() but here decimal point is allowed.

8. Number.parseInt() Method

→ It parses a string and returns a whole Number
→ Same as parseInt()

9. Number.parseFloat() Method

→ Returns Nos with floating Nums.
→ Same as parseFloat().

Notes :-

→ These methods can only be accessed like Number.—() not with some variable.—() then it will return (undefined) / Error.

/* Objects

```
let obj = { name : "Subham",
            Age : 24
          };
```

console.log(obj) // To acces keys & object

To access the values // console.log(obj.name) // Subham
console.log(obj["name"]) // Subham

To modify // obj.name = "Shibam"

To freeze // Object.freeze(obj);

Date Objects

→ Date objects are static

Ex → const d = new Date(); // Sun Aug 13 2023 22:34:24 GMT +0530 (IST)

const d = new Date("2022-03-25"); //

→ Date objects are created with the new Date() constructor

→ Ways to create a new date object:

1. new Date()

creates a date object with the current date & time.

Ex → const d = new Date() // Sun Aug 13 2023 22:40:44 GMT +0530 (IST)

2. new Date(date string)

creates date object from a date string from the parameter

Ex → const d = new Date("October 13, 2014") //

Mon Oct 13 2014 11:13:00 GMT +0530 (IST)

3. new Date(year, month)

4. new Date(year, month, day)

5. new Date(year, month, day, hours)

6. new Date(year, month, day, hours, months)

7. new Date(year, month, day, hours, months, seconds)

8. new Date(year, month, day, hours, months, seconds, ms)

9. new Date(miliseconds)

Notes:-

→ Javascript counts months from 0 to 11. ie Jan = 0 Dec = 11

→ Specifying a month higher than 11, won't give in error but adds the overflow to the next year.

Ex → const d = new Date(2018, 15, 24, 10, 33, 30);

is same as const d = new Date(2019, 3, 24, 10, 33, 30);

And that's how, the overflow also works for all

→ 6 numbers as the parameter specifies year, month, day, hour, minute, second.
→ 5 numbers specify year, month, day, hour, minute and so on till 2 numbers for hour and minutes.
→ Month cannot be ommited, if ommitted, the number in the parameter will be treated as miliseconds.
→ One or two digit years will be interpreted as 19XX
Ex → const d = new Date (99, 12) // Dec 1999
→ Javascript stores dates as miliseconds since January 01, 1970.
Ex → const d = new Date (100000000000)
↳ Sat Mar 03 1973 15:16:40 GMT +0530 (IST)
↳ Jan 01, 1970 + 100000000000 ms
And that's how we can set the miliseconds in -ve for going back from January 01, 1970.
→ For 24 hours —
↳ const d = new Date (24 * 60 * 60 * 1000); or
↳ const d = new Date (86400000)
↳ Jan 02, 1970
→ Javascript will (by default) output dates using toString() method.

⇒ Some Date Methods

1. toDateString() → converts a date to more readable:
Ex → const d = new Date()
d.toDateString(); // → Sun Aug 13 2023
2. toUTCString() → converts a date to UTC method.
Ex → const d = new Date()
d.toUTCString() // → Sun, 13 Aug 2023, 17:39:00 GMT
3. toLocaleTimeString() → 11:30:26 PM

Date 03

⇒ Date input formats

Type	Example
ISO Date	"2015-03-25" (The International Standard)
Short Date	"03/25/2015" or "15-03-25"
Long Date	"Mar 25 2015" or "25 Mar 2015"

- ISO format follows a strict standard in Javascript

⇒ ISO Dates - Notes:

→ ISO 8601 is the international standard for the representation of dates and times.

→ The ISO 8601 syntax (YYYY-MM-DD) is also the preffered Javascript date format.

→ When just year & month is specified, day becomes 01 when just year is specified, day and month becomes 01 and january respectively.

→ Date and time can be given as parmeter.

Ex → const d = new Date ("2015-03-25T12:00:00Z");

T → Date and time separator.

Z → defines GMT time

Z can be ommitted, by its place +HH:MM or -HH:MM can be added.

Ex → const d = new Date ("2015-03-25T12:00:00-06:30");

↳ Wed Mar 25 2015 23:00:00 GMT +0530 (IST)

Reduces 1 hour ie. from +0530 to (-0630) so the time wont be 00:00:00 but 23:00:00

→ Short dates must remain under its format i.e.) "03-25-2015" or "15/03/25" will give error, undefined or

→ In case of Long dates, the month and day section can be swapped according to user convinience. words are case insensitive and can be written full or abbreviated.

⇒ Get Date() Methods

→ The get methods return local time.

→ The get methods returns information from existing date objects. In a date object, the time is static. The "clock" is not "running".

1. getFullYear() Method

returns the year of a date as a four digit number.

Ex → const d = new Date("2021-03-25");
console.log(d.getFullYear()); // 2021

2. getMonth() Method

returns the (month no. - 1)

Ex → const d = new Date("2021-03-25");
console.log(d.getMonth()); // 2 because Jan 0, Feb 1

Ex → const months = ["January", "February", "March", "April", "May", "June", "July", "August", "September", "october", "november", "december"]
const d = new Date();
let month = months[d.getMonth()];
console.log(month); // August

3. getDate() Method → returns value from 0 - 31
4. getHours() Methods → returns value from 0 - 23
5. getMinutes() Method → returns value from 0 - 59
6. getSeconds() Method → returns value from 0 - 59
7. getDay() Method → returns value from 0 - 6 // 0 - Sunday, 6 - Saturday
8. getTime() Method → returns no. of miliseconds from 01 Jan, 1970.
9. Date.now() Method → similar to getTime().

Set Date() Methods

→ Set Date() Methods let set date values for a date object

1. setFullYear(arg).
2. setMonth(arg)
3. setDate(arg)
4. setHours(arg)
5. setMinutes(arg)
6. setSeconds(args)

Notes:-

→ setFullYear(args) can additionally set month & day.

Ex→ const d = new Date();
d.setFullYear(2023, 08, 14);
console.log(d); // Mon 14 Sept 2023 11:35:17 GMT....

→ SetDate() can be used to add days to a date also

Ex→ const d = new Date();
d.setDate(d.getDate()+50);
console.log(d); // ↳ gets today's date ↳ adds 50 days

→ If adding days shifts the month or year, the changes are handled automatically by the Date object.

→ Dates can be compared for...

Ex→ let text = " ";
const today = new Date();
const someday = new Date();
someday.setFullYear(2100, 0, 14);
if (someday > today)
text = "Today is before Jan 14, 2100.";
else
text = "Today is after Jan 14, 2100";

Math Object

→ It allows to perform mathematical operations on num
→ Unlike other objects, It doesn't work with constructor
→ Math object is static. So all methods and properties can be used without creating a Math object first.
→ Syntax for any Math property is : Math.property
→ Javascript provides 8 mathematical constants that can be accessed as Math properties —

(i) Math.E → returns Euler's number
(ii) Math.PI → returns PI
(iii) Math.SQRT2 → returns square root of 2
(iv) Math.SQRT1_2 → returns square root of 1/2
(v) Math.LN2 → returns the natural log of 2
(vi) Math.LN10 → returns the natural log of 10
(vii) Math.LOG2E → returns base 2 log of E
(viii) Math.LOG10E → returns base 10 log of E.

⇒ ## Math Methods

(i) Math.round(num) → returns num rounded to its nearest int
(ii) Math.ceil(num) → returns num rounded upto its nearest int
(iii) Math.floor(num) → returns num rounded down to its nearest int
(iv) Math.trunc(num) → returns integer part of x.
(v) Math.sign(num) → returns 1 (if +ve), 0 (if NULL) or -1 (if -ve)
(vi) Math.pow(x, y) → returns x to the power y
(vii) Math.sqrt(x) → returns square root of x
(viii) Math.abs(x) → returns positive value of x
(ix) Math.sin() & Math.cos() returns the value accordingly
(x) Math.min() & Math.max() take many args & returns accordingly
(xi) Math.cbrt(x) → returns cuberoot of x
(xii) Math.log(x), Math.log2(), Math.log10()

(xii) Math.Random() → returns a random number between 0 (inclusive) and 1 (exclusive)
→ Math.floor() along with Math.random() can generate random numbers till the specifications.
Ex→ Math.floor(Math.random()*10); → 0-9 // 10 excluded
→ Math.floor(Math.random()*100)+1; → 1 to 100

Javascript In Browser

→ Javascript can be added to HTML page by two ways —
(i) linking <script src="script.js"></script> to HTML in <head>
(ii) writing all the script between <script> tag.
→ Console is an object which has some methods —
(i) console.assert(cond") → returns error if condition is false.
(ii) console.clear() → clears the console.
(iii) console.log("m") → prints its arguments.
(iv) console.count() → counts the no. of iterations done over it.
(v) console.debug() → outputs a msg to the web console at "debug".
(vi) console.error("m") → prints the argument as "error".
(vii) console.info("m") → prints the argument as an "information".
(viii) console.table(arg) → prints an object by making table of arguments
(ix) console.time() <some code> console.timeEnd() → shows how much time the code has taken for execution.
(x) console.warn("m") → prints the argument as warning.
→ For user interaction we use → prompt("for input")
→ alert("to show something")
→ confirm("m") → waits till user clicks OK or cancel

Javascript in Browser

Date Page 22

DOM (Document Object Model)

- When a web page is loaded, the browser creates a Document Object Model of the page.
- It has got the power to change the following dynamically:
  - all the HTML elements
  - all the HTML attributes
  - all the CSS styles in the page.
  - remove existing HTML elements and attributes
  - add new HTML elements and attributes.
  - can react to all existing HTML events in the page.
  - can create new HTML events in the page.
- In other words, DOM defines a standard for accessing documents
- The HTML DOM is a standard for how to get, change, add, or delete HTML elements.
- The DOM methods are actions one can perform on HTML elements
- DOM properties are values of HTML elements that one can set or change.
- In DOM, all HTML elements are defined as objects
- A property is a value that one can get or set
- A method is an action that one can do i.e adding or deleting an HTML elements.

Ex →

```
<html>
  <body>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML="Subham";
    </script>
  </body>
</html>
```

// "getElementById" is a method which is performed on object "demo"

through the property "innerHTML" for replacing the content of the HTML elements */

⇒ HTML DOM Document Object

- The HTML DOM document object is the owner of all other objects in the web page. If any element needs to be accessed for performing an actions, always they are accessed through the document object.

- Finding HTML Elements

Methods	used for
document.getElementById("id")	To find a particular element through its id.

```
document.getElementsByClassName("class")[0].innerHTML = "Text";
document.getElementsByTagName("Tag")[0].innerHTML = "Text";
document.getElementsByName("Name").innerHTML = "Text";
```

- Children of an element

For accessing the children of body having the following tags inside them —

```
<body>
  <div> This is paragraph 1 </div>
  <span> This is paragraph 2 </span>
  <p> This is paragraph 3 </p>
</body>
```

Methods	result
document.body.firstChild	→ div
document.body.lastChild	→ p
document.body.childNodes	→ [div, span, p]

/* Ignoring #text which come for indentation */

/* .childNodes[0] === .firstChild (optimized)
.childNodes[.childNodes.length-1] === .lastChild (optimized)
.hasChildNodes() → returns true if the caller has child Nodes */

/* Yeah, childNodes during inspection looks like an array but in actual they are a collection. They can be converted into array by the method Array.from(collection)
∴ Array.from(document.body.childNodes). */

/* let there be the following structure —
 Blue
 Red
 Black
→ when written this inside the script tag, the get —
document.querySelector("div").style.color = "blue" → Blue Red Black
→ & in the following case, this happens —
let a = document.querySelectorAll("div");
for (let i = 0; i < a.length; i++) {
a[i].style.color = "Red";
} */ → Blue Red Black

⇒ Siblings and Parents
Sibling nodes are children of same parents
Ex → <head> and <body> are siblings of parent <html>. <head> is said to be the "previous" of 'left' sibling of <body>.
implementation
<body>

Subham
Das

</body>

→ shows element DOM Tree

/* console.log(document.getElementsByClassName("div")[0]) → Subham
console.dir(document.getElementsByClassName("div")[0]) → ▶div
↳ shows element as as object with its properties
*/

let a = document.body.firstChild // div
console.log(a.firstChild) // div + class → first div.first
console.log(a.firstChild.nextSibling) // → div → class → second. div.second
<!-- In the format our structure is written we won't get the first or second child as expected, because DOM considers indentation, comments etc as node list -->

⇒ Element Only Navigation
- To avoid the indentation and comments which are not valid document elements, when we access the child Nodes or siblings, we need to navigate through elements in the documents only.
- To do it, we can replace, for example first child, which we've been using till now to .firstElementChild
- Considering the previous structure —
  let b = document.body.
  console.log(b.firstChild); // #text // for indentation
  console.log(b.firstElementChild); // div.first
- That's how we can use —
  → .previousElementSibling → .nextElementSibling
  → .lastElementChild

/*
let a = document.querySelector(".first");
a.style.color = "blue";
let b = document.querySelectorAll(".first")[0]; } Both are equall
b.style.color = "blue";
→ QuerySelectorAll takes the tagname as well then we can go for advancement of the indexes [0] as per our requirem
*/

⇒ Inner HTML & Outer HTML and Other DOM properties

InnerHTML property allows to get the HTML inside the element as a string.

Ex → console.log(document.getElementById("~idName~").innerHTML);

↳ will show the inner content of id specified.

OuterHTML property contains the full HTML i.e innerHTML + the element itself.

Ex → Hey

console.log(document.getElementById("div1").innerHTML);

↳ Hey

console.log(document.getElementById("div1").outerHTML);

↳ Hey

div1.innerHTML = "Subham"; → Subham

div1.outerHTML = " Subham " → Subham

/* We can also use .nodeValue or .data for inner content */

/* console.log.textContent is also used to extract the text content of a webpage. */

⇒ Attribute Methods

<body>

Hello and Welcome to Monday Night Law

</body>

-1. hasAttribute ("Attribute name");
Ex → first.hasAttribute ("class");
↳ true // checked if the specified attribute has class.
-2. getAttribute ("Attribute name");
Ex → console.log (first.getAttribute ("class");
↳ firsta // gets or prints the class of log having id first
-3. setAttribute ("Name of the attribute", "value of the attribute");
Ex →
first.setAttribute ("class", "first")
↳ // replaces the attribute value
-4. removeAttribute ("AttributeName");
Ex →
first.removeAttribute ("class");
↳ // removes the specified attribute name.
-5. attributes
Ex → first.attributes
↳ {0: id, 1: class} // gets all the attributes

⇒ Insertion of a Node inside a DOM

<body>
A
</body>

1. document.getElementById("first").innerHTML = "Hello";
2. let a = document.getElementByTagName("div")[0];
let d = document.createElement("div"); // created a div
d.innerHTML = "Hello"; // gave values to div.
a.append(d); →
Hello → A Hello

- a.prepend(d) → Hello A → Hello A (inside same div of id "first")
- a.before(d) → Hello A → Hello A (two different divs)
- a.after(d) → A Hello → A Hello (two different divs)
- a.replaceWith(d) → Hello → "d" replaced a

>

3. <body>
Container

<script>
first.insertAdjacentElement("beforeBegin", "Before Begin");
first.insertAdjacentElement("beforeEnd", "Before End");
first.insertAdjacentElement("afterBegin", "After Begin");
first.insertAdjacentElement("afterEnd", "After End");
</script>
</body>

⇒ Before Begin
After Begin Container Before End
After End

/* In the Display */

<body>
⇒ "Before Begin"

After Begin
Container
Before End

"After End
</body>

/* In the console */

/* first.remove() → removes div with id first */

⇒ Insertion and deletion of classes in DOM

Insertion of class in a dom can be done through —

Ex → <body>

 My Name is Subham

<script>

first.className = "red bg-white";

</script>

</body>

→ My Name is Subham

/* This method can add or can change the class */

/* In CSS, bcd we've defined red to change color to red & change background to white. Similarly, if we have defined — .blue {

color: #00f;

}

then the following script will have changed them to-

<script>

first.className = "blue bg-white";

</script>

→ My Name is Subham

- Addition can also be done by → first.classList.append("-");

*/

Deletion of class from a DOM can be done by —

<script>

first.classList.remove("red");

</script>

→ My name is Subham

/* Removes the existing class from the DOM dynamically */

/* id.classList.toggle("className")
↳ add the class Name if that is not present in the DOM
removes the class name if that is present in the DOM

⇒ Best way to add or remove class is —
id.classlist.add/remove("className");

⇒ To check if the particular element contains the class —
id.classlist.contains("class Name");
*/

Set Interval and Set Timeout

⇒ Set Timeout
- When we need to perform some task after a period of time, we use setTimeout.
Ex → When we need something to happen or something to be displayed on the screen after some user defined time, we can do this —

```
setTimeout(function(){
    alert("Hey, How may I help you");
},2000);
```
↳ After 2000 ms or 2 sec, the alert will be displayed.

- setTimeout returns a Timer ID. we can use it by —

```
let b = setTimeout(function(){
    prompt("Do you want to run setTimeout ":(y/n)");
})
if(b == 'n'){ ↳ if the user chooses No(n),
    clearTimeout(b);
}   ↳ function to clear the given timeout
```

Basic syntax can be —

let b = setTimeout (function(), <Time>, arg1, arg2, ...);

Ex → const sum = (a, b) => {
    console.log(a+b);
}

setTimeout (sum, 5000, 2, 6);

↳ In the console, 8 will be printed.

*Analysis

First of all, a function named sum was defined which work is similar to its name.

Then we defined setTimeout. Inside it we called the above defined function, as the second argument, we provided the time in ms. As the 3rd and 4th argument, we've provided actual arguments to the function sum and as a result it sums up and prints the sum of its formal argument.

⇒ Set Interval

- When we need to perform a task repeatedly after a specified time duration then we use setInterval.

Ex → When we need to display the time & date repeatedly we do like this —

showTime=() => {
    console.log(Date());
}

let b = setInterval(showTime, 3000);

↳ It will print the date & time every 3 seconds.

- Similarly, like setTimeout, to clear the Interval we use —

clearInterval(b); → stops the printing of date every 3 sec.

Events

Events are "things" that happen to HTML elements. When javascript is used in HTML pages, Javascript can "react" on these events.

Generally, the event can be something the browser does, or something a user does. So often when events happen, to do something we need to perform some practises.

onclick

When something is clicked on the HTML DOM, we can record that. for example —

→ `<button onclick = "alert ("The button is clicked")> click </button>`

When button is clicked an alert will popped up.

→ `<div class = "first"> has something to do </div>`

```
<script>
    let b = document.getElementByClassName("first")[0];
    b.onclick = () => {
        b.innerHTML = "Hello";
    }
```

It will change "has something to do" to "Hello" when clicked.

Some commonly used Events are —

ondrag, onclick, dblclick, oninput, onmouseover, onmouseenter, onmouseleave, onmousemove, onmouseout, onmousewheel, onselect, onsubmit, etc.

Javascript Advanced

Callbacks

- A callback is a function passed as an argument to another function. This technique allows a function to call another function. A callback function can run after another function has finished.
- In a nutshell, a callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete an action.

```
Ex→ const callback = (a, b, operation) => { // defined the callback function
        return operation(a, b); // called the function which was passed to it.
    }

    const add = (num1, num2) => { // defined the add function
        return num1 + num2;
    }
```

Or we could've done `const add = (num1, num2) => num1 + num2`

```
const callback = (4, 3, add); // passed arguments with add function.
console.log(callback); // 7

const sub = (num1, num2) => num1 - num2; // defined callback
Const resSub = callback(4, 3, sub) // called callback
console.log(resSub); // 1
```

- We can also use callback for our basic needs -

```
Const a = [4, 1, 6, -2, -5, 3, 2, -8, 6, 7]
```

Ex: For finding the 1st negative value in the array -

```
const findNeg = (a) => { if (a < 0) return a; }
const res = a.find(findNeg); const resu = a.findIndex(findNeg);
console.log(res); // -2
console.log(resu); // 3
```

⇒ Advantages of Callback

- Asynchronous Control

Callbacks are essential for handling asynchronous operations which means a code can run while other tasks are happening in the background.

- Modularity

Callbacks can be used to create modular code that is easier to understand and maintain. This also makes it easier to reuse code and to keep your code organized.

- Flexibility

Callbacks are flexible and can be used in a variety of ways. For ex, one can use callbacks to pass data between functions, to control the order in which functions are executed or to handle errors.

⇒ Disadvantages of Callback

- Callback Hell

When callbacks are used extensively the code can become difficult to read and debug. This is because callbacks can be nested deeply making it difficult to follow the flow of code.

- Error Handling

It can be difficult to handle errors in a callback-based code. This is because error can occur anywhere in the callback chain and so it can become horrible to track down the source of the error.

- Testing

Callback based code can be difficult to test. This is because it can be difficult to mock our callbacks in unit tests.

Overally, Callbacks are a powerful tool that can be used to write more efficient, modular and flexible code. However, it is important to use callbacks carefully and to be aware of the potential drawbacks.

⇒ Tips to use callback effectively
- Avoiding nesting callbacks deeply.
- Using descriptive function names for callbacks for readability
- Using a try/catch block for handling errors
- Writing tests for callbacks to identify and fix errors before they deploy to production.

Promises :
- Too much of callback usage can make the programmer exhausted which trying to catch error or debuging. It can sometimes gives the programmer tough time through its most popular problem i.e. "callback hell" or "pyramid of doom" which happen while nesting many callbacks.
- The most effective way to tackle this problem is using "promises" instead.
- A promise is a "promise of code execution". The code either executes or fails, in both cases the programmer will get notified.
- The syntax of a Promise look like this :-

```
let promise = new promise (function (resolve, reject){
        // codes
});
```

Notes:-

- resolve and reject are two callbacks provided by javascript itself. They are called like this:-
  resolve(value):→ If the job is finished successfully.
  reject(value):→ If the job fails
- The promise is an object and it returned by the new promise constructor which has the following attributes:-
  ☆ state :- Initially "pending", then changes to either "fulfilled" whenever it is resolved and resolve is called or "rejected" when the code execution fails and reject is called.
  ☆ result :- Initially "undefined", then changes to value which is given in the code or "error" when rejected.

/* resolve(value) && reject(error) */

For Ex →

```
1. const promm = new promise ((resolve,reject) => {
2.     const don = true;
3.     if (don) resolve ("It's working");
4.     else reject ("It has failed");
   })
promm
5. .then((data) => console.log(data))
6. .catch((data) => console.log(data))
7. .finally(() => console.log("It will execute everytime"))
```

/*

1 → new promise was created by invoking the constructor "promise".

2→ taken a variable "dom" and made it "true"
3→ if "dom" is "true", resolve will get executed
4→ if "dom" is false or not executed, "reject" will be executed
5→ .then will get executed only when inside the promise block, the codes had run successfully and resolve has executed eventually which means the code has generated "no error".
6→ .catch will be executed iff the code inside the promise block has not been executed successfully or throws error. .catch is useful in catching the error and instead of showing the error in the console we can get some messages inside the .catch block.
7→ .finally will run everytime the promise is executed ~~whether~~ its successful or not.

Notes :-
- .then() takes a callback function that receives whatever inside the resolve as parameter or argument. It is same to .catch() also.
- If we had changed dom = false, the output would have been:-
  It has failed // for we has this msg inside reject which will passed to .catch as an arg i.e. data and will be eventually printed.
  It will execute everytime
- Basically through .then(value) and .catch(value), the consuming code receive the final result of a promise.

⇒ Promise Chaining

- We can chain promises and make them pass the resolved values to one another.
- It is the most significant way to tackle callback hell.
- Promise chaining can be done like this —

```
P:- let p1 = new Promise ((resolve, reject) => {
        setTimeout (() => {
            console.log ("Resolved after 2 seconds")
            resolve (22)
        }, 2000)
    }).then ((value) => {
        console.log (value)
        let p2 = new Promise ((resolve, reject) => {
            resolve ("Promise 2")
        })
        return p2
    }).then ((value) => {
        console.log ("Now we are done")
        return 2
    }).then ((value) => {
        console.log ("Now only we are done")
    })
```

* Analysis

- In P1 we assigned a promise which has a timeout of 2sec. It will also print "Resolved after 2 seconds" with returning the value "22" which is inside the "resolve()".
- ".then()" will catch the "value" returned by "P1" i.e "22" and prints the value. It will also create a "new promise" inside the ".then()" which will return the value "Promise2"

which is inside "resolve" as "P2". The ".then()" beside it will catch the value to "Promise 2" and as it is "true", the ".then()" value will print "We are done" and return the value "2" which will eventually caught by the ".then()" behind it and it will print "We are done Now only".

⇒ Promise chaining for Callback Hell

- Promise chaining is actually designed in javascript to handle asynchronous operations in a more structured and readable way compared to traditional callback functions.
- Promise chaining can help in the following ways :-

1. Sequential Execution :- With promise chaining, one can perform asynchronous operations sequentially. This makes the code more readable because the order of execution is clear.
2. Error Handling :- Promises allow a programmer to attach ".catch()" handler at the end of the chain to handle errors in a centralized manner. This is cleaner than handling errors within each callback.
3. Avoiding Pyramid of Doom :- Promise chaining avoids the nesting of callbacks, commonly reffered to as the "PD", where multiple asynchronous operations are indented deeply within one another. This result is cleaner and more maintainable.

⇒ Attaching Multiple Handlers

We can attach multiple handlers to one promise. They don't pass the result to each other ; instead they process it independently.

let p is a promise

p.then (handler 1)
p.then (handler 2)
p.then (handler 3)

Runs independently

* Notes

- Attaching multiple handlers is different from promise chaining. Promise chaining says to attach ".then()" one by one which make the code a single piece but when we attach multiple handlers we do it separately.
- In case of attaching multiple handlers, each handler function will be called in the order that it was attached, regardless of whether the previous handler function resolved or rejected the promise. It
- While in promise chaining, each handler returns a new promise, and the next handler function in the chain is called with the resolved value of the previous.
- Ex →

Multiple handlers

```
const promise = new Promise(resolve, reject) => {
        resolve('success');
});
promise.then((result) => {
        console.log(result); // will print <success>
});
promise.then(() => { // Even though nothing is returned by the prev .then(), this function will be executed also will print <promise resolved> as both then .then() are separate entity so no chaining done.
        console.log('promise resolved');
});
```

Promise Chaining

```
const promise = new Promise((resolve, reject) => {
        resolve("success");
});
promise.then((result) => { // took the parameter of resolve as result
        console.log(result); // success
        return result + "!";
```

```
promise.then((result) => {
        console.log(result); // success!
});
```

-Main difference

Promise chaining allows to pass the solved value of one handler function to the next handler function in the chain. This can be useful for performing multiple async. opr. in sequence, or for processing result of an async. opr. in diff ways.

⇒ When to use what

-Attaching multiple handlers to a promise is a good approach when we need to perform multiple independent operations on the resolved value of the promise.

-Promise chaining is a good approach when we need to perform multiple asynchronous operations in sequence, or when we need to process the result of an asynchronous operation in different ways.

The Promise API

-It is a mechanism for handling asynchronous operations. Promises are objects that represent the eventual completion or failure of an asynchronous operation and allow you to work with them in a more structured and readable way compared to traditional callback-based approaches.

-There are 6 static methods of Promise class:-

```
let p1 = new Promise((resolve, reject) => {
        setTimeout(() => {
                resolve(22);
        }, 2000);
});
```

```
let p2 = new Promise ((resolve, reject) => {
        setTimeout (() => {
                resolve (44);
        }, 4000);
});
```

ii) Promise.all ([Promises]) :→

- waits for all given promises to resolve and returns the array of their results.
- If any one fails, it becomes error and all other results, may they have resolve, would be resolved.

```
Ex→ let p = Promise.all ([p1, p2])
    p.then (value) => {
            console.log ("All resolved")
    })
```

//When both the p1 & p2 are resolved, p will show the msg
//If we have → (console.log(value)) it end have printed → <[22, 44]>

ii) Promise.allSettled ([promises]) :→

- What if one of our two promises got failed or get some error, then the Promise.all() wont work. That's the time when one needs Promise.allSettled().
- It would provide the status of every promise as object

```
Ex→ let p = Promise.allSettled ([p1, p2])
    p.then (value) => {
            console.log (value);
    });
```

//if all resolved :-
{status: 'fulfilled', value: 22},
{status: 'fulfilled', value: 44}

//We wont be getting error like 'Promise.all' here.

//if p2 has rejected
{status: 'fulfilled', value: 22};
{status: 'rejected', reason: 44};

iii) Promise.race([promises]):→
- Waits for the first promise to be settled and its result/error becomes the outcome.

iv) Promise.any([promises]):→
- Waits for the first promise to be fulfilled (not rejected) and its result becomes the outcome.
- It is similar to Promise.race() but only prints the value of the promise that is resolved first.
- If all the promises failed, it throws an aggregate error.

v) Promise.resolve(value):→
Makes a resolved promise with given value.

vi) Promise.reject(error):→
Makes a rejected promise with given error.

⇒ Features of Promise API
- Promise API should be used while working with operation that don't block the main thread (eg., network requests, file I/O, timers) to ensure the code remains responsive and doesn't freeze the user interface.
- Promises help to manage the flow of asynchronous code and handle errors more elegantly, making the code more maintainable and robust.
- Javascript has also introduced Async/Await, which is built on top of Promise and offers an even more concise way to work with asynchronous codes.

Async/Await

- It is a feature in javascript that was introduced to simplify working with asynchronous code, particularly when dealing with Promises.
- It provides a more concise and readable way to write asynchronous code compared to using callbacks or chaining ".then()" methods.
- A function can be made async by using async keyword
- We have solved the problem of callback hell by going through Promises but still the problem arises while attaching multiple handlers or using .then() or .catch() chaining. It sometimes make the code look ugly and thus decreases readability.
- Async/Await was introduced to solve the above problem
- We can make any function async and make any promise form to await inside it

⇒ Async

- Async will always return a promise. If in case it is returning some values (i.e. strings, numbers etc), they will wrap inside a promise automatically & is returned

```
Eg→ async function getData(){
        return "Subham";
    }

const data = getData()   ⇒ Promise {<fulfilled>: 'Subham'}
console.log(data);
data.then((res)=>{
    console.log(res);    ⇒ Subham
});
```

⇒ Async with await

- Async with await is used to handle Promises.
- Before async/await, for handling promises we used .then() and .catch() methods.

Ex:-

```
const p = new Promise ((resolve, reject) => {
    resolve ("It is resolved");
})
function getResolved () {
    p.then ((result) => {
        console.log (result);  ⇒ It is resolved
    });
}; getResolved (); // function called
```

Before

```
async function getResolved () {
const val = await p;
console.log (val);  ⇒ It is resolved
}
getResolved (); // function is called
```

Notes :-

* await is a "keyword" that can only be used inside an async function
* While using .then() inside the function for a particular promise —

```
const p = new Promise ((resolve, reject) => {
    setTimeout ( () => {
        resolve ("It is resolved");
        console.log ("Print it after some time")
    }, 10000);
})
```

```
function getResolved(){
    p.then((res)=>console.log(res));
    console.log("Inside console.log under .then()")
}
getResolved();
```

↳ As we know, time fide and javascript waits for none. After calling the function getResolved() the output will be —

Inside console.log under .then()
Print it after sometime. (after 10 seconds)
It is resolved

* For this feature of javascript, developers get confused often because in the promise, we wrote .then() first and console.log second but while printing, it got reversed.

* While using async/await —

```
async function getResolved(){
    const val = await p;
    console.log(val);
    console.log("Promise is resolved")
}
getResolved();
```

Output (after 10 seconds)

Print it after sometime.
It is resolved
Promise is resolved

* If there would've been a console.log in promise under setTimeout

.then()	async/await
Inside promise, outside Timeout Inside console.log under .then() after 10 seconds.... Print it after sometime It is resolved	Inside Promise, outside Timeout after 10 seconds..... Print it after sometime It is resolved Promise is resolved

* If we had like the below for both the promises p1 & p2:

```
const p1 = new Promise ((resolve, reject) => {
    setTimeout (() => {
        resolve ("The promise 1 is resolved");
    }, 10000);
});
const p2 = new Promise ((resolve, reject) => {
    setTimeout (() => {
        resolve ("The promise 2 is resolved");
    }, 5000);
});

async function getResolved () {
    console.log ("Hello async/await");
    const val1 = await p1;
    console.log (val1);

    const val2 = await p2;
    console.log (val2);
}

getResolved (); // called the function
```

Output

Hello async/await
waited for 10 seconds...
The promise 1 is resolved // should take 10 seconds
The promise 2 is resolved // should take 5 seconds.

// In the above case after 10 seconds both p1 & p2 got promised as p2 got resolved in 5 seconds and waited as p1 was called first and et took 10 seconds. So after 10 seconds both of them resolved together.

⇒ Error Handling

Sometimes the script can have errors. To handle them we have try { } and catch () { } blocks.

The try... catch syntax allows us to catch errors so that script instead of dying can do something more reasonable.

Syntax

It has two main blocks —

```
try {
    codes for execution
}
catch (error) {
    if some error occured, this block of code will be
    executed instead of making the program halt.
}
```

Notes

* For not making the program halt unreasonably, try... catch is used
* We should write our code which we think might throw some unexpected errors. and write the code inside catch block saying if error what to do next.
* If the try block containing code executed successfully, then catch will be ignored.
* try... catch works synchronously. Also it handles the codes' errors which are synchronous in nature.

```
Ex → try {
        setTimeout(() => {
            console.log(ss-190$); // some error
        }, 2000);
    }
catch (error) {
    console.log("It got error"); // won't work as in try there is a async code
}
```

⇒ The Error Object

- For all built in errors, the error object has two main properties

```
Ex: try {
      console.log(hello);
    }
    catch(error){
        console.log(error.name); // Reference Error
        console.log(error.message); // hello is not defined
```

- We can also throw custom errors —

```
Ex: try {
        throw new ReferenceError("hello is not defined");
    }                    // we could have also used SyntaxError
    catch(error){
        console.log(error.message); //hello is not defined
        console.log(error.name); // Reference Error
```

⇒ The finally clause

- The try ... catch may throw error or if the error occurred inside the try then catch will handle it but if catch throws an error, then the program will halt.
- To overcome it, the try ... catch construct has one more clause : finally

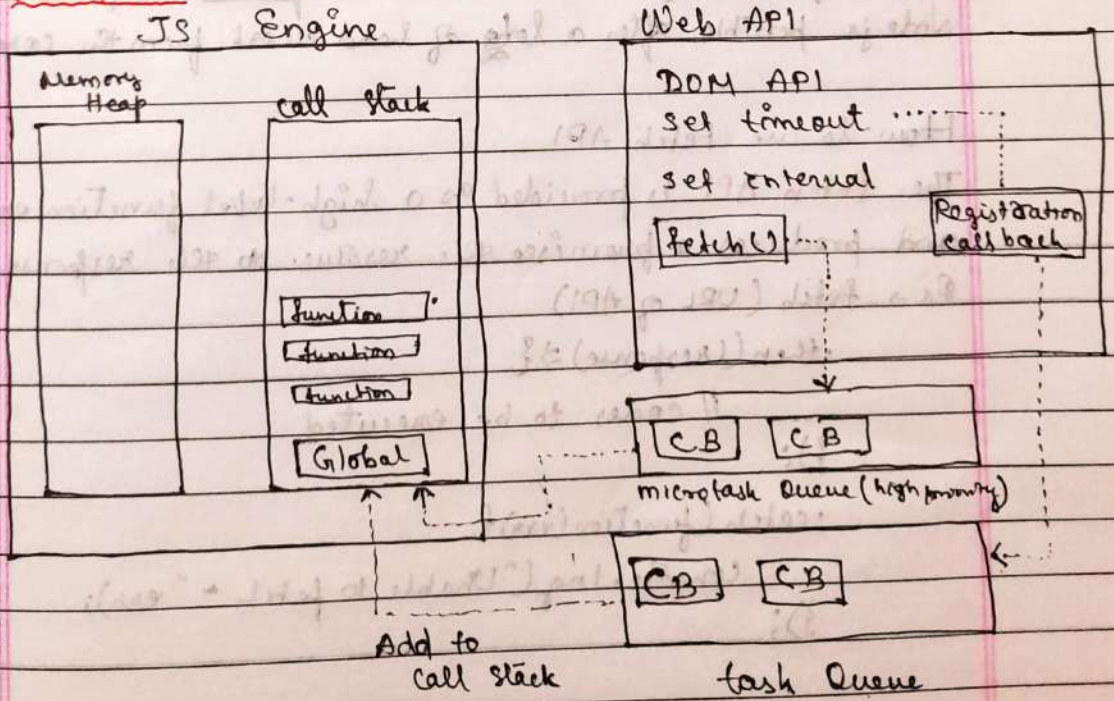
```
Ex →
    try {
        console.log(subham); // Reference Error which is
    }                          handled by catch.
    Catch(error){
        console.log(das); // Reference Error, not handled by
    }                       anything
```

```
finally {
    console.log("The execution is over");
} // having error in catch block which is not handled by any other block, then finally will still run.
```

*Notes:-

* If the program exits, the finally block will still run.
* After try if there were no error or after catch if there were errors, the finally clause will be executed.
* If there is a return in try to pass the control over to next block of the code, then also the finally will be executed.

Fetch API



Event Loop

- In early days, it was difficult to perform asynchronous requests across websites; developers had to use clumpsy approaches to interact across multiple networks.
- Internet Explorer changed this in 1988 with the introduction of XMLHttpRequest, an API meant to overcome this limitation. It was initially designed to fetch XML data via HTTP, but as the web grew, it became so difficult to work with raw Javascript frameworks.
- In 2015, the Fetch API was launched as a modern successor to XMLHttpRequest and was the de facto standard for making asynchronous calls in web apps.
- One significant advantage Fetch has over XMLHttpRequest is that it leverages Promises, allowing for a simpler and cleaner API while avoiding callback hell.
- In 2018, Undici was introduced as a newer and faster HTTP/1.1 client for Node.js. Undici made the fetch() implementation in Node.js possible after a long of hard work from the core team.

How to use Fetch API

The fetch API is provided as a high-level function on a URL and produces a promises the resolve to the response:

```
fetch (URL of API)
    .then (response) => {
        // codes to be executed
    });
    .catch (function (err) {
        console.log ("Unable to fetch -" err);
    });
```

*Notes

- The fetch() method starts the process of fetching a resource from the network, returning a promise which is fulfilled once the response is available.
- The promise resolves to the Response object representing the response to your request.
- A fetch() promise only rejects when a network error is encountered which is usually when there's a submission issue or similar. A fetch() promise doesn't reject on HTTP errors i.e. — a then() handler must check the response.ok and/or response.status properties.

Objects : in depth

Objects can be declared or defined in two ways.
(i) Through literals ⇒ const obj = {key : value};
(ii) Through Constructors ⇒ Object.create

Making Objects Through literals

Ex → const obj = {
name : "Subham",
Age : 24
};

/*
We can use arrays, functions or methods, Symbols and also another object as the value of the object
*/

⇒ How to access Objects ??
console.log(obj); // will show all keys & values
console.log(obj.name); // Subham
console.log(obj["name"]); // Subham /* "name" bcoz the value of the key can holds a String value */

/*
Best eeay to access object values through key is console.log(obj["name"]) because when we have a key like → "full name" : "Subham Prasad Das", we can not access the full name as obj.fullname but we can do it through console.log(obj["full name"])
*/

/* To access keys ⇒ console.log(Object.keys(obj)); ['Name', 'Age']
To access values ⇒ console.log(Object.values(obj)); ['Subham', 24]
⇒ console.log(Object.entries(obj)); // [['Name' : 'Subham'], ['Age' : '24']] → 2D array
*/

⇒ How to use Symbol as key & value &

```
const mySym = Symbol("key1");
```

When we do like

```
const obj = {
    name : "Subham",
    Age : 24,
    mySym : "Symbol"
};
```

It doesn't show error but is not the right way to use symbol in the object, because when we let know its type, it comes as String

```
console.log(typeof obj.mySym); // String
```

To use a Symbol, we can do this —

```
const obj = {
    name : "Subham",
    Age : 24
    [mySym] : "SymbolValue"
}
```

⇒ How to overwrite/modify/change the key values &

```
obj.name = "Shibam";
console.log(obj["name"]); // "Shibam"
```

⇒ How to make an object unchangable/unmodificable &

```
Object.freeze(obj);
obj.name = "Shibam"; ..... // tried to overwrite
console.log(obj["name"]); // Subham // Its unchanged.
```

⇒ How to make object Singleton / through Constructor?

```
const obj = new Object(); // Singleton Object
const obj = {}            // Non-singleton Object
/* Both the above declared objects are empty objects */
```

⇒ How to add keys & values into the above declared singleton object?

```
obj.id = 12345
obj.name = "Subham"
console.log(obj); // { id: 123, name: 'Subham' }
```

⇒ How to merge two Objects?

```
const obj1 = { 1: "a", 2: "b" }
const obj2 = { 3: "c", 4: "d" }
const obj3 = { obj1, obj2 }
console.log(obj3); // { obj1: {1: "a", 2: "b"}, obj2: {3: "c", 4: "d"} }
```

It would be an incorrect syntax to merge 2 objects as here the objects are not merged but they are forced to come into same entity.

```
const obj3 = Object.assign(obj1, obj2)
console.log(obj3); // {1: 'a', 2: 'b', 3: 'c', 4: 'd'} /* Successfully Merged */
```

It also can be written as —

⇒ `const obj3 = Object.assign({}, obj1, obj2);`

`{}` → target, `obj1, obj2` → sources

⇒ Syntax : `const newTarget = Object.assign(target, sources);`

⇒ Another Syntax can be —

```
const obj3 = {...obj1, ...obj2};
console.log(obj3); // {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

⇒ Object De-structuring.

```
const course = {
    courseName : "Javascript",
    price : "999"
}
```

For accessing the values we write obj.keys. But when we need to use the values many times we can destructure the objects keys in our own way.

⇒ Syntax : const {destructured-word} = Object-name ;
console.log (destructured-word) // value of the key destructured

Ex → const {id} = obj
console.log (id) // 12345

⇒ Another syntax when the key is too long, for making it user readable —

Syntax : const {key : key-to-be-used} = Object-name ;

Ex → when we want to use "name" as "n" —

```
const {name : n} = obj ;
console.log (n) ; // Subham
```

"this" keyword

'this' keyword refers to an object that is executing the current piece of code. It references the object that is executing the current function.

Ex → let obj = {
    Name : "Subham",
    Age : 24,
    Greetings : function () {
        console.log (`${this.Name}, Welcome`);
    }
}

obj.Greetings() ; // Subham, Welcome
obj.Name = "Shubam"; // Overwritten
obj.Greetings(); // Shibam, Welcome

* this generally prints the current content
* when written → console.log (this) inside the greeting function it would have printed the whole object
* when written → console.log (this) outside the object, it would print empty parenthesis → {}
  because we are in node environment, so globally printing 'this' will be empty.
* when written → console.log (this) globally in browser, it would print window objects. Because Window is global in case of browser.
* when printed → console.log (this) inside a function we get many random values inside an object.

Ex → function print () {
    let Name = "Subham";
    console.log (this.Name); // undefined → don't work with functions unlike objects
}

* when written → console.log(this); inside an arrow function, it will give empty parenthesis → {}

⇒ WAYS TO CREATE OBJECTS

⇒ const person = {name: 'Subham', age: 30}; // simple way

⇒ function Person(name, age) {
    this.name = name; this.age = age;
}
const john = new Person('John', 24);

// using constructor
used when want to create object of similar keys but different values.

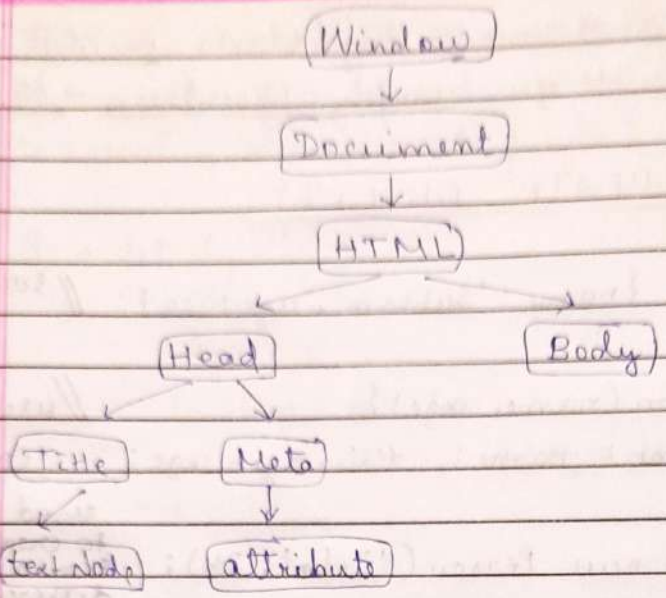
⇒ const personProto = {sayHello: function() {console.log('Hello');}};
const John = Object.create(personProto);
john.name = 'John';
john.age = 24;

⇒ class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
const john = new Person('John', 24);

// Modern way to create object. Follows the pattern of creating object through constructor, just wraps all inside a class, introduced in ES6.

⇒ function createPerson(name, age) { return {name, age}; }
const john = createPerson('John', 24);
// used when one wants to encapsulate the creation of an object within a function.

DOM & EVENTS : in depth




```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta ...>
    <title> DOC </title>
  </head>
  <body>
    <h1 id="para" class="para1"> Javascript </h1>

  </body>
</html>
```

Document

```
const d = document.getElementsById('para');
d.innerHTML; // Javascript
d.innerText; // Javascript
d.textContent; // Javascript
```

* The difference between innerText and textContent is — let's suppose, there exist some text in a span with display: none; ie, hidden.
innerText won't show the hidden text but textContent shows everything. It can be visible ones or hidden ones.
* Lets suppose there is a h1 with id para and also has a span with some text inside it —→

```
<h1 id="para"> Javascript <span hidden> Hidden </span> </h1>

const d = document.getElementById('para');
d.innerHTML; // 'Javascript <span hidden> Hidden </span>'
d.innerText; // Javascript
d.textContent // Javasiript Hidden.
```

⇒ querySelector
* for getting contents of a particular id —→
`document.querySelector('#id-name').innerHTML;`
* for getting contents of a particular class —→
`document.querySelector('.para1').innerHTML;`
* for getting the ~~for~~ h1 —→
`document.querySelector('h1').innerHTML;`
* querySelector only selects the first child.

⇒ querySelectorAll
Let's suppose there are 4 headings (h1) & we need to color the 2nd heading to green.

```
⇒ const d = document.querySelectorAll('h1')[1]; // Take it as array
d.style.color = "green";
⇒ const d = document.querySelectorAll('h1');
d[1].style.color = "green";
```

Let's suppose we want to color or do any operation on every Nodelist of the document where the document has 5 headings (h1).

```
⇒ const d = document.querySelectorAll('h1');
d.forEach((h1) => {
    h1.style.color = "green";
});
```

* We can convert the given Nodelist or HTMLCollection to Array by the method, Array.from(Nodelist or HTMLCollection)

⇒ Relationships

Sample to be taken into consideration

```
<body>
  <div class = "parent">
    <div class = "day"> Monday </div>
    <div class = "day"> Tuesday </div>
    <div class = "day"> Wednesday </div>
    <div class = "day"> Thursday </div>
  </div>
</body>
```

```
⇒ const parent = document.querySelector('.parent');
console.log(parent);
  ⇒ <div class = "parent"> ... </div>
⇒ Console.log(parent.children)
  ⇒ HTMLCollection(4) [div.day, div.day, div.day, div.day]
⇒ Console.log(parent.children[1]);
  ⇒ <div class = "day"> Wednesday </div>
```

⇒ console.log(parent.children[2].innerHTML);
⇒ Tuesday.
⇒ for (let i = 0; i < parent.children.length; i++) {
    console.log(parent.children[i].innerHTML);
}
⇒ Monday
Tuesday
Wednesday
Thursday
⇒ parent.firstElementChild.innerHTML // Monday
⇒ parent.lastElementChild.innerHTML // Thursday
⇒ parent.children.length // 4
⇒ const dayOne = document.querySelector('.day');
dayOne.parentElement // ...
⇒ dayOne.nextElementSibling.innerHTML // Tuesday
⇒ parent.childNodes // NodeList(9) [text, div.day, text, div.day, text, div.day, text, div.day, text]

⇒ Creating An Element

⇒ const div = document.createElement('div');
div.setAttribute('class', 'div'); // div.className = 'div';
div.setAttribute('id', 'div1'); // div.id = 'div1';
div.style.color = 'red';
div.innerText = 'Subham';

document.body.append(div); // document.body.appendChild(div)
⇒ const elem = (lines, color, bgco) => {
    const para = document.createElement('div');
    para.innerText = lines;
    para.style.color = color;
    para.style.backgroundColor = bgco;
    document.body.appendChild('para');
}

elem('Subham Das', 'blue', 'red');

↳ Subham Das

⇒
```
<body>
  <ul class="languages">
    <li> C </li>
  </ul>
</body>
<script>
  const addList = (lang) => {
    const listName = document.createElement('li')
    listName.innerText = lang;
    document.querySelector('.languages').appendChild(listName)
  }
  addList('C++');
  addList('Javascript');
```

⇒ • C
• C++
• Javascript

⇒ Editing the document

```
⇒ const newLang = document.querySelectorAll('li')[0];
  newLang.innerText = "Python";
```

⇒ • Python
• C++
• Javascript

```
⇒ const newLang = document.querySelector('li:nth-child(1)');
  document.createElement('li') = const newli;
  newli.textContent = "Python";
  newLang.replaceWith(newli);
```

⇒ • Python
• Python
• Javascript

```
⇒ const firstLang = document.querySelector('li:first-child');
  firstLang.outerHTML = '<li> C++ </li>';
```

⇒ • C++
• Python
• Javascript

⇒ Removing the Document

```
const lastLang = document.querySelector('li:last-child');
lastLang.remove();
```

⇒ Python

⇒ EVENTS and EVENT-LISTENERS

Events are actions or occurences that happen in the browser. Javascript allows to respond to these events by attaching functions, known as event handlers, to specific elements or the document itself.

⇒ `<button id="button" onclick="alert('clicked')">Button</button>`

⇒ `<button id="button">Button</button>`

```
<script>
    document.getElementById('button').onclick =
        function() {
            alert('The button is clicked');
        }
</script>
```

⇒ `<button id='button'>Click</button>`

```
<script>
    document.getElementById('button').
        addEventListener('click', function() {
            alert('Button is clicked')
        });
</script>
```

OOP in Javascript

Javascript does have classes. This feature was introduced with the ECMAScript 2015 specification (often reffered as ES6). However, it's important to note that Javascript is primarily a prototype-based language, and its classes are syntactic sugar over existing prototype-based inheritance mechanisms. In other words, it provides a more familiar syntax for developers coming from class-based languages such as java and C++, but under the hood, it works somewhat differently.

Example :-

```
class Person {
    constructor (name, age) {
        this.name = name;
        this.age = age;
    }
    sayHello() {
        console.log(`Hello, my name is ${this.name} and I'm
                    still ${this.age} years old`);
    }
}

let person1 = new Person('Subham', 24);
person1.sayHello(); // Hello my name is Subham and I'm
                    still 24 years old.
```

⇒ The 'new' keyword / Constructors

Ex → const promiseOne = new Promise();
const date = new Date();

The 'new' here is a constructor function. It allows us to make multiple instances from a single object literal. It is used for making new contents.

⇒ Significance of the new keyword.

Let there be a function →

```
function User(username, name){
    this.username = username;
    this.name = name;
    return this; // returning is not necessary, it will
                 return implicitely.
}

const fun1 = User(1122, 'Subham');
console.log(fun1.name); // Subham
const fun2 = User(1133, 'Das');
console.log(fun1.name); // Das
```

Here when we made an other variable and passed two different values, still the "name" of first variable got changed. This can be disgusting when we ~~were~~ a no. of developers are using the same function, they will end up overwritting the entire stuffs again and again. To get over from it, we can use "new" keyword.

```
const fun1 = new User(1122, 'Subham');
console.log(fun1.name); // subham
const fun2 = new User(1133, 'Das');
console.log(fun1.name); // Subham
```

/*
→ whenever we use 'new' keyword, an ~~new~~ empty object is being created which is called as instance.
→ For the new keyword, a constructor function is being called.
→ All the arguments (we want to pass) is injected into the 'this' keyword.
→ Then the 'this' is returned implicitely.
*/

⇒ Behind the scenes of 'new' keyword
→ A new object is created : The new keyword initiates the creation of a new Javascript object.
→ A prototype is linked : The newly created object gets linked to the prototype property of the constructor function. This means that it has access to properties and methods defined on the constructor's prototype.
→ The constructor is called : The constructor function is called with the specified arguments and this is bound to the newly created object. If no explicit return value is specified from the constructor, Javascript assumes this, the newly created object, to be the intended return value.
→ The new object is returned : After the constructor function has been called, if it doesn't return a non-primitive value (object, array, function etc), the newly created object is returned.

Call : in depth

It calls a method of an object, substituting another object for the current object.

- Suppose, there is a function which sets the username

```
function setUserName(username){
    this.username = username;
}
```

- Now we have a function to make object through 'new' keyword where the username is set through calling setUserName() ~~function()~~

```
function User(username, name, password){
    setUserName(username);
    this.name = name;
    this.password = password;
}
const firstUser = User(123, 'Subham', 4567);
```

- But when we pass all those three parameters, we can see the name & password is set but the setUserName() is not called so username is not passed to it so the username is never set.

- Here to call the function outside the function which creates object ("User" in this case), we can substitute 'setUserName(username)' to 'setUserName.call(this, username);' (setUserName)

- When we call setUserName() inside User(), the execution was successful but after the successful execution, the reference of setUserName() get poped out from the call stack. so to make this reference stay with User(), User() give the its own 'this' to setUserName() as after execution the reference stay with User().

How Can We Make Our Own Constants ??

- Constants are those which can not change ⇒
  Ex. The value of Math.PI = 3.141592653589793

- When we access its property, we can see. ⇒
console.log(Object.getOwnPropertyDescriptor(Math, 'PI'));
we get ⇒
{
    value : 3.141592653589793
    writable: false,
    enumerable: false,
    configurable: false.
  }
It is so hard-coded that, it can never be overwritte

- For making our own constant, we can do this —
# const owner = {
      name = 'Subham'
  }
  Object.defineProperty(owner, 'name', {
      writable: false,
      enumerable: false,
      configurable: false
  });

- Now when we change the name & found —
  owner.name = 'Shibam';
  console.log(owner.name); // 'Subham' // didnot change.

Lexical Scoping & Closure

⇒ Lexical scoping defines the scope of a function.

Example :→

```
{
  const var1 = "Subham";

  const fun1 = () => {
      const var2 = "Prasad";
      const fun2 = () => {
          const var3 = "Das";
          console.log(var1); //accessible
          console.log(var2); //accessible
          console.log(var3); //accessible
      }
      fun2();
      console.log(var1); //accessible
      console.log(var2); //accessible
      console.log(var3); //not accessible
  }
  fun1();
  console.log(var1); //accessible
  console.log(var2); //not accessible
  console.log(var3); //not accessible
}
console.log(var1); //not accessible
console.log(var2); //not accessible
console.log(var3); //not accessible.
```

Closure

When we return a function, we actually do not return the function but we return its lexical scope. This procedure is called as "Closure".

Example :-

```
const fun1 = () => {
    const name = "Subham";
    console.log("Inside the fun1");
    const fun2 = () => {
        console.log(name);
        console.log("Inside the fun2");
    }
    console.log("Outside the fun2");
    return fun2;
}
console.log("Outside the fun1");
const funHolder = fun1();
funHolder();
```

Output

```
Outside the fun1
Inside the fun1
Outside the fun2
Subham
Inside the fun2
```

⇒ Who called fun2()

→ When we return the reference of fun2, when the fun1() is invoked, coming to that returning statement the fun2() is invoked automatically.

⇒ How are we returning the scope of the whole fun1 and fun2 ?

→ Because when we are returning fun2(), we are actually returning the lexical scope of fun2. As per lexical scoping, fun2() can access both global as well as fun1(), so returning the lexical scope of fun2 will be advantageous to access the whole function.

Destructuring

- In Javascript, destructuring assignment is a powerful feature that allows to extract values from arrays or object and assign them to variables in a more consise and expressive ways.
- It makes code cleaner and more readable.
- There are two main types of destructurings -

⇒ Destructurings in arrays :-

⇒
```
const array = [1, 2, 3, 4, 5];
const first = arr[0]          // the old and traditional way to
const second = arr[0]         // extract the elements of array into variables
```

⇒
```
const array = [1, 2, 3, 4, 5];
const [first, second, third, fourth, fifth] = array
console.log(first); // 1
console.log(second); // 2
```

⇒ For extracting first two elements and putting rest all -
```
const [first, second, ...rest] = array;
console.log(first); // 1
console.log(second); // 2
console.log(rest); // [3, 4, 5]
```

⇒ For skipping all other except the third which we want as as variable -
```
const [, , third, ,] = array;
console.log(third); // 3
```

⇒ Destructuring Objects :-

Like arrays Objects are also destructured.

⇒ const person = { name: "John", Age : 10 };
const {name, age} = person;
console.log(name) ; // John
console.log(age) ; // 10

// Here, in this process, it is mandatory to use the keys name & age if we use n & a it won't work.

⇒ We can also assign different variable names to the values of every keys of the Object.

const person = {name : "John", Age : 10};
const {name : personName , age : personAge} = person;
console.log(personName); // John
console.log(personAge) ; // 10.

⇒ Destructuring functions :-

const person = { name : 'Subham', age : 24 }; // the Object.
const print = (persona) => {
    console.log(`${persona.name} is good`); // Subham
}
print(person);

We can print the same with destructuring, without the use of dot operator. The function definition will be like —

const print = ({name, age}) => {
    console.log(`${name} is ${age} years old`);
} // Subham is 24 years old.

⇒ Wrapping the variables or keynames with braces and further with parenthesis does the job.

Object Literal Enhancement

- Object literal Enhancement is also called as Destructuring or putting the objects back together as it is the opposite of destructuring.
- When we have variables and their values, we can make them keys and their properties respectively.

```
const name = "Subham";
const age = 24;
const person = {name, age};

console.log(person); // {name: 'Subham', age: 24}
console.log(Object.keys(person)); ['name', 'age']
console.log(Object.values(person)); ['Subham', 24]
```

Spread Operator

- The spread operator is three dot (...) that performs several varities of tasks.

⇒ Combining two Arrays...

```
const arr1 = [1, 2, 3, 4, 5]
const arr2 = [6, 7, 8, 9]
const arr3 = [...arr1, ...arr2]
console.log(arr3); // 1, 2, 3, 4, 5, 6, 7, 8, 9
```

⇒ Printing the last Element

⇒ /* The traditional way */

```
const a = arr1.reverse();
const last = a[0];
console.log(last); // 5
```

⇒ /* Using destructuring Operator */

```
const [last] = arr1.reverse();
console.log(last); // 5
```

⇒ /* Using Spread Operator with destructuring Prator. */

```
const [last] = [...arr1].reverse();
console.log(last); // 5
```

// When we reversed the array and stored the first element on last in destructuring Operator section, we actually reversed the original array. But using "spread Operator" we are making a new array and assigning the reversed value of the array.

⇒ Taking arguments of a function

```
const fun=(...arguments)⇒{
    const [first, ...rest] = arguments;
    const [last, ...left] = rest.reverse();
    const [second, ...args] = rest;
```

```
    console.log(first); // 1
    console.log(last); // 5
    console.log(second); // 5
    console.log(rest); // [5, 4, 3, 2]
    console.log(left); // [4, 3, 2]
    console.log(args); // [4, 3, 2]
}
fun(1, 2, 3, 4, 5);
```

⇒ Combining Objects

```
const obj1 = { Name : "Subham", Age : 24 }
const place = 'Cuttack';
const person = { ... obj1, place };

console.log(person); // { Name : 'Subham', Age: 24, place : 'Cuttack' }
```

Subham