JAVASCRIPT

Content -

Tools to know

1. Figma
2. Browser Stack
3. Rest client / PostMan
4. Redux dev tool
5. Git Bash and git commands.

==

1. Javascript Intro -

JavaScript is a scripting language you can use to make web pages interactive. It is one of the core technologies of the web, along with HTML and CSS, and is supported by all modern browsers.

scripting languages do not require the compilation step and are rather interpreted. For example, normally, a C program needs to be compiled before running whereas normally, a scripting language like JavaScript need not be compiled.

You can simply write your javascript and open it in any browser and see output in the browser's console because javascript auto compiled by v-8 engine.

2. Js Data Types(Primitive, Non-Primitive)

JavaScript provides different data types to hold different types of values. There are two types of data types in JavaScript.

Primitive data type
Non-primitive (reference) data type

JavaScript is a dynamic type language, meaning you don't need to specify the type of the variable because it is dynamically used by the JavaScript engine. You need to use var here to specify the data type. It can hold any type of values such as numbers, strings etc. For example:

```
1. var a=40;//holding number
2. var b="Rahul";//holding string
```

JavaScript primitive data types

There are five types of primitive data types in JavaScript. They are as follows:

Data Type	Description
String	represents sequence of characters e.g. "hello"
Number	represents numeric values e.g. 100 or 99.10 or 0.99 etc
bigInt	Holds lager integer values e.g 123456789012345678901234567890
Boolean	represents boolean value either false or true
Undefined	represents undefined value , var x;
Null	represents null i.e. no value at all, var x= null;

Symbols	Symbols are immutable (cannot be changed) and are unique. For example,
	
	`// two symbols with the same description`
	`const value1 = Symbol('hello');`
	`const value2 = Symbol('hello');`
	`console.log(value1 === value2); // false`

JavaScript non-primitive data types

The non-primitive data types are as follows:

Data Type	**Description**
Object	represents instance through which we can access members
	The object data type can contain: 1. An object 2. An array 3. A date

Javascript Variables

Variables are containers for storing information.

There are 3 Types of Javascript Variables.

1. var
2. let
3. const

var - The var is a javascript predefined keyword.

Creating a variable in JavaScript is called "declaring" a variable.

```
var carName; // undefined

var summer_months = [“April”,”May”,”June”];
```

Let - JavaScript let is a keyword used to declare variables in JavaScript that are block scoped.

```
 let x = 31;
```

===

Let Vs Var : -

var variables are scoped to the immediate function body, while let variables are scoped to the immediate enclosing block denoted by { }.

```
function example() {
    var x = 10;
    let y = 20;

    if (true) {
        var x = 30;
        let y = 40;
        console.log(x, y); // Output: 30 40
    }

    console.log(x, y); // Output: 30 20
}

example();
```

In this example, var and let are used inside a function called example().

With var, variables are function-scoped or globally scoped, but they are not block-scoped. This means that when you declare a variable with var inside a block (such as an if statement or a for loop), the variable is accessible outside of that block. The value of x is modified within the if block and those changes are reflected outside the block as well.

On the other hand, let is block-scoped, which means that variables declared with let are only accessible within the block they are defined in. In the example, the y variable declared with let inside the if block has a separate scope from the y variable declared outside the block. The changes made to y within the if block do not affect the value of y outside the block.

Another Example -

```javascript
function varTest() { // function definition
    var wifi_pass = "123@abc";
    if (wifi_pass == "123@abc") {
        var wifi_pass = "001@xyz";
        console.log(wifi_pass); // 001@xyz
    }
    console.log(wifi_pass); // 001@xyz

}

function letTest() {
    let wifi_pass = "123@abc";
    if (wifi_pass == "123@abc") {
        let wifi_pass = "001@xyz";
        console.log(wifi_pass); // 001@xyz
    }
    console.log(wifi_pass); // 123@abc
}

varTest(); // function call
letTest();
```

Const - The const declaration creates block-scoped constants, much like variables declared using the let keyword. But the difference is **"The value of a const variable can't be changed" i.e known as immutable value.**

```
function constTest() {
    const x = 31;
    x = 99;
    console.log(x);  // error Assignment to constant variable.
}

constTest();
```

Why should we not use var -

Scoping — var variable has a global or function scope, while let variables has a block level scope.

By using let and const, you can take advantage of block scoping, avoid hoisting-related issues, and write more predictable and maintainable code. It is generally recommended to use let for variables that need to be reassigned and const for variables that should remain constant.

Javascript Variable Naming convention -

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter.
- Names can also begin with $ and _ (but we will not use it in this tutorial).
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.(**let class = "hi" // will give an error. class is a reserved keyword in js**)

E.g = let userName = “Peter” or let user_name = “Peter”.

Note -

JavaScript Variables are case-sensitive.

let empName and let EmpName both are different.

— Hoisting --------------------------

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

In Javascript, A variable can be used before it has been declared like below.

```
function abc() {
    console.log(x); // undefined
    var x = 5;
}
abc();
```

How compiler compile this -

```
function abc() {
    var x; // initialized with undefined
    console.log(x);
    x = 5;
  }
  abc();
```

Hoisting with Let & Const : -

variables declared with let and const are also hoisted. But during the compilation it won't be initialized with any value even not with undefined. So it will give the below error.

```
ReferenceError: Cannot access 'x' before initialization.
```

```
function abc() {
   console.log(x);// Cannot access 'x' before initialization
    let x = 5;
  }
  abc();
```

----->> How compiler execute Above Code ---->>

```
function abc() {
    let x; // not initialized with undefined.
    console.log(x);
    x = 5;// this step will not happen.
  }
  abc();
```

Comments -

Single Line Comments

Single line comments start with //.

Multi-line Comments

Multi-line comments start with /* and end with */.

Javascript Operators : -

Types of JavaScript Operators

There are different types of JavaScript operators:

- **Arithmetic Operators ()**

  Var a = 3;

```
Var x = (100 + 50) * a;
```

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- ## Assignment Operators

Assignment operators assign values to JavaScript variables.

The Addition Assignment Operator (+=) adds a value to a variable.

```
Var x = 10;
x += 5; means x = x + 5;
Or let a = 10;
let b = a;
```

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y

/=	**x /= y**	**x = x / y**
%=	**x %= y**	**x = x % y**

Difference between pre and post operators

In the pre-increment operation ++num, the value of num is incremented by 1 before it is used in the console.log statement. So, when we log ++num, we see the output as 6 because num becomes 6 after the increment operation.

```
let num = 5;

// Pre-increment

console.log(++num); // Output: 6

console.log(num);   // Output: 6
```

But In the post-increment operation num++, the value of num is incremented by 1 after it is used in the console.log statement. So, when we log num++, we see the output as 5 and 6.

```
let num = 5;

// Post-increment

console.log(num++); // Output: 5

console.log(num);   // Output: 6
```

- ## Comparison Operators

Operator	**Description**
==	equal to
===	equal value and equal type
!=	not equal

!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

syntax :- condition ? TrueBlockExecuted : FalseBlockExecuted

- **Logical Operators**

Operator	**Description**
&&	logical and
||	logical or
!	logical not

- **Type Operators typeof() - using this we can know type of any variable**
- e.g = let x = “hello”
- console.log(typeof(x)) // string

JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

Types of Functions

- **Function declaration.**

```
<script type="text/javascript">
function multiplyTwoNumbers(p1, p2) { // function declaration
  return p1 * p2;
  console.log("Hi");
}
console.log(multiplyTwoNumbers(4, 3)); // function call
// Or you can store the function call in any variable and console
the variable.
let printOutput = multiplyTwoNumbers(4,3); // function call
console.log(printOutput);
</script>
```

- **Function expression**

```
let printOutput = function multiplyTwoNumbers(p1, p2) {
    return p1 * p2;
  }
  console.log(printOutput(4,3));
```

- **Arrow functions (Introduced in Js ES6 knows as ECMAscript6)**

ECMAScript, also known as JavaScript, is a programming language adopted by the European Computer Manufacturer's Association as ***a standard for performing computations in Web applications.*** *ECMAScript is the official client-side scripting language.*

JavaScript was invented by Brendan Eich in 1995. It was developed for Netscape 2, and became the ECMA standard in 1997.

```
let multiplyTwoNumbers = (x, y) => {
    let result = x * y
    return result;
}
let printOutput = multiplyTwoNumbers(4, 3);
console.log(printOutput);

// If you don’t use return statement you can code like below

let myFunction = (x, y) => {
    let result = x * y
    console.log(result);
}

myFunction(4, 3);
```

- **Calling a Function on any Event like below click event.**

```
<!DOCTYPE html>

<html>

<head>

    <title>Demo</title>

   <script type="text/javascript">

        function myFunction(num1, num2) { // function declaration

          let output =  num1 * num2;

          console.log(output);

        }

    </script>

</head>
```

```
<body>

    <div>

            <h1>Js Functions</h1>

            <button onclick="myFunction(4,3)">click me</button>

    </div>

</body>
```

- **Calling an Arrow Function on any Event like below click event.**

```
<!DOCTYPE html>

<html>

<head>

    <title>Demo</title>

   <script type="text/javascript">

        let myFunction = (p1, p2) => { // function declaration

          let output =  p1 * p2;

          console.log(output);

        }

    </script>

</head>

<body>

    <div>

        <div>
```

```
            <h1>Js Variable Hello</h1>

            <button onclick="myFunction(4,3)">click me</button>

        </div>

    </div>

</body>

</html>
```

Note : - You can save the Js function in the .js file, Then include the js file inside your html page and simply call the function on click.

- **Automatically (self invoked functions)**

Self invoking function :-

<script>

```
(function() {
    // Code inside the self-invoking function
    var message = "Hello, world!";
    console.log(message);
  })();
```

</script>

JavaScript Arrays

An array is a special variable, which can hold more than one value:

```
let cars = ["Honda", "Volvo", "BMW"];

console.log(cars) // Honda volvo BMW

console.log(cars[0]) // Honda
```

JavaScript Array Methods

JavaScript Array pop()

The pop() method removes the last element from an array:

Example

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

fruits.pop(); // Banana,Orange,Apple
```

JavaScript Array push()

The push() method adds a new element to an array (at the end):

Example

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

fruits.push("Kiwi"); // Banana,Orange,Apple,Mango,Kiwi
```

JavaScript Array shift()

The shift() method removes the first array element and "shifts" all other elements to a lower index.

Example

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

fruits.shift(); // Orange,Apple,Mango
```

JavaScript Array unshift()

The `unshift()` method adds a new element to an array (at the beginning), and "unshifts" older elements:

Example

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

fruits.unshift("Lemon"); // Lemon,Banana,Orange,Apple,Mango
```

Merging (Concatenating) Arrays

The `concat()` method creates a new array by merging (concatenating) existing arrays:

Example (Merging Two Arrays)

```
const myBooks = ["Cecilie", "Lone"];

const myNotes = ["Emil", "Tobias", "Linus"];

const myChildren = myGirls.concat(myBoys); // Cecilie,Lone,Emil,Tobias,Linus
```

JavaScript Array splice()

The splice() method is used to modify the contents of an array by **removing** or **replacing** existing elements or **adding** new elements. It allows you to modify an array, rather than creating a new array.

The syntax for the splice() method is as follows:

```
array.splice(start, deleteCount, item1, item2, ...)
```

Adding Items -

```
const fruits = ['apple', 'banana', 'orange'];

// Adding elements deletion count 0

fruits.splice(1,0, 'grape', 'kiwi');

console.log(fruits); // Output: ['apple', 'grape', 'kiwi', ''banana',
'orange']
```

It starts at index 1 (the second position in the array) and specifies 0 for the deleteCount parameter, indicating that no elements should be removed. Instead, 'grape' and 'kiwi' are inserted at the 1 index,

Other example -

```
const fruits = ['apple', 'banana', 'orange'];

// Adding elements deletion count 2

fruits.splice(1,2, 'grape', 'kiwi');

console.log(fruits); // Output: ['apple', 'grape', 'kiwi']

const fruits = ['apple', 'banana', 'orange'];
```

```
// Adding elements deletion count 1

fruits.splice(1,1, 'grape', 'kiwi');

console.log(fruits); // Output: ['apple', 'grape', 'kiwi', 'orange']
```

Using splice() to Remove Elements

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

Example

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

fruits.splice(0, 1); // Orange,Apple,Mango
```

JavaScript Array slice()

The slice() method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

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

const citrus = fruits.slice(1); //Orange,Lemon,Apple,Mango
```

JavaScript Array find()

find the value of the first element with a provided value.

```
<script>

const ages = [3, 10, 18, 20,22,43,55];

function checkAge(age) {

  return age > 18;

}

console.log(ages.find(checkAge)); // 20

</script>
```

Javascript Array Filter -

The filter() method creates a new array filled with elements that pass a test provided by a function.

```
const ages = [32, 33, 16, 40];

function checkAdult(age) {

  return age >18;

}

console.log(ages.filter(checkAdult)); // 32,33,40
```

Javascript Array Sort :-

To sort string array elements use the sort method.

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

console.log(fruits.sort());
```

Note - The sort() method sorts arrays alphabetically by default. **However, keep in mind that the sorting is case-sensitive,** so uppercase letters will come before lowercase letters. If you want a case-insensitive sort, you can provide a custom comparison function to the sort() method.

```
const fruits = ["Banana", "orange", "Apple", "mango"];

fruits.sort((a, b) => a.localeCompare(b));

console.log(fruits);
```

It ignores the casing and sorts the items in ascending order.

To sort Numbers -

Sort method converts elements into string first.

```
const points = [5, 40, 3, 2, 1];
```

When comparing numbers in an alphabetical order, the comparison is based on their string representations. In this case, the string representations of the numbers 5 and 40 are "5" and "40", respectively because the Sort **method converts elements into string.**

When sorting these numbers in alphabetical order, "40" would come before "5" because the comparison is done character by character from left to right. Since "4" comes before "5" in the ASCII(**ASCII stands for "American Standard Code for Information Interchange". It is a character encoding standard that assigns unique numeric codes to represent characters. E.g Uppercase letters: A-Z (65-90), Lowercase letters: a-z (97-122), Digits: 0-9 (48-57) etc..**) character set, "40" is considered smaller than "5" in alphabetical order.

```
const points = [5, 40, 3, 2, 1];

console.log(points.sort()); // [1, 2, 3, 40, 5]
```

To sort the points array in ascending order, you can use a comparison function. like below

```
const points = [5, 40, 3, 2, 1];

points.sort((a, b) => a - b);

/* it compare a and b and if

 1. a - b < 0 , then a comes first

 a - b = 0, nothing will change

 a - b > 0 , b comes first. */

console.log(points); // [1, 2, 3, 40, 5]
```

Execution - step by step

```
const points = [5, 40, 3, 2, 1]; a=5 , b=40
```

```
[5, 40, 3, 2, 1]; a=40, b=3

[5, 3, 40, 2, 1]; a=40; b=2

[5, 3, 2, 40, 1]; a=40, b=1

[5, 3, 2, 1, 40];

[5, 3, 2, 1, 40]; a=5, b=3

[3, 5, 2, 1, 40]; a=5, b=2

[3, 2, 5, 1, 40]; a=5, b=1

[3, 2, 1, 5, 40];

[3, 2, 1, 5, 40]; a=3, b=2

[2, 3, 1, 5, 40]; a=3, b=1

[2, 1, 3, 5, 40]; a=2, b=1

[1, 2, 3, 5, 40];
```

Note- that the elements are sorted based on their numerical value. So, in this case, 40 comes after 5 because numerically it is larger, even though 5 comes after 40 alphabetically.

Sort can be also used with objects like below -

```
const pdt = [{

    name : "IPhone",

    price : 2000
}, {

    name : "MI Note2",

    price : 1000

}, {

    name : "Oneplus Nord 2t",

    price : 500

}]

pdt.sort((a, b) => a.price - b.price);

console.log(pdt);
```

Array includes()-

The includes() method determines whether an array includes a certain value among its entries, returning true or false as appropriate.

```
const array1 = [1, 2, 3];

console.log(array1.includes(2));
// Expected output: true

const pets = ['cat', 'dog', 'bat'];

console.log(pets.includes('cat'));
// Expected output: true

console.log(pets.includes('at'));
// Expected output: false
```

Array Test() -

The test() method in JavaScript is used to check if a specified pattern matches a string. It returns true if there is a match, and false otherwise.

```
const regex = /^\d{3}-\d{3}-\d{4}$/; // Match a phone number in the format
XXX-XXX-XXXX
const phoneNumber = '123-456-7890';

console.log(regex.test(phoneNumber)); // Output: true
```

Array Reduce -

The reduce() method is used to reduce an array to a single value. It iterates over each element of the array and applies a function (often called the reducer function) that accumulates(save sum of two array items)the values and returns a final result.
Syntax -

```
const array = [1, 2, 3, 4, 5];
const sum = array.reduce((accumulator, currentValue) => accumulator +
currentValue, 0);
console.log(sum); // Output: 15
```

Accumulator: It accumulates(save sum of two array items)during each iteration.

Current Value: The current element being processed in the array.

Initial / default value : The default value should be zero.

JavaScript Objects

You have already learned that JavaScript variables are containers for data values.

```
let car = "Fiat";
```

This code assigns a simple value (Fiat) to a variable named car:

Objects are variables too. But objects can contain many values.

In JavaScript, an object is a standalone entity, with a property (key) and value pair.

In the real world an object may be anything which has property(key) and value (value) like a car with color,size,price, max speed, mileage as properties and red, big, 12 Lakhs, 140 km/h, 18 km/h as values and so on...

This code assigns many values (Fiat, 500, white) to a variable named car:

```
const car = {

    type:"Kia",

    model:"1500cc",

    price:100000

    };

    console.log(car.type + "of " + car.model + "is sold at Rs" +
car.price); // Kia of 1500cc is sold at Rs 10000.
```

Objects are combinations of key, value pairs.

Javascript objects methods -

Object.assign()

Object.freeze()

Object.keys()

Object.values()

object.entries();

Delete object

object.assign() - It is used to copy source objects to target objects. Properties in the target object are overwritten by properties in the sources if they have the same key.

```
const target = { a: 1, b: 2 };

const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target); // output: Object { a: 1, b: 4, c: 5 }
```

object.freeze() - The Object.freeze() static method freezes an object. Freezing an object makes existing properties non-writable and non-configurable. A frozen object can no longer be changed:

```
  const car = {

    name : "Tata",

    price: 1200000

  };

  Object.freeze(car);

  car.name = "Hundai";

  car.price = 1800000;

  console.log(car);
```

Object.keys - The Object.keys() method returns an array of all the keys or properties of the given object.

```
const car = {
    price: 1200000,

    color: "red",

    inStock: true

  };

  console.log(Object.keys(car)); // output: Array ["price", "color",
"inStock"]
```

Object.values - The Object.values() method returns an array of all the values of the given object.

```
const car = {

    price: 1200000,

    color: "red",

    inStock: true

  };

  console.log(Object.values(car));

  // output: [1200000, ["red", "white", "black"], true]
```

Object.entries - The Object.entries() method returns an array of a given object's property with both key-value pairs.

```
const car = {

    name: 'Innova',
  price: 1500000

};

let output = Object.entries(car);

console.log(output);//  [["name", "Innova"], ["price", 1500000]]
```

Delete object - delete keyword can be used to delete any object property and its value.

```
const car = {

    price: 1200000,

    color: "red",

    inStock: true

  };

  delete car.price;

  console.log(car); // undefined.
```

Javascript if else statement, for loop , map function and switch statements.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

Use **if** to specify a block of code to be executed, if a specified condition is true
Use **else** to specify a block of code to be executed, if the same condition is false
Use **switch** to specify many alternative blocks of code to be executed.

```
const car = {
    price: 1200,
    color: "red",
    inStock: true
  };
  if(car.price < 1500) {
  console.log("i can buy");
  } else {
  console.log("it is out of my budget");
  }
```

If else ladder -

```
const car = {
    price: 1200,
    color: "red",
    inStock: true
};
if (car.price < 1500) {
    console.log("i can buy");
} else if (car.price >= 1500 && car.price <= 2000) {
    console.log("i need to arrange more money");
} else if (car.price >= 2000 && car.price <= 3000) {
    console.log("i can take car loan");
} else {
    console.log("out of my budget.");
}
```

Switch statement -

The switch statement is used to perform different actions based on different conditions.
In a switch statement, the expression is the value or variable that is being evaluated against the different case conditions. It determines which code block should be executed based on the matching condition.

The expression in a switch statement is typically placed within the parentheses after the switch keyword. Here is the general syntax:

```
switch (expression) {
    case value1:
      // code to be executed if expression matches value1
      break;
    case value2:
      // code to be executed if expression matches value2
      break;
    // additional cases...
    default:
      // code to be executed if expression doesn't match any case
  }
```

```
let day = new Date().getDay();
console.log(day);
switch (day) {
    case 0:
        day = "Sunday";
        break;
    case 1:
        day = "Monday";
        break;
    case 2:
        day = "Tuesday";
        break;
    case 3:
        day = "Wednesday";
        break;
    case 4:
        day = "Thursday";
        break;
    case 5:
        day = "Friday";
        break;
    case 6:
        day = "Saturday";
}
console.log(day);
```

For objects -

```
const tataCar3 = {
    name : "Tata Harrier",
    price : 2100000,
    inStock : 10
}

function checkPrice() {
   const pricesOfCar = tataCar3.price;
   console.log(pricesOfCar);
    switch(true) { // will execute only true block
        case (pricesOfCar < 1200000) :
            return "I want to buy"
            break;
        case (pricesOfCar == 1200000) :
            return "I can plan";
            break;
        case (pricesOfCar > 1200000 && pricesOfCar < 2200000) :
            return "out of budget";
            break;
        default :
            return "Nothing"
    }
}
console.log(checkPrice());
```

Small assignment - check coupon code

Html file -

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <title>coupon</title>
        <script type="text/javascript" src="./couponCode.js"></script>
    </head>
    <body>
        <h1 id="output">Output</h1>
        <input type="text" placeholder="Enter Coupon Code"
id="txtCoupon"/>
        <button type="button" onclick="checkCoupon()">Check
Coupon</button>

    </body>
</html>
```

Js file -

```
const checkCoupon = () => {
    const val = document.getElementById("txtCoupon").value;
    let outcome = "";
    switch (val) {
        case "grab40":
            outcome = "Congrats you got Rs 40 off...";
            break;
        case "grab80":
            outcome = "Congrats you got Rs 80 off...";
            break;
        case "grab100":
            outcome = "Congrats you got Rs 100 off...";
            break;
        default :
            outcome = "Better Luck next time"
    }
    document.getElementById("output").innerHTML = outcome;
}
```

For Loop -

Allows you to repeatedly execute a block of code for a specified number of times or iterate over a collection of elements.

Syntax -

```
for (initialization; condition; updation) {
    // code to be executed in each iteration
}
```

```
const cars = ["BMW", "Volvo", "Ford", "Honda", "Tata"];

for (let values = 0; values < cars.length; values++) {
    let arrayItems = cars[values];
    console.log(arrayItems); // "BMW", "Volvo", "Ford", "Honda", "Tata"
}
```

With objects -

```
const cars = [{
    name : "BMW",
    price : 1200,
    },{
    name : "Volvo",
    price : 16000,
    }];
   console.log(cars.length);

   for (let carsLength = 0; carsLength < cars.length; carsLength++) {
       let arrayItems = cars[carsLength];
       console.log(arrayItems);
   }
```

Javascript Map Function -

The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.

```
const array1 = [1, 4, 9, 16];

array1.map(items => {
    console.log(items * 2); // Expected output: Array [2, 8, 18, 32]
});

const cars = ["BMW", "Volvo", "Ford", "Fiat", "Audi"];

const carList = cars.map(names => {
    return names;
});
Or
const carList = cars.map(names => names);
console.log(carList); // ["BMW", "Volvo", "Ford", "Fiat", "Audi"]
```

With Objects -

```
const cars = [
    {
      name: "BMW",
      price: 1200,
    },
    {
      name: "Volvo",
      price: 16000,
    },
  ];

  const prices = cars.map((car) => {
    return car.price;
  });

  console.log(prices);
```

Javascript Closures

A closure is created when a function is defined inside another function and has access to the outer function's variables, even after the outer function has finished executing. In simpler terms, a closure "closes over" the variables it needs, preserving their values within its own scope.

```
function outerFunction() {

  var outerVariable = 'I am from the outer function';

  function innerFunction() {

    console.log(outerVariable);

  }

  return innerFunction;

}

var closure = outerFunction();

closure(); // Output: 'I am from the outer function'
```

In this example, outerFunction defines a variable called outerVariable and declares an inner function called innerFunction. The innerFunction has access to the outerVariable, even though it is executed outside the scope of outerFunction. This is because the inner function forms a closure that retains a reference to the variables in its outer scope.

Closures are useful in -

Data Privacy: Closures allow you to create private variables that are not accessible from outside the function. This helps in encapsulating data and preventing unwanted access or modification.

JavaScript Events

HTML events are "things" that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can "react" to these events.

HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, with JavaScript code, to be added to HTML elements.

With single quotes: ‘’ or With double quotes: “”

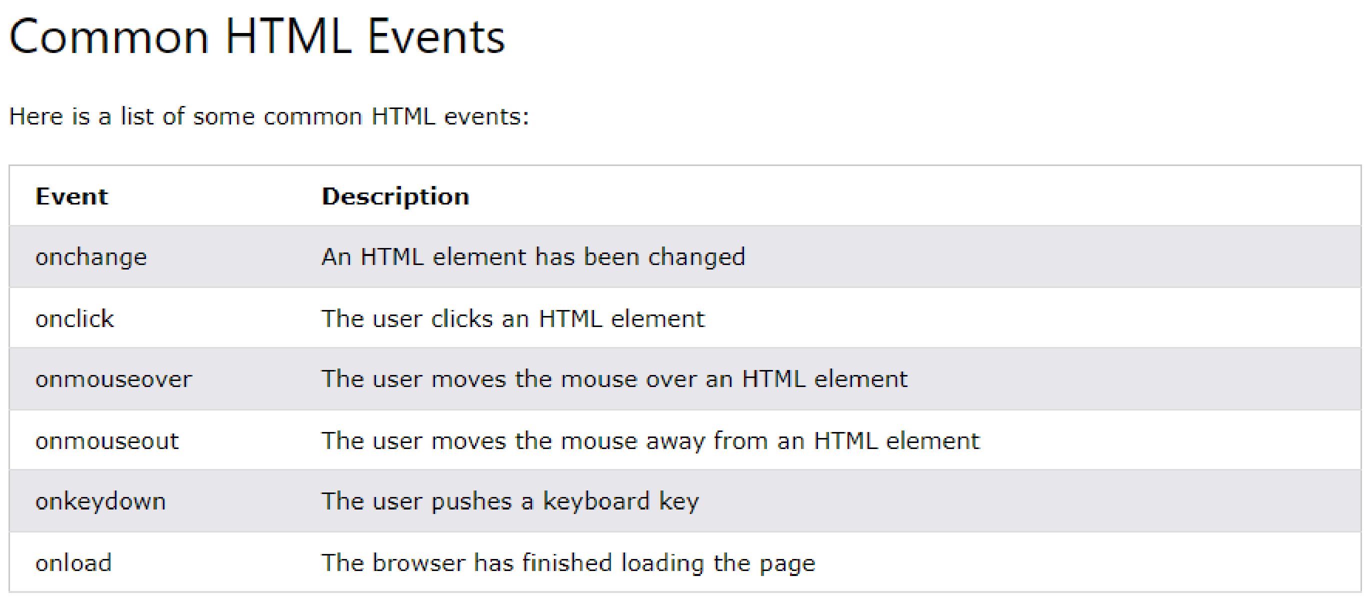
```
<!DOCTYPE html>

<html>

<head>

    <script type="text/javascript">

        const ages = [3, 10, 18, 20, 22, 43, 55];

        function showAllAbove() {

            function checkAge(age) {

                return (age > 18);

            }
            let peopleAboveEighteen = ages.filter(checkAge);

            document.getElementById("userAge").innerHTML =
peopleAboveEighteen;

        }

        // using Arrow function

        const ages1 = [3, 10, 18, 20, 22, 43, 55];

        const showAllBelow = () => {

            const checkAge = (age) => {

                return (age < 18);

            }

            let peopleAboveEighteen = ages1.filter(checkAge);

            document.getElementById("userAge").innerHTML =
peopleAboveEighteen;

        }

    </script>

</head>

<body>

    <h1 id="userAge">People Above 18</h1>

    <button onclick="showAllAbove()">Age Above 18</button>

   <button onclick="showAllBelow()">Age Below 18</button>

</body>

</html>
```

Taking input from textbox and checking the condition on text change

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">
        function checkAge() {
            let input = document.getElementById("txt1").value;
            console.log(input);
            if (input > 0 && input < 110) { // MIn Age is 1 and Max is 110
                if (input > 18) {
                    alert("you can vote");
                    document.getElementById("showResult").innerHTML =
input;
                } else {
                    alert("you are not eligible");
                }
            } else {
                alert("Not a valid Age");
                document.getElementById("showResult").innerHTML = "";
            }

        }

    </script>

</head>

<body>

    <h1>Your Age is - <span id="showResult"></span></h1>

    <input type="text" id="txt1" onchange="checkAge()" />

</body>

</html>
```

Taking input from textbox and checking condition on button click

```
<!DOCTYPE html>

<html>

<head>

    <script type="text/javascript">

        function checkAge() {

            let input = document.getElementById("txt1").value;

            console.log(input);

            if (input > 0 && input < 110) {

                if (input > 18) {
                    alert("you can vote");

                    document.getElementById("showResult").innerHTML =
input;

                } else {

                    alert("you are not eligible")

                }

            } else {

                alert("Not a valid Age");

                document.getElementById("showResult").innerHTML = "";

            }

        }

    </script>

</head>

<body>

    <h1> Your age is - <span id="showResult"></span></h1>

    <input type="text" id="txt1" />

    <button type="button" onclick="checkAge()">Check Age</button>

</body>

</html>
```



Common HTML Events

Here is a list of some common HTML events:

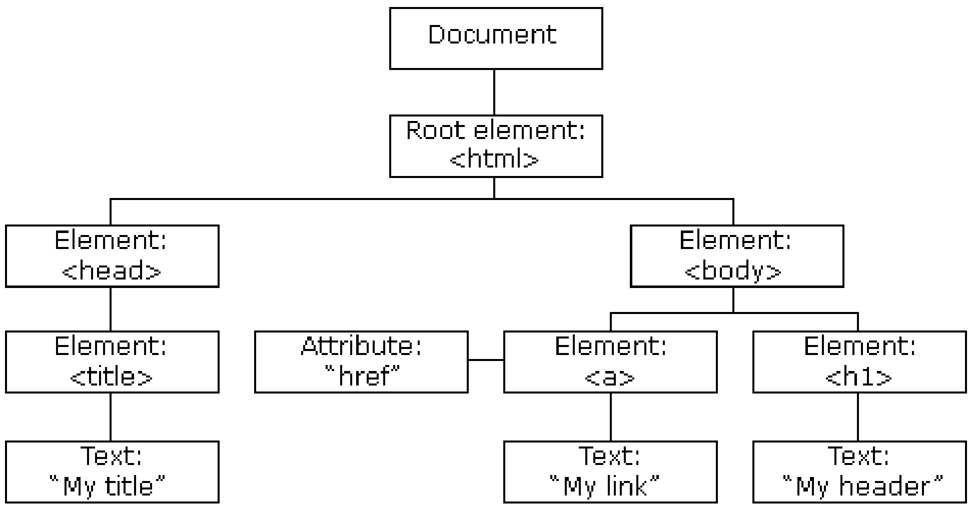
Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JS DOM Manipulation

When a web page is loaded, the browser creates a Document Object Model of the page.

The HTML DOM model is constructed as a tree of Objects:

The HTML DOM Tree of Objects



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

What is the HTML DOM?

The HTML DOM is a standard object model and programming interface for HTML. It defines:

- The properties of all HTML elements

- The methods to access all HTML elements (onclick, onload, onchange etc.)
- The events (click, change, submit, load etc.) for all HTML elements

In other words: The HTML DOM is a standard for how to get, change, add, or delete HTML elements using javascript.

JavaScript ES6 — write less, do more

ES6 stands for ECMAScript 6. ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.

Arrows functions

Classes

Enhanced object literals

Template strings

Destructuring

Map method

Spread operator and rest parameters

Modules (import, export of file or function)

Promises

Arrow functions - Arrow functions allow a short syntax for writing function expressions.

You don't need the **function** keyword, the **return** keyword (if you write in the same line), and the curly brackets.

They are syntactically similar to the related feature in C#, Java 8 and CoffeeScript.

```
const sum = (a,b) => {
     return a + b ;
}

console.log(sum(3,5));
```

Can also be like -

```
const sum = (a,b) => a + b ;

console.log(sum(3,5));
```

Class - In ECMAScript 5 and earlier, classes never existed in JavaScript. Classes are introduced in ES6 which look similar to classes in other object oriented languages, such as Java, PHP, etc., however they do not work exactly the same way. ES6 classes make it easier to create objects, implement inheritance by using the `extends` keyword, and reuse the code.

In ES6 you can declare a class using the new `class` keyword followed by a class-name

Classes are a template for creating objects. Classes hold properties and methods also known as members of class.

To access these properties and methods we need to create an object of that class also known as instance and then we can access any member (any property and methods).

syntax -

```
class class_name {

  constructor() {
  }
}

let classObj = new class_name(); // creating object of class

console.log(classObj.member_name); // access class member
```

Simple example -

```
class abc {
    constructor(name) {
        this.myname = name;
    }
    helloUser() {
        return "hello" + this.myname;
    }
}

let myClassObj = new abc("abc");
console.log(myClassObj.myname); // abc
console.log(myClassObj.helloUser()); // hello abc
```

Another example -

```
class MyHouseSize {
    // Class constructor
    constructor(length, width) {
        this.lengthOfMyHouse = length;
        this.widthOfMyHouse = width;
    }
    // Class method
    getArea() {
        return this.lengthOfMyHouse * this.widthOfMyHouse;
    }
}

let classObj = new MyHouseSize(600, 20);
console.log(classObj.lengthOfMyHouse);
console.log(classObj.widthOfMyHouse);
console.log("My Home size is - " + classObj.getArea() + " / square
meter.");
```

======== Inherit one class into other ===========

Inheritance is a mechanism in which one class acquires the property and methods of another class.
For example, a child inherits the properties of his/her parents(Like - height, blood group, eyes color etc.). With inheritance,
we can reuse the properties and methods of the existing class.
Hence, inheritance facilitates Reusability and is an important concept of OOPs

```
class MyHouseRegistrationNumber { // Parent Class
    constructor() {
        this.RegNum = "MH23432432HR"
    }


}
class MyHouseSize extends MyHouseRegistrationNumber { // child
    // Class constructor
    constructor(length, width) {
        super();
        this.lengthOfMyHouse = length;
        this.widthOfMyHouse = width;
    }

    // Class method
    getArea() {
        console.log(this.RegNum);
        if (this.RegNum !== null && this.RegNum == "MH23432432HR") {
            return this.lengthOfMyHouse * this.widthOfMyHouse + " /square
meter.";
        } else {
            return "Your House is not Registered.";
        }

    }
}

let classObj = new MyHouseSize(600, 20);
console.log("My Home size is - " + classObj.getArea());
```

Accessing child class properties into parent class -

```
class AreaOfRectangle { // parent or base class
    constructor() {
        this.length; // passing the variables of the child class
constructor inside the parent class.
        this.width;
    }
    area() {
        console.log("Length is =", this.length);
        console.log("Width is =", this.width);
        console.log("Area is =", this.length * this.width);
    }
}
```

```
class LengthAndWidthOfRectangle extends AreaOfRectangle { // child derived
class
    constructor(len, wid) {
        super();
        this.length = len;
        this.width= wid;
    }
}

let obj = new LengthAndWidthOfRectangle(200,30);
obj.area();
```

Enhanced Object Literals - Object literals make it easy to quickly create objects with properties inside the curly braces. To create an object, we simply notate a list of key: value pairs delimited by comma. ES6 makes the declaring of object literals concise and thus easier. It provides a shorthand syntax for initializing properties from variables.

Normal Way -

```
let price = 34232423;
let car = {
 name : "kia",
 price : price
}

console.log(car.price); // 34232423
```

Using Enhanced Object Literal -

```
let price = 34232423;
let car = {
 name : "kia",
 price
}

console.log(car.price); // 34232423
```

Template String - Template Literals use back-ticks (``) rather than the quotes ("") to define a string:

Syntax -

` ${variable_name}`

Older way -

```
let todaysOffer = "10% off on any Mobile";
let endDate = "1 April.";

console.log ("Hurry " + todaysOffer + " shop now."  + "Offer till " +
endDate);
```

Using template string -

```
let todaysOffer = "10% off on any Mobile.";
let endDate = "1 April.";

console.log (`Hurry ${todaysOffer} shop now. Offer till ${endDate}`);
```

Destructuring in JS -

We may have an array or object that we are working with, but we only need some of the items contained in these.

Destructuring makes it easy to extract only what is needed.

```
let grocery = ["sugar", "milk", "tea leaf", "salt", "flour", "oil", "chili
powder","turmeric"];

console.log(`To make tea we need ${grocery[0]} and ${grocery[1]} and
${grocery[2]}`);
```

```
// To make tea we need sugar and milk and tea leaf
```

we can only choose , what we need using destructuring -

```
let grocery = ["sugar", “pea”, "milk", "tea leaf", "salt", "flour", "oil",
"chili powder","turmeric"];

let [item1, ,item2, item3] = grocery;

console.log(`To make tea we need ${item1} and ${item2} and ${item3}`);
```

```
// To make tea we need sugar and milk and tea leaf
```

With Objects -

```
let grocery = {
    item1: "sugar",
    item2: "milk",
    item3: "tea leaf",
    item4: "salt",
    item5: "flour",
    item6: "oil",
    item7: "chili powder",
    item8: "turmeric"
};

let { item1, item2, item3 } = grocery;

console.log(`To make tea we need ${item1} and ${item2} and ${item3}`);
```

Using destructuring inside function -

```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    age: 30,
    city: "New York",
    country: "USA"
};

let displayUserName = ({ firstName, lastName }) => {
    console.log(`Hello ${firstName} ${lastName}. You are Welcome to Google
Head Office.`);
}

displayUserName(person);
```

Map function -

map() is a predefined function used to loop through array elements/items.
map() creates a new array for array elements.
map() does not execute for empty elements.
map() does not change the original array.
map() also help to modify the existing array items, like Multiplying arrays having items like [2,4,6,8] by 2. Will give the result [4,8,12,16].

Syntax -

```
arrayName.map((params) => {
     console.log(params);
});
```

Example -

```
let grocery = ["sugar", "milk", "tea leaf", "salt", "flour", "oil", "chili
powder", "turmeric"];
```

```
grocery.map((items) => {
    console.log(items);
});

const groceryItems = grocery.map((items) => {
    return (` ${items} is available`);
});

console.log(groceryItems); // ['sugar', 'milk', 'tea leaf', 'salt',
'flour', 'oil', 'chili powder', 'turmeric']
```

Note : - The map() method allows you to iterate over an array and modify its each element.

For example, suppose you have the following array element:

```
const array1 = [1, 4, 9, 16];

const result = array1.map((x) => {
 return x * 2;
});

console.log(result); // [2, 8, 18, 32]
```

Spread operator -

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

```
const arr1 = [1,3,5,7];
const arr2 = [2,4,6,8];

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

Objects -

```
const obj1 = {
    sugar: 100,
    salt: 20
};
const obj2 = {
    rice: 50,
    pulse: 120
};

const newObj = { ...obj1, ...obj2 };
console.log(newObj);
```

Rest parameters -

The rest parameter is an improved way to handle function parameters, allowing us to more easily handle various inputs as parameters in a function. The rest parameter syntax allows us to represent an indefinite number of arguments. With the help of a rest parameter, a function can be called with any number of arguments

Using rest parameter -

Advantage - when you need to pass multiple values inside a function call, you also need to pass multiple arguments inside function as parameters. By using rest parameters you just need to call one parameter by using . . . (three dots)

Simple function -

```
function sum(a, b, c, d) {
    console.log(a + b + c + d); // 1 2 3 4


}
```

```
sum(1, 2, 3, 4);
```

Sum of numbers using rest parameter -

```
function sum(...theArgs) {
    let total = 0;
    theArgs.map(items => {
        total = total + items;
    });
    console.log(total);
}

sum(1, 2, 3, 4);
```

Another way -

```
// function definition
function getSum(...sumOfValues) { // rest parameter
    console.log(sumOfValues);
    const sum = sumOfValues.reduce((accumulator, currentValue) =>
accumulator + currentValue, 0);
    console.log(sum);
}

getSum(1,2,3,4,5,6,7,8,9,10,11) // function call
```

Modules (import and export) -

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain a code-base.

modules have two parts -

export - export your variables or functions from file1.js (where file1 is just any javascript file name).

import - import your variables and functions into file2.js (where file2 is just any javascript file name).

Example -

file1.js

```
let msg = "Hello";
// below syntax
export default msg;
```

Htmlfile file2.html

```
<!DOCTYPE html>
<html>

<head>
    <title>Demo</title>
    <script type="module">
        import abc from './file1.js';
        console.log(abc);
        document.getElementById("one").innerHTML = abc;
    </script>
</head>

<body>
    <div>
        <div>
            <h1 id="one">Js Variable Hello</h1>
        </div>
    </div>

</body>

</html>
```

Exporting multiple -

file1.js ------------------

```
let msg1 = "Hello";
let msg2 = "bye!!";
// Exporting multiple modules
export {msg1, msg2}
```

Importing -
index.html ---
```
<!DOCTYPE html>
<html>

<head>
    <title>Demo</title>
    <script type="module">
        import { msg1, msg2 } from './file1.js';
        document.getElementById("one").innerHTML = `${msg1} Everyone.
Sorry I need to go ${msg2}`;
    </script>
</head>

<body>
    <div>
        <div>
            <h1 id="one">Js Variable Hello</h1>
        </div>
    </div>
</body>

</html>
```

Another Example -

Selectedplayer.js ====================== (js file) ============

```
let bumrahData = {
    name: "Bumrah",
    profession: "bowler"
}
let viratData = {
    name: "virat",
    profession: "Batsman"
}
export { bumrahData, viratData };
```

ShortListedForWorldCup.html ========= (js file) ================

```
<!DOCTYPE html>
<html>

<head>
    <title>Demo</title>
    <script type="module">
        import { bumrahData, viratData } from './file1.js';
        document.getElementById("output").innerHTML =
Object.entries(bumrahData);
        document.getElementById("output1").innerHTML =
Object.entries(viratData);
    </script>
</head>

<body>
    <div>
        <div>
            <h1 id="output">Js Variable Hello</h1>
            <h1 id="output1">Js Variable Hello</h1>
        </div>
    </div>
</body>

</html>
```

Import and Export Functions -

todayDate.js

```
const showTodayDate = () => {
    let date = Date();
    console.log(date);
    return date;
}
export default showTodayDate;
```

ShowDate.js

```
<!DOCTYPE html>
<html>

<head>
    <title>Demo</title>
    <script type="module">
        // When you export as default, You can import with any name
        // Exery Js file must have only one module as default export. not
multiple
        import tdyDate from './todayDate.js';
        tdyDate();
        document.getElementById("one").innerText = tdyDate();
    </script>
</head>
<body>
    <div>
        <div>
            <h1 id="one">Js Helloooo</h1>
        </div>
    </div>
</body>

</html>
```

Calling Event using Modules -

index.html

```
<!DOCTYPE html>
<html>

<head>
    <title>Form Validation</title>
    <script type="module">
```

```
        import validate from './form1.js';
        document.getElementById("btnSubmit").addEventListener('click',
function() {
            validate();
        });
    </script>
</head>

<body>
    <h1 id="output">Output</h1>
    <input type="text" id="txtName" placeholder="Enter Name" />
    <input type="text" id="txtEmail" placeholder="Enter Email" />
    <input type="text" id="txtPhone" placeholder="Enter Phone"
maxlength="10"/>
    <button type="button" id="btnSubmit">Submit</button>
</body>

</html>
```

form1.js

```
const validate = () => {
    const name = document.getElementById("txtName").value;
    const email = document.getElementById("txtEmail").value;
    const phone = document.getElementById("txtPhone").value;
    const values = []; // store these values and display on UI
    const nameRegex = /^[a-zA-Z]+(([',. -][a-zA-Z ])?[a-zA-Z]*)*$/;
    const mailRegex = /^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$/;
    const phoneRegex = /^[0-9]{10}$/;
    if (name !== '' && nameRegex.test(name)) {
        values.push(name);
    } else {
        alert("Please Provide Name with Valid Value");
    }
    if (email !== '' && mailRegex.test(email)) {
        values.push(email);
    } else {
        alert("Please Provide Email with Valid Value");
    }
    if (phone !== '' && phoneRegex.test(phone)) {
        values.push(phone);
    } else {
        alert("Please Provide Phone with Valid Value");
    }
    document.getElementById("output").innerHTML = values;
};
export default validate;
```

Synchronous And Asynchronous Server Requests -

A synchronous request is a type of request where the JavaScript execution is paused while waiting for the response from the server. In other words, the code that initiates the request will not proceed until the response is received or until a timeout occurs.
when there are multiple server side requests and you are doing it synchronously, if any one request fails others will also fail.

Synchronous requests were commonly used in the past, but they have several drawbacks

1. **Blocking Behavior:** Synchronous requests block the main thread of the browser, which means that other JavaScript code and user interactions are frozen until the request completes. This can make the page unresponsive and negatively affect the user experience.

2. **Increased Perceived Load Time:** Since synchronous requests block the main thread, the page appears to load slower and may give the impression that the website is not functioning properly.

3. **Limited Features:** Synchronous requests don't allow for additional functionality like timeouts or error handling in a straightforward manner.

Asynchronous -

An asynchronous request is a type of request where the JavaScript code continues to execute without waiting for the response from the server. Instead of blocking the execution of other code, asynchronous requests work in the background, allowing the rest of the code to continue running.

Asynchronous requests are commonly used in modern web development because they provide a better user experience and make web applications more responsive. When making an asynchronous request, the browser sends the request to the server and immediately continues executing the remaining JavaScript code. Once the server responds to the request, a callback function or a **Promise** is used to handle the response and update the page accordingly.

Javascript Ajax -

AJAX stands for **asynchronous** javascript and xml. An Ajax can -

Read data from a web server - after the page has loaded.

Update a web page without reloading the page.

Send data to a web server - in the background.

```
<!DOCTYPE html>

<html>

<head>

    <title>API Data using XMLHttpRequest</title>

</head>

<body>

    <p id="data"></p>

    <script type="text/javascript">
```

```
        const apiUrl =

            'https://jsonplaceholder.typicode.com/comments';

        const paraTag = document.getElementById('data');

        // Step 1: Create XMLHttpRequest object

        const xhr = new XMLHttpRequest();

        // Step 2: Set up callback function to handle response

        xhr.onreadystatechange = function () {

            if (xhr.readyState === 4 && xhr.status === 200) {

                // Step 4: Process the received data in the callback

                const responseData = JSON.parse(xhr.responseText);

                // Step 5: Update the content of the <p> tag with


                paraTag.textContent = JSON.stringify(responseData);

            }

        };

        // Step 3: Make the API request

        xhr.open('GET', apiUrl);

        xhr.send();

    </script>

</body>

</html>
```

The XMLHttpRequest object is created, and its onreadystatechange property is set to a callback function. When the request is completed and the response is received, the callback function is called, and it checks if the readyState is 4 (request is done) and the status is 200 (OK). If these conditions are met, it means the response is successful, and we can parse the received JSON data and update the content of the <p> tag with the JSON data.

When you open the HTML file in a web browser, it will fetch data from the provided API and display it inside the <p> tag. Note that the response data is displayed as a JSON stringified representation.

Ready State All Values -

The readyState property represents the current state of the request.

0 (UNSENT): The request has not been initialized yet. The open() method has not been called.

1 (OPENED): The request has been set up. The open() method has been called, but the send() method has not been called yet.

2 (HEADERS_RECEIVED): The request has been sent, and the response headers have been received. However, the body of the response has not been received yet.

3 (LOADING): The request is in the process of fetching the response body (partial data has been received). This state is typically used for streaming data.

4 (DONE): The request is complete. The entire response has been received, and the operation is finished.

All Server Status values -

In the context of XMLHttpRequest, the status property represents the HTTP status code returned by the server as part of the response.

1. 200 (OK): The request was successful, and the server has returned the requested data.

2. 201 (Created): The request was successful, and a new resource has been created as a result.

3. 204 (No Content): The request was successful, but there is no data to return (e.g., for a successful DELETE request).

4. 400 (Bad Request): The server cannot understand the request, usually due to a client-side error (e.g., malformed request syntax).

5. 401 (Unauthorized): The client must authenticate itself to get the requested response, often indicating that the user needs to log in or provide valid credentials.

6. 403 (Forbidden): The client does not have permission to access the requested resource.

7. 404 (Not Found): The requested resource could not be found on the server.

8. 500 (Internal Server Error): A generic error message indicating that something has gone wrong on the server-side.

9. 503 (Service Unavailable): The server is not ready to handle the request. Commonly used when the server is undergoing maintenance or overloaded.

Promises(Modern way of calling/using Ajax)

Promises are a new feature of ES6. It’s a method to write **asynchronous** code. It can be used when, for example, we want to fetch data from an API (Application Programming Interface - It gives you the ability to get data from any server and also send data to a server.)

(https://jsonplaceholder.typicode.com/comments), or when we have a function that takes time to be executed. Promises make it easier to solve the problem.

The Promise takes two parameters: **resolve** and **reject** to handle an expected error.

How to use Promise or ways of calling API

Fetch

```
function showData() {

    fetch('https://jsonplaceholder.typicode.com/posts')

        .then((response) => response.json())

        .then((data) => console.log(data))

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

}
```

Axios :-

Axios is a promise based HTTP client for the browser and Node.js. Axios makes it easy to send asynchronous HTTP requests to REST endpoints and perform Get/Post/Put/Delete operations. It can be used in plain JavaScript or with a library such as Vue or React.

```
<!DOCTYPE html>

<html>

<head>

    <title>API Data using XMLHttpRequest</title>

</head>

<body>

    <p id="data"></p>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/axios/1.1.3/axios.min.js"></script>

    <script type="text/javascript">

        axios.get('https://jsonplaceholder.typicode.com/posts')

            .then(function (response) {

                console.log(response.data);

                document.getElementById("data").innerHTML =
                    JSON.stringify(response.data);

            })

            .catch(function (error) {

                console.log(error);

                document.getElementById("data").innerHTML =

                    JSON.stringify(error.message);

            });

    </script>

</body>
```

Async await

Cleaner Code: async/await reduces the nesting of callbacks and avoids the need for multiple then() chains, making the code more concise and easier to follow.

Error Handling: With async/await, you can use try-catch blocks to handle errors in a synchronous style, which is more natural and intuitive for developers.

Sequential Execution: async/await allows you to write asynchronous code that executes in a more sequential manner, resembling traditional synchronous code. This makes it easier to understand the flow of the program.

Easier Debugging: Debugging asynchronous code can be challenging. async/await makes debugging simpler by providing clearer stack traces and error messages.

```
<!DOCTYPE html>

<html>

<head>

    <title>JSONPlaceholder Posts</title>

</head>

<body>

    <h1>JSONPlaceholder Posts</h1>

    <p id="postContent">Loading...</p>

    <script>

        async function fetchPosts() {

            try {

                const response = await
fetch('https://jsonplaceholder.typicode.com/posts');

                const data = await response.json();
                // Get the <p> element to display the data

                const postContent =
document.getElementById('postContent');

               // Prepare the content to be displayed in the <p> tag

                let content = '';

                data.map(post => {

                    content += `<strong>Title:</strong>
${post.title}<br><strong>Body:</strong> ${post.body}<br><br>`;

                });

                // Update the <p> tag content with the fetched data

                postContent.innerHTML = content;

            } catch (error) {

                console.error('Error fetching data:', error);

                // Display an error message if there was an issue
fetching the data

                postContent.innerHTML = 'Error fetching data. Please
try again later.';

            }
```

```
        }

        // Call the fetchPosts function when the page loads

        fetchPosts();

    </script>

</body>

</html>
```

Post data using Axios and Asyn Await -

```
<!DOCTYPE html>

<html>

<head>

    <title>Signup Form</title>

</head>

<body>

    <h1>Signup Form</h1>

    <form id="signupForm">
```

```
        <input type="text" id="username" placeholder="UserName">

        <input type="email" id="email" placeholder="Email Id">

        <input type="password" id="password" placeholder="Password">

        <button type="submit">Signup</button>

    </form>

    <script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

    <script>

        // Handle form submission


document.getElementById('signupForm').addEventListener('submit',

            async function(event) {

                event.preventDefault();

                // Get form data

                const username =
document.getElementById('username').value;

                const email =
document.getElementById('email').value;
```

```
                const password =
document.getElementById('password').value;

                // Create signup data object

                const signupData = {

                    username: username,

                    email: email,

                    password: password

                };

                try {

                    // Make a POST request to the server

                    const response = await

axios.post('https://jsonplaceholder.typicode.com/posts',

                            signupData,

                             // headers

                            {

                                'Content-Type': 'application/json',
                                Authorization: 'Bearer
your_access_token',

                                // Add your access token here (if
needed)

                            });

                    alert(response.data.message);

                    // Clear the form

                    document.getElementById('signupForm').reset();

                } catch (error) {

                    console.error('Error during signup:', error);

                }

            });

    </script>

</body>

</html>
```

HTTP headers

A crucial part of the Hypertext Transfer Protocol (HTTP), which is the foundation of data communication on the World Wide Web. When you request a web page or any other resource from a web server, the server responds with an HTTP response that includes a set of headers. These headers provide additional information about the response and can be used by both the client (usually a web browser) and the server to handle the communication.

Here are some commonly used HTTP headers and their purposes:

Content-Type: Indicates the media type of the data in the response body. For example, "text/html" for HTML pages or "application/json" for JSON data.

Content-Length: Specifies the size, in bytes, of the response body.

User-Agent: Provides information about the client, typically the web browser or user agent software being used.

Authorization: The HTTP Authorization request header can be used to provide credentials that authenticate a user agent with a server, allowing access to a protected resource.

Cache and Cookie Browser’s Features -

Caching and cookies are two different mechanisms used in web applications to improve performance and enhance user experience, but they serve different purposes:

Cache:

Caching is a technique used to store copies of frequently accessed web resources (like HTML pages, images, CSS, JavaScript files, etc.) closer to the client-side, reducing the need to fetch them repeatedly from the server.

Cookie:

A cookie is a small piece of data stored by the client-side (usually in the web browser) to maintain stateful information about the user's interactions with a website.

Cookies are often used for session management, tracking user preferences, and maintaining user login status.

- Javascript Important Questions -

1. **Data types in Javascript.**

   Ans - There are 7 types of data types.

   Number, string, boolean, array, objects, null, undefined.

2. Difference between var, let and const.
3. What is hoisting in Javascript?
4. What is closure in Javascript?
5. Name some array and object methods and their uses.

   Array Methods -

   pop(), push(), shift(), unshift(), concat(),

   splice(), slice(), find(), filter(), sort().

   JavaScript Objects Methods -

   Object.assign(), Object.freeze(), Object.keys()

   Object.values(), object.entries(); Delete object

6. What is Javascript DOM?
7. What are all Javascript ES6 features?
8. What is the difference between normal function and arrow function?
9. What is destructuring in JS, give one example.
10. What is class ? how to inherit one class into another?
11. What is the difference between spread operator and rest   parameter.
12. What is a promise in Javascript?
13. What is the difference between fetch and Axios?
14. What is the difference between synchronous and asynchronous calls?
15. Is javascript oops based?
16. **Is javascript loosely typed or strict type?**

    Ans -

Loosely typed" means language does not bother with types too much, and does conversions automatically,

i.e. you can easily add a string to an integer and get the result as a string (or even as integer in some languages like Perl).

JavaScript is both dynamically and loosely typed language.

17. **What is Cross domain Error ?**

Ans - Cross domain or origin is a mechanism that allows restricted web resources or domain to be requested from another domain.

Cross domain error occurs when -

i. Js try to access info. which it should not Like -

you try to read cookies from another domain.

if you try to do XMLHTTP hijack, steal visitor sessions etc.

It is a security feature which is now standard in all browsers.

18. **What are HTTP Headers used for ?**

Ans - They provide required info. about request and response.

19. **Can we do inheritance in es5 ?**

Ans - Yes we can do inheritance using prototypes. Each object is attached with a prototype.

```
function Bird(type, color) {

    this.type = type;

    this.color = color;

}

Bird('Indian Bird', 'Red');

Bird.prototype.name = 'Parrot';
```

20. diff. b/w normal func. and arrow func. ?

Ans - Arguments and objects are not available in arrow function also arrow fun do not have its own this.

They have some nice properties which allow them to work well as callback functions.

21. What is async await used for ?

Ans -

An async function can contain an await expression that pauses the execution of

the async function and waits for the passed Promise's resolution,

and then resumes the async function's execution and returns the resolved value.

Remember, the await keyword is only valid inside async functions.

```
async function asyncCall() {
    console.log('calling');

    var result = await resolveAfter2Seconds();

    console.log(result);

    // expected output: 'resolved'

}

asyncCall();
```

- Javascript Important Programs -

1.Write output of below programs -

```
console.log(1+"2"+"3");

console.log(1++"2"+"3");

console.log(1++);

console.log(1++2);
```

2.check duplicate value in array -

```
const array = [1, 4, 8, 2, 4, 1, 6, 2, 9, 7];

function findDuplicates(arr) {

    return arr.filter((currentValue, currentIndex) =>
arr.indexOf(currentValue) !== currentIndex);

}

console.log(findDuplicates(array));
```

3. difference between let and var -

```
function varTest() {

    var x = 1;

    if (true) {

        var x = 2;  // same variable!

        console.log(x);  // 2

    }

    console.log(x);  // 2

}

varTest();
```

=============

```
function letTest() {

    let x = 1;

    if (true) {

        let x = 2;  // different variable

        console.log(x);  // 2

    }

    console.log(x);  // 1

}

letTest();
```

4. Remove duplicates from an array and return unique values.

Ans -

What is the set method used in javascript ?

Ans - set is used to return values in an array and ignore the duplicate values.

```
const ages = [26, 27, 26, 26, 28, 28, 29, 29, 30]

const uniqueAges = [...new Set(ages)]

console.log(uniqueAges) //Array [26, 27, 28, 29, 30]
```

5. What will the following code output?

```
(function() {

  var a = b = 5;

})();

console.log(b);
```

Ans -

The code above will output 5 even though it seems as if the variable was declared within a function and can't be accessed outside of it.

6.write a program to square array elements.

Ans -

```
let arr = [1, 6, 7, 9];

let result = arr.map(x => x ** 2); // ** is es6
exponential

console.log(result);

output - [1,36,49,81]
```

7. checking duplicate value in array -

1st way

```
let array1 = [5,6,7,8,13,8,6];

function dublicateFunction(arg) {

    return arg.filter( (item,i) => arg.indexOf(arg[i])
!== i);

}

console.log(dublicateFunction(array1));// [8,6]
```

8. Explain closure with an example.

displayName doesn't define any local variables,

yet it is able to alert name because name has been defined in the scope in which the closure was created — that of its outer function.

Note - Closures are useful because they let you associate some data with a function that operates on that data.

```
function myName() {

    var name = "Piter Pan"; // outer func. variable
    function displayName() {
        alert (name); // no inner func var but it can
access name;
    }
    function setName(newName) {
        name = newName;
    }
    displayName();
    setName("Peter England");
    displayName();
}
myName();
==================================
calling inner func through outer fun variable
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}
```

```
var myFunc = makeFunc(); // outer func. variable.

myFunc();
```