JavaScript (Yash Chouhan)

RoadMap →

▼ **All About JavaScript**

- **What is JavaScript**
  - It is a programing language that is used to create webpages and make them dynamic.
- **History of JavaScript**
  - JavaScript was introduced in 1995 and since then it's been helping in making websites more functional
  - JavaScript is in all browsers including Chrome, including Firefox, or any other
  - The Browser actually reads html and CSS and JavaScript
- **JavaScript Version**
  - JavaScript was a scripting language and the technical name of it we can say is ECMAScript based on that the abbreviation of that the version are named ES1, ES2…… ES6
  - After ES6 2016 we started naming the as ES6 2017, 2018, 2019….
- **Running JavaScript**
  - JavaScript is run only in the browser we don't have to install any other thing for it as it's already there in our browser
  - Nevertheless, we may have to install another library if we are using that like node, react, and so on.

▼ **Variables in JavaScript**

- Variables are a container where we can store our data and access it later from the storage location.
- We can specify the name of a Variable whichever we like but there are some conditions for it
    - Names can contain letters, digits, underscores, and dollar signs.
    - Names must begin with a letter.
    - Names can also begin with $ and _
    - Names are case-sensitive (y and Y are different variables).
    - Reserved words (like JavaScript keywords) cannot be used as names.
- We should always declare the variables first which we are going to use in our code.
- Scope
    - In JavaScript, variables have some kind of scope.
    - scope describes where we can access a variable or we can say to what extent we can access the variable, if I declare a variable in a function it will only be accessed in that function and not outside of that.

    ▼ **There are a few kinds of Scope**

    - **Block Scope**
        - The variable that is declared inside {} will only be accessed inside that brackets.
        - This was introduced after 2015.
    - **Local Scope**
        - The variable that is declared inside a function will only be accessed inside that function only
    - **Function Scope**
        - The variable that is declared inside a function will only be accessed inside that function only, it is the same as the local function
    - **Global JavaScript Variables**

        - The variable that is declared outside the function will be a global variable we can access it inside the function or outside the function basically anywhere we want.

- The JavaScript variables can be declared in 4 ways
    - **Automatically**
        - It’s easy to declare a variable automatically, for example below

```
x = 5;
y = 6;
z = x + y;
```

    - **Var**
        - To declare the variable using Var we have to add the word before declaring the variable.

```
var x = 5;
var y = 6;
var z = x + y;
```

        - Back when JavaScript was just introduced we mostly used to use Var to declare the variable
        - The Var variable doesn’t support block scope
        - Using var we can redeclare the same variable with the same name
    - **Let**
        - Let and const were two types that were introduced in 2015 we never used them before the reason for having let and const is that they both provide some extra benefits than var

```
let x = 5;
let y = 6;
let z = x + y;
```

        - Some latest versions of Chrome don’t support var so we have to use let and const
        - The variable that is declared using let can be changed later in the code, it doesn’t make the value constant. but we can’t Redeclare the same variable again with the same name. However, if we are using it in different functions that can work.
    - **Const**
        - Const is one of the most important declaration ways in JavaScript because the variable that is declared using const can’t be changed later in the code because the value becomes **constant**

```
const x = 5;
const y = 6;
const z = x + y;
```

- We can declare a variable in any of the declarations.
- Also, we have to use the assigning operator to declare the variable since we are declaring the variable a value
- In JavaScript, we use ‘=’ to assign the variable a value, in JavaScript it’s not an equal too it’s an assignment operator.

> 💡 “Equals to” is used using ‘==’ in JavaScript

▼ Hoisting

- In JavaScript, a variable can be declared after it has been used. In other words; a variable can be used before it has been declared.
- 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 simple words, we can declare a variable after it is used

```
x = 5;
alert(x);
var x;
```

- The above example will show the x value 5 without any error.
- but if we do it like this and use let instead of var.

```
x = 5;
alert(x);
let x;
```

- This will show a Reference error because the variable declaration let and const will are hoisted at the top meaning declared at the top of the code and initialized later
- only Var supports hoisting and let and const don't
- In JavaScript, initializations are not hoisted. For example.

```
console.log(a);
var a = 5;
```

- This will show 'undefined'.
- Also, when the variable is used inside the function, the variable is hoisted only to the top of that function.

> **Note**: In hoisting, though it seems that the declaration has moved up in the program, the actual thing that happens is that the function and variable declarations are added to memory during the compile phase.

▼ Data Type in JavaScript

- The data type is nothing but a defined form or a type in which we will store the variable.
- Data type tells the computer about the type of value which we are going to store in that variable like string, number, boolean, etc
- There are **9 types of Data types** in JavaScript
    1. String
    2. Number
    3. BigInt
    4. Boolean
    5. Undefined
    6. Null
    7. Symbol
    8. Object
    9. Symbol
- Different types of data type store different types of values

Data Types	Description	Example
`String`	represents textual data	`'hello'`, `"hello world!"` etc
`Number`	an integer or a floating-point number	`3`, `3.234`, `3e-2` etc.
`BigInt`	an integer with arbitrary precision	`900719925124740999n` , `1n` etc.
`Boolean`	Any of two values: true or false	`true` and `false`
`undefined`	a data type whose variable is not initialized	`let a;`
`null`	denotes a `null` value	`let a = null;`
`Symbol`	data type whose instances are unique and immutable	`let value = Symbol('hello');`
`Object`	key-value pairs of collection of data	`let student = { };`

- The **Object Datatype**
  - Object
  - Array
  - date
- These are Object data type
- Examples:-

```
// Numbers:
let length = 16;
let weight = 7.5;

// Strings:
let color = "Yellow";
let lastName = "Johnson";

// Booleans
let x = true;
let y = false;

// Creating Object
        const obj = {
            "Yash" : 20,
            "Akash" : 25
        }

// Creating Array
        const arr = [
            "Yash",
            "Akash",
            "Ram",
            "Krshn"
        ];
        console.log(arr);
```

```
// Date object:
const date = new Date("2022-03-25");
```

▼ **Operators in JavaScript**

- Operators in JavaScript are the symbols that we use to assign some value to variables

▼ **Type Casting**

- When we what to convert one data type to another variable then we use type casting.
- There are two types of type casting
  - Implicit
    - It’s also known as Widening
    - This means converting a large data type to a small data type
    - This we can do without losing any data
    - We usually don’t have to do this because it’s done automatically
  - Explicit
    - It’s also known as Narrowing
    - This means converting large data type to small data type
    - This has the risk of losing data because we are converting large data types to small.
    - This we have to do manually by using some inbuild functions on JavaScript.
    - Example 1: Converting number data type to string

```
let a = 1;
        console.log(a);
```

```
        console.log(typeof(a));
        let b = a.toString();
        console.log(b);
        console.log(typeof(b));
```

- Example 2: Converting Float to number

```
let a = 1.11;
        console.log(a);
        console.log(typeof(a));

        let b = parseInt(a);
        console.log(b);
        console.log(typeof(a));
```

▼ Data Structures

- Data Structure is a way to efficiently use data and the memory in the system.
- There are two types of data structures in JavaScript
  - **Primitive**
    - These come with the Language itself meaning they are already there in the language like Array, Objects and all
  - **Non-Primitive**
    - These DS don't come with the language we have to create them by ourselves meaning we have to code them
    - They include
      - Stack
      - Queue
      - Linked List and all

- **Key Collection**
  - key collection is the collection of data collections, mainly there are two map and set and they are iterable in the same order they were inserted
  - Types
    - **Map**
      - Creating a map in JavaSript is easy

```
const testmap = new map();
console.log(testmap);
```

      - This creates a map and returns and empty map value like **Map(0)**
      - To Store Value in the map we do it like this

```
const testmap = new Map([
    ["marks", 200],
    ["Tests", 300]
])
console.log(testmap);
```

      - This will store the value and return like this **Map(2) {'marks' => 200, 'Tests' => 300}**
      - It displays the value in a list kind of form
      - There is another way also to insert data into the map

```
const testmap2 = new Map();
testmap2.set("marks", 200);
testmap2.set("Tests", 300);
console.log(testmap2);
```

      - First, we created a map and then stored value in it using the '`Map.set("key", "value");`' function it's a pre-build function that is used with maps

> When we assign the value to the same key again and again it will replace the old value with the new value always. Example Below

```
const maptest = new Map();
maptest.set("hey", 1);
maptest.set("hey", 2);
maptest.set("hey", 3);
console.log(maptest);
```

- This will return `Map(1) {'hey' => 3}`

- **Get**
  - Get is a function in the map through which we can get the data from the map basically view it, so instead of doing `consloe.log(map_name)` we can use get.

```
const maptest3 = new Map();
maptest3.set("Yash", 23);
maptest3.set("XYZ", 25);
maptest3.set("Akash", 22);
maptest3.get("Yash");
```

  - This will return **23,** The value which is value which is assigned to **Yash** named key.

- There are other functions like this
  - Map.clean();
  - Map.delete();
  - Map.size();
  - Map.has();

- **Json**

- {"name":"Yash", "Work at" : "JT tech"}

- **Indexed Value**
  - It’s a collection of data that are ordered by their index value
  - One of the best examples of this can be Array because they are stored with their index value

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, ini
    <title>Index Value</title>
</head>
<body>
    <h2>Array index value 1</h2>
    <p id="arr1"></p>
    <h2>Array index value 2</h2>
    <p id="arr2"></p>

    <script>
        let arr1store = "";
        let arr2store ="";
        const arr1 = ["Yash", " ", "Chouhan"];
        for(i = 0; i<arr1.length; i++){
            arr1store+=arr1[i];
        }
        document.getElementById("arr1").innerHTML = arr1st

        const arr2 = ["Akash", " ","Kakadiya"];
        for(i = 0; i<arr2.length; i++){
            arr2store+= arr2[i];
        }
        document.getElementById("arr2").innerHTML = arr2st
    </script>
```

```
</body>
</html>
```

- In the above example, there are two arrays and they are being iterated as per their index value their value is stored in a variable ‘arr1store’ and ‘arr2store’ and that’s being displayed in the browser using HTML
- array starts from 0 so ‘Yash’ is on 0 index and ‘Chouhan’ is on 1 index

- **Array**
  - The array is a non-primitive data type that is used to store multiple data into a single variable.
  - The array is one of the most important topics since they clear many other concepts and use them many times.
  - **Typed Array**
    - JavaScript typed arrays are array-like objects that provide a mechanism for reading and writing raw binary data in memory buffers.
    - Basically, they are the array that helps us to manage the buffer time in the array
    - Example: we have 30 data in the array and when we fetch that array it will take some time and buffer a bit so the typed array will allow us to manage that buffer time.

▼ Equality Comparisons

- These are a type of operators that we use to compare values, data, or variables
- there are several types of operators for comparison:

Given that `x = 5`, the table below explains the comparison operators:

Operator	Description	Comparing	Returns	Try it
==	equal to	x == 8	false	Try it »
		x == 5	true	Try it »
		x == "5"	true	Try it »
===	equal value and equal type	x === 5	true	Try it »
		x === "5"	false	Try it »
!=	not equal	x != 8	true	Try it »
!==	not equal value or not equal type	x !== 5	false	Try it »
		x !== "5"	true	Try it »
		x !== 8	true	Try it »
>	greater than	x > 8	false	Try it »
<	less than	x < 8	true	Try it »
>=	greater than or equal to	x >= 8	false	Try it »
<=	less than or equal to	x <= 8	true	Try it »

- There is one more `Object.is` **,** it's a function that will help us to know the exact value of the given data.

```
let objectistest = console.log(Object.is(1, "1"))

// return: false
```

- `Object.is` the return value in the boolean data types
- **==**
  - They are known as LooselyEqual
  - Checks whether its two operands are equal, returning a `Boolean` result. It attempts to convert and compare operands that are of different types.
  - They check if given values are equal or not they don't care about the type if the value is the same they will return true
  - Example: we give 1 and "1" then it will return true because even one is a number and the other in String but it doesn't check the type.
- **===**

- They are known as Strick equality.
- They are just like the LooselyEqual but it’s just that they not only check the value given they also check the type
- Meaning if we give 1 and “1” then it will return false because one is a number and the other is a string so it also checks the type

▼ **Loops in JavaScript**

- Loops offer a quick and easy way to do something repeatedly.
- There are a Few Types of loops
  1. **For loop**
  2. **Do-while loop**
     - The do-while loop is a bit different from the other loops.
     - It first shows the output and then applies the condition
     - Syntex:

```
do {
  // Code to be executed
} while (condition);
```

     - Example:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-widt
    <title>Do-while Loop</title>

    <script>
        checker = () => {
            let number = document.getElementById("te
```

```
            do{
                if(number != 4){
                    alert("Not a Right Number")
                }else{
                    alert("Great!!")
                }
            }while(true);
        }

    </script>
</head>

<body>
    Do-while Loop<br><br>
    <label>Enter 4</label>
    <input id="text" type="text">
    <button onclick="checker()">Check</button>
</body>

</html>
```

- We have used the do while loop to check if the user-entered number is equal to 4 or not.

3. **While loop**

```
while (condition) {
  // code block to be executed
// increment
}
```

- Here we add the condition first and then the code and then the increment.

4. **For-in loop**

- For-in loop is written like

```
for(key in object){
}
```

- Example:-

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    <title>Loops in JS</title>
</head>
<body>
    Loops in JS
    <p id="marks_dis"></p>

    <script>
        let marks_dis = "";
        const marks = {
            "Yash" : 77,
            "Akash" : 75,
            "Poojan" : 78
        }
        for(let i in marks){
            marks_dis += i + ": " + marks[i] + "<br>";
        };
        document.getElementById("marks_dis").innerHTML
    </script>
</body>
</html>
```

- The above example has an object named ‘marks’ and using a for-in loop we iterated over it and fetch the value

5. **For-of loop**

- For-of loop is also almost similar to the for-in loop but there are some differences
- Example:-

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-widt
    <title>For-of Loop</title>
</head>
<body>
    For-in Loop
    <p id="marks_dis"></p>
    For-of Loop
    <p id="marks_dis2"></p>

    <script>
        let marks_dis = "";
        const marks = [54,64,66]
        for(let i of marks){
            marks_dis += i + "<br>";
        }
        document.getElementById("marks_dis").innerH

        let marks_dis2 = "";
        const marks2 = {
            "Yash" : 77,
            "Akash" : 75,
            "Poojan" : 78
        };
        // console.log(marks2);

        for(let i of Object.keys(marks2)){
            marks_dis2 += i +  " " + marks2[i] + "<
```

```
            document.getElementById("marks_dis2").in
        }
    </script>
</body>
</html>
```

- Here we have taken two things, one is an array,, and another is an object.
- In the object we have used a function “key()” which will convert the object keys into the array so that it will fetch the data from the object.

▼ Expressions and Operators

- Expressions and operators are nothing but the expression that when we assign a value to a variable like `a = 4;` that is an expression and the operator used to assign that value to a is `=`
- There are several types of operators
  - **Assignment operators**
    - An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (`=`), which assigns the value of its right operand to its left operand. That is, `x = f()` is an assignment expression that assigns the value of `f()` to `x`.
  - **Comparison operators**
    - Comparison operators are the operators that compare values and return true or false. The operators include: `>`, `<`, `>=`, `<=`, `==`, `===`, `!=` and `!==`
  - **Arithmetic operators**

    The Arithmetic operators perform addition, subtraction, multiplication, division, exponentiation, and remainder operations.

    Arithmetic operators in JavaScript are as follows:

    - `+` (Addition)

- `-` (Subtraction)
- `*` (Multiplication)
- `*` (Exponentiation)
- `/` (Division)
- `%` (Modulus i.e. Remainder)
- `++` (Increment)
- `-` (Decrement)

- **Bitwise operators**
  - Bitwise operators treat arguments as 32-bits (zeros & ones) and work on the level of their binary representation. Ex. Decimal number `9` has a binary representation of `1001`. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

    Bitwise operators in JavaScript are as follows:

    - `&` (AND)
    - `|` (OR)
    - `^` (XOR)
    - `~` (NOT)
    - `<<` (Left SHIFT)
    - `>>` (Right SHIFT)
    - `>>>` (Zero-Fill Right SHIFT)
- **Logical**
  - There are four logical operators in JavaScript: `||` (OR), `&&` (AND), `!` (NOT), `??` (Nullish Coalescing).
- **String operator**
  - String operators are the ones that are used with strings

- Like the `+` operator when used in string, it will concatenate two or more strings
- The shorthand assignment operator `+=` can also be used to concatenate strings.

```
// Example for += operator and how it concatenates s
let str = " My";
str += " Name";
str += " is";
str += " Yash Chouhan";
console.log(str);

// Output:- My Name is Yash Chouhan
```

- **Conditional operator**
  - `variablename = ( condition ) ? value1 : value2`
  - Example: `let voteable = (age < 18) ? "Too young":"Old enough";` Try it Yourself »

```
a = 10;
a = (a < 9) ? "under 9" : "above 9";
// 'above 9'
```

- **Comma operator**
  - You can use the comma operator when you want to include multiple expressions in a location that requires a single expression.
- **Unary operator**
  - Unary operators are:-

Operator	Explanation
Unary plus (`+`)	Tries to convert the operand into a number
Unary negation (`-`)	Tries to convert the operand into a number and negates after

Increment (`++`)	Adds one to its operand
Decrement (`--`)	Decrements by one from its operand
Logical NOT (`!`)	Converts to boolean value then negates it
Bitwise NOT (`~`)	Inverts all the bits in the operand and returns a number
`typeof`	Returns a string which is the type of the operand
`delete`	Deletes specific index of an array or specific property of an object
`void`	Discards a return value of an expression.

- **Relational operator**
  - A comparison operator compares its operands and returns a boolean value based on whether the comparison is true.

  `<` (Less than)

  Less than operator.

  `>` (Greater than)

  Greater than operator.

  `<=`

  Less than or equal operator.

  `>=`

  Greater than or equal operator.

  `instanceof`

  The `instanceof` operator determines whether an object is an instance of another object.

  `in`

  The `in` operator determines whether an object has a given property.

▼ Functions in JavaScript

- Functions exist so we can reuse code. They are blocks of code that execute whenever they are invoked. Each function is typically written to perform a particular task, like an addition function used to find the sum of two or more numbers.

- There are two type of function in JavaScript
  1. **Normal function**
     a. Syntex:

```
Function function_name(parameter){
            // functions work
}
```

     b. Example:-

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
  <title>JavaScript Function Example</title>
</head>
<body>
  <script>
    function calculateRectangleArea(length, width) {
      var area = length * width;
      return area;
    }

    var length = 5;
    var width = 3;
    var rectangleArea = calculateRectangleArea(lengt
    console.log("Area of the rectangle: " + rectangl
  </script>
</body>
</html>
```

  2. **Arrow function**

a. Arrow functions are the functions which are a bit different then the normal functions

b. Syntex:-

```
function_name = () =>{
        // code
}
```

c. Example:-

```
calculateRectangleArea = (length, width) => {
      var area = length * width;
      return area;
    }

    var length = 5;
    var width = 3;
    var rectangleArea = calculateRectangleArea(lengt
    console.log("Area of the rectangle: " + rectangl
```

d. Just change the function in the previous code

> 💡 Use of `new` keyword:
>
> - Normal Function: Normal functions can be used as constructors with the `new` keyword to create new objects.
> - Arrow Function: Arrow functions cannot be used as constructors. They do not have their own `this` binding, which is necessary for creating new objects.

- There are few differences between both the types of functions.
  - It's important to note that arrow functions are not a complete replacement for normal functions. They have some limitations and differences, particularly related to the `this` keyword and function behavior. The choice

between normal functions and arrow functions depends on the specific use case and the desired behavior.

- **Arguments object.**
- The arguments object is an Array-like object accessible inside functions that contains the values of the arguments passed to that function, available within all non-arrow functions.

```
function test(a,b,c){
        console.log(arguments[0]);
        console.log(arguments[1]);
        console.log(arguments[2]);
    }
    test(1,4,3)
```

- **Function Stack / Call Stack**
  - Function stack or Call Stack is like when a function is running and there is another function in it so that will also run so the function stack is a function stack, it keeps records of the functions that are being called and running right now

    A **call stack** is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc.

    - When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.
    - Any functions that are called by that function are added to the call stack further up, and run where their calls are reached.
    - When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing.
    - If the stack takes up more space than it was assigned, a "stack overflow" error is thrown.

```
function greeting() {
  // [1] Some code here
  sayHi();
  // [2] Some code here
}
function sayHi() {
  return "Hi!";
}

// Invoke the `greeting` function
greeting();

// [3] Some code here
```

- Visit To understand how it will execute *https://developer.mozilla.org/en-US/docs/Glossary/Call_stack*

- In summary, then, we start with an empty Call Stack. Whenever we invoke a function, it is automatically added to the Call Stack. Once the function has executed all of its code, it is automatically removed from the Call Stack. Ultimately, the Stack is empty again.
- **Recursion**
  - One of the **most powerful and elegant concept** of functions, recursion is when a function invokes itself.
  - Example:-

```
function pow(x, n) {
  return (n == 1) ? x : (x * pow(x, n - 1));
}
alert(pow(2,3));
```

  - This will return `8` because the function is calling itself until the n is equal to 1.

- Here is the break down of how this function will run again and again and call itself
    - First the value of x is 2 and n is 3

```
2 * pow(2, 3-1);
2 * pow(2, 2-1)
2 * pow(2, 1-1);
// this will run because till now the n was 3 the
```

- **Lexical scoping**
    - Lexical scoping is a concept that is related to the source code and JavaScript is a ***lexically-scoped***

```
var a = 'static';

function f1() {
   console.log(a);
}

function f2() {
   var a = 'dynamic';
   f1();
}

f2();
```

    - Before reading further think about what this code will return
    - While running this code it will print ‘static’ and not ‘dynamic’
    - Why? Because as said before JavaScript is lexically scoped meaning that when we call a variable first it searches it in the global environment and then goes to the lexical environment which can be inside a functions
-

▼ This Keyword In JavaScript

- In JavaScript, the `this` keyword refers to an **object**. **Which** object depends on how `this` is being invoked (used or called).
- This keyword refers to an object which is executing the current piece of code.
- When we use “this” keyword inside and function it will refer to the global scope.

```
<script>
        var a = 7;
        function test(){
            var a = 8;
            alert(a);
            alert(this.a);
        }
        test();
    </script>
```

  - The Expected output of this would be 8 and then 7, we called the same variable but when we called it normally it showed a local variable value but when we called it using the ‘this’ keyword then it showed a global variable value,
  - conclusion when we use “this” in a function it will always refer to the global scope and not local.

- And when we use “this” keyword inside an object then it will refer to the local scope.

```
const obj = {
            name : "Yash",
            age : "21",
            hey(){
                console.log("age is " + this.age);
            }
        }
        obj.hey();
```

- The expected output of this would be 21 because as said before when we use the “this” keyword in the object it will always refer to the local scope.

▼ **Asynchronous JavaScript**

- Synchronous means the code runs in a particular sequence of instructions given in the program, whereas asynchronous code execution allows to execution of the upcoming instructions immediately. Because of asynchronous programming, we can avoid the blocking of tasks due to the previous instructions.
- In a synchronous environment, a function will run only after the before function is finished and in asynchronous this will be different because asynchronous can run more than one function or task at a time.

```
function test(){
                console.log("this is test function");
}

function test1(){
                test()
                console.log("This is test1 function");
}
```

  - Now here the first function is ‘test’ that will run first and only after that is done running the other function will run which is ‘test1’, this is what makes it slow, we won’t see this on a small scale but imagine having more than 100 functions in a code that will surely make it slow.
  - However, this won’t happen in the Asynchronous environment because in that we can run multiple functions at a time meaning that we can do more than one task at a time.
- Examples of synchronous and asynchronous functions.
  - Synchronous function:

```
function syncFunction() {
  const result = performTask(); // Blocking operation
```

```
  return result;
}
```

- Asynchronous Functions:

```
async function asyncFunction() {
  try {
    const result = await performAsyncTask(); // Non-blo
    return result;
  } catch (error) {
    // Handle errors asynchronously
    console.error('An error occurred:', error);
    throw error;
  }
}
```

- The main difference here is using the keywords which are ‘async’ and ‘await’.
- The key differences between synchronous and asynchronous functions lie in their execution flow and blocking behavior. Synchronous functions execute code sequentially and block other code, while asynchronous functions allow concurrent execution and do not block the execution of other code. Asynchronous functions leverage mechanisms like promises, callbacks, or `async/await` syntax to handle the results of asynchronous operations.

▼ **API in JavaScript**

- API stands for the **Application programming interface**
- Basically, API is that when we use data from another server we have to add API, for example, I want to integrate a location field in my website which will tell users about the distance from one place to another and that I can find using google maps so fortunately google maps provide it’s API which I can add in my code through which I can assess the data of my need and display or use in my website.

- There are two functions that we can use while working with APIs
  - **XMLHttpRequest()**
    - it's a way to fetch data from the server without reloading the page completely meaning a part of the page will reload and give the required information and not the whole page
    - This is usually what we must see in Trading sites and apps because there the numbers keep updating at the current time without the whole page being reloaded.
    - This is done with the help of AJAX **(Asynchronous JavaScript And XML)**
  - **fetch()**
    - The fetch() method in JavaScript is used to request to the server and load the information on the web pages. The request can be of any APIs that return the data of the format JSON or XML. This method returns a promise.

▼ **Classes in JavaScript**

- Class is a collection of methods and functions basically
- Syntex:

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

- We can create an object of a class like this.

```
const class_name = new object_name();
```

▼ Javascript Iterators.

- In JavaScript, there are some ways to iterate over something like an array, object, or something else
- one of the powerful ways out of loops is the ‘for of’ loop, when iterating over an array it makes working much easy and more efficient.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, in
    <title>For of Loop</title>

    <script>
        let a = ['a','b','c'];
        for(arr of a){
            console.log(arr);
        }
    </script>
</head>
<body>
</body>
</html>
```

  - Imagine doing this with for loop would be a bit tricky and a bit longer too but a ‘for of’ loop made it much easier. we just created the array and then in the loop created a variable ‘arr’ and we iterated it over the array ‘a’ and printed the ‘arr’ variable values in the console.

▼ Module in JavaScript

- A module is just a file. One script is one module. As simple as that.
- Modules can load each other and use special directives `export` and `import` to interchange functionality, and call functions of one module from another one:

- The module helps us to make the code much clearer and more understandable
- By making modules of some code or functions we can reuse it in a much better way.

JavaScript Practice