**What is JavaScript?**

JavaScript is a high-level, interpreted programming language primarily used for building **dynamic and interactive web applications**.

It is one of the core technologies of the World Wide Web and runs on the **client-side** (in web browsers) as well as on the **server-side** (with the help of frameworks like Node.js).

Here are some key features and aspects of JavaScript:

1. **Scripting Language:** JavaScript is often referred to as a scripting language because it is executed in real-time as the program runs. It doesn't need to be compiled before running, unlike languages like Java or C++.
2. **Object-Oriented:** JavaScript is based on an object-oriented programming (OOP) paradigm. It supports objects, classes, inheritance, and other OOP principles.
3. **Dynamic Typing:** JavaScript uses dynamic typing, meaning you don't need to explicitly declare variable types. Variables can hold values of any type, and their type can change during runtime.
4. **Event-driven and Asynchronous:** JavaScript is event-driven, allowing you to respond to user actions or system events. It also supports asynchronous programming with features like callbacks, promises, and async/await, enabling non-blocking and efficient execution of code.
5. **DOM Manipulation:** JavaScript is commonly used to interact with the Document Object Model (DOM) of web pages. It allows you to manipulate the structure, content, and styling of HTML elements dynamically.
6. **Libraries and Frameworks:** JavaScript has a vast ecosystem of libraries and frameworks that extend its capabilities and simplify development. Examples include React, Angular, Vue.js, and many others.
7. **Cross-platform:** JavaScript can run on multiple platforms, including web browsers, servers, mobile devices, and even IoT devices.

**What are the different data types present in JavaScript?**

To know the type of a JavaScript variable, we can use the **typeof**operator.

* **Primitive data types** can store only a single value. To store multiple and complex values, **non-primitive data** types are used.

#### **Note- It is important to remember that any data type that is not a primitive data type, is of Object type in JavaScript.**

**Data Types in JavaScript:**

JavaScript has seven types.

1. Number
2. String
3. Boolean
4. Undefined
5. Null
6. Symbol
7. Object

The Primitive Data types in JavaScript include Number, String, Boolean, Undefined, Null and Symbol.

The Non-Primitive data type has only one member i.e. the Object.

### Explain Hoisting in JavaScript.

Hoisting is the default behaviour of JavaScript where all the variables and function declarations are moved on top.

***Note - To avoid hoisting, you can run JavaScript in strict mode by using “use strict” on top of the code.***

### Why do we use the word “debugger” in JavaScript?

**Using the debugger statement:** This statement can be inserted into your code at a specific point where you want to pause execution and inspect the state of your code. When the debugger statement is encountered, the JavaScript interpreter will pause execution, and you can then use the browser's debugging tools to inspect variables, call stacks, and more. The remaining section of the code should stop execution before moving on to the next line while debugging.

### Scopes in JavaScript:

1. **Global Scope:**
   * Variables declared outside of any function or block have global scope.
   * They can be accessed from anywhere in the code, both inside and outside functions.
   * Example:

javascriptCopy code

var globalVariable = "I am global"; function exampleFunction() { console.log(globalVariable); // Accessing the global variable }

1. **Local (Function) Scope:**
   * Variables declared inside a function have local scope, meaning they are only accessible within that function.
   * Example:

javascriptCopy code

function exampleFunction() { var localVariable = "I am local"; console.log(localVariable); // Accessing the local variable } // Trying to access localVariable here would result in an error

1. **Block Scope (Introduced in ES6):**
   * With the introduction of **let** and **const** in ES6, JavaScript now has block-scoped variables.
   * Variables declared using **let** and **const** are scoped to the nearest enclosing block (e.g., if statements, loops).
   * Example:

javascriptCopy code

if (true) { let blockScopedVariable = "I am block-scoped"; console.log(blockScopedVariable); } // blockScopedVariable is not accessible here

1. **Lexical Scope (Closures):**
   * Lexical scope refers to the ability of a function to access variables from its outer (enclosing) scope, even after the outer function has finished execution.
   * This concept is often seen in closures, where an inner function has access to the variables of its outer function.
   * Example:

javascriptCopy code

function outerFunction() { var outerVariable = "I am outer"; function innerFunction() { console.log(outerVariable); // Accessing outerVariable from the outer scope } return innerFunction; } var closure = outerFunction(); closure(); // This will log "I am outer"

### Difference between var, let and const keyword in JavaScript.

‘**var**’ can be redefined and even redeclared anywhere throughout its scope.

‘**let**’ can be redefined within its scope but cannot be re-declared within its scope.

‘**const**’ cannot be redefined or re-declared within its scope.

**Explain Implicit Type Coercion in JavaScript.**

Implicit type coercion in JavaScript is the automatic conversion of value from one data type to another. It takes place when the operands of an expression are of different data types.

* **String coercion**

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

Example 1:

**var** x = 3;

**var** y = "3";

x + y // Returns "33"

* **Boolean Coercion**
* **Logical operators**
* **Equality Coercion**

### Is JavaScript a statically typed or a dynamically typed language?

JavaScript is a dynamically typed language. In a dynamically typed language, the type of a variable is checked during **run-time**in contrast to a statically typed language, where the type of a variable is checked during **compile-time.**

### What is NaN property in JavaScript?

NaN property represents the **“Not-a-Number”**value. It indicates a value that is not a legal number.

console.log(isNaN(123)); // false, because it’s a valid number.

**typeof**of NaN will return a **Number**.

To check if a value is NaN, we use the **isNaN()**function.

### Explain passed by value and passed by reference.

**In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.**

**In pass-by value in JavaScript, a copy of the original variable is created, so any changes made to the copied variable do not affect the original variable.**

**In pass-by reference in JavaScript, we pass the reference of the actual parameter. No copy is created in the memory.**

**Pass by Value:**

let num1 = 5;

function changeNumber(num) {

num = 10;

document.write(num); // output: 10

}

document.write(num1); // output: 5

**Pass by Reference:**

let arr1 = [1, 2, 3];

function changeArray(arr) {

arr.push(4);

document.write(arr); // output: [1, 2, 3, 4]

}

document.write(arr1); // output: [1, 2, 3, 4]

### What is an Immediately or Self-Invoked Function in JavaScript?

**An Immediately Invoked Function (known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.**

Without being requested, a **self-invoking** expression is automatically invoked (initiated). If a function expression is followed by (), it will execute automatically. A function declaration cannot be invoked by itself.

(function () {  
  // I will invoke myself  
})();

**Examples:**

// Regular Function

    function Greet() {

        console.log("Hello!");

    };

    Greet();

// IIFE creation and execution.

    (function() {

        console.log("Hello!");

    })();

Note: The above function is called as an **anonymous function**, a function w/o name.

**11. What do you mean by strict mode in JavaScript and characteristics of JavaScript strict-mode?**

In ECMAScript 5, a new feature called JavaScript **Strict Mode allows you to write a code or a function in a "strict" operational environment**. In most cases, this language is 'not particularly severe' when it comes to throwing errors. In 'Strict mode,' however, all forms of errors, including silent errors, will be thrown. As a result, debugging becomes a lot simpler. Thus, programmer's chances of making an error are lowered.

Characteristics of strict mode in JavaScript:

1. Duplicate arguments are not allowed by developers.
2. Global variables are not allowed.
3. In strict mode, you won't be able to use the JavaScript keyword as a parameter or function name.
4. The **'use strict'** keyword is used to define strict mode at the start of the script. Strict mode is supported by all browsers.

**12. Explain Higher Order Functions in JavaScript.**

**Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.**

map() , reduce() , filter() , and sort() – 4 higher order functions using arrays.

**Normal Function:**

const arr1 = [1, 2, 3];

const arr2 = arr1.map(function(item) {  
 return item \* 2;  
});

console.log(arr2);

**Arrow function:**

const arr1 = [1, 2, 3];

const arr2 = arr1.map(i => i \* 2);

console.log(arr2);

### 13. Explain “this” keyword.

**This” keyword refers to an object that is executing the current piece of code. It references the object that is executing the current function.**

**If the function being referenced is a**[**regular function**](https://www.simplilearn.com/tutorials/javascript-tutorial/javascript-functions)**, “this” references the global object.**

**If the function that is being referenced is a method in an**[**object**](https://www.simplilearn.com/tutorials/javascript-tutorial/javascript-objects)**, “this” references the object itself.**

const person = {

firstName: "John",

lastName: "Doe",

fullName: function() {

return this.firstName + " " + this.lastName;

}

};

console.log(person.fullName()); // output: "John Doe"

**15. Explain call(), apply() and, bind() methods.**

**In JavaScript, call, apply, and bind are methods that are used to manipulate the this value of a function. They are often used in the context of function invocation. Here's a brief explanation of each:**

**call method:**

**Syntax: function.call(thisArg, arg1, arg2, ...)**

**The call method is used to invoke a function with a specified this value and individual arguments passed as separate parameters.**

**Example:**

function sayHello(name) {

console.log(`Hello, ${name}!`);

}

sayHello.call(null, 'John');

**In this example, call is used to invoke the sayHello function with the this value set to null and the argument 'John'.**

**apply method:**

**Syntax: function.apply(thisArg, [argsArray])**

**Similar to call, the apply method is used to invoke a function with a specified this value. However, the arguments are passed as an array.**

**Example:**

function sayHello(name) {

console.log(`Hello, ${name}!`);

}

sayHello.apply(null, ['John']);

**In this example, apply is used to invoke the sayHello function with the this value set to null and the argument 'John' passed as an array.**

**bind method:**

**Syntax: function.bind(thisArg, arg1, arg2, ...)**

**The bind method is used to create a new function with a specified this value and initial arguments. However, unlike call and apply, bind doesn't immediately invoke the function; instead, it returns a new function that, when called, has the this value and arguments pre-specified.**

**Example:**

function sayHello(name) {

console.log(`Hello, ${name}!`);

}

const sayHelloToJohn = sayHello.bind(null, 'John');

sayHelloToJohn();

**In this example, bind is used to create a new function sayHelloToJohn with the this value set to null and the argument 'John'. The new function is then invoked.**

**In summary:**

**call and apply are used to invoke a function immediately with a specified this value and arguments.**

**bind is used to create a new function with a specified this value and initial arguments, without immediately invoking the function.**

**16. What is the difference between exec () and test () methods in JavaScript?**

* **test ()** and **exec ()** are RegExp expression methods used in JavaScript.
* We'll use **exec ()** to search a string for a specific pattern, and if it finds it, it'll return the pattern directly; else, it'll return an 'empty' result.
* We will use a**test ()** to find a string for a specific pattern. It will return the Boolean value 'true' on finding the given text otherwise, it will return 'false'.

17**. What is currying in JavaScript?**

Currying is a technique in functional programming that involves transforming a function that takes multiple arguments into a series of functions, that each take a single argument. This allows you to partially apply arguments to a function, which can be useful for creating reusable functions and for creating functions with more specific behaviour.

In JavaScript, you can implement currying using closures. Here's an example of how you can curry a function that takes two arguments:

function simpleFunction(param1, param2, param3, .....) => function

curriedFunction(param1)(param2)(param3)(....

**Normal Function:**

 function calculateVolume(length, breadth, height)

{

        return length \* breadth \* height;

    }

    console.log(calculateVolume(4, 5, 6));

**Curried Function:**

    function calculateVolume(length) {

        return function (breadth) {

            return function (height) {

                return length \* breadth \* height;

            }

        }

    }

    console.log(calculateVolume(4)(5)(6));

For Example, if we have a function **f(a,b)**, then the function after currying, will be transformed to **f(a)(b).**

**18. What are some advantages of using External JavaScript?**

Some advantages of external JavaScript are

1. It allows web designers and developers to collaborate on HTML and JavaScript files.
2. We can reuse the code.
3. Code readability is simple in external JavaScript.

**19. Explain Scope and Scope Chain in JavaScript.**

Scope in JS determines the accessibility of variables and functions at various parts of one’s code.  
In general terms, the scope will let us know at a given part of code, what are variables and functions we can or cannot access.  
  
There are three types of scopes in JS:

* Global Scope
* Local or Function Scope
* Block Scope

**Global Scope:** Variables or functions declared in the global namespace have global scope, which means all the variables and functions having global scope can be accessed from anywhere inside the code.

let i = 0;

function incrementCounter() {

i++;

document.write(i);

}

incrementCounter(); // Output: 1

incrementCounter(); // Output: 2

incrementCounter(); // Output: 3

**Function Scope:**Any variables or functions declared inside a function have local/function scope, which means that all the variables and functions declared inside a function, can be accessed from within the function and not outside of it.

function outerFunction() {

const outerVariable = 'Hello, ';

function innerFunction(name) {

const innerVariable = 'world!';

console.log(outerVariable + name + ' ' + innerVariable);

}

innerFunction('John'); // Output: Hello, John world!

}

outerFunction();

**Block Scope:**Block scope is related to the variables declared using let and const. Variables declared with var do not have block scope. Block scope tells us that any variable declared inside a block { }, can be accessed only inside that block and cannot be accessed outside of it.

function blockScope() {

for (let i = 0; i < 5; i++) {

console.log(i); // Output: 0, 1, 2, 3, 4

}

console.log(i);

// Output: Uncaught ReferenceError: i is not defined

}

blockScope();

### 20. Explain Closures in JavaScript.

Closures are an ability of **a function to remember the variables and functions** that are declared in its outer scope.

function createMultiplier(a) {

return function(b) {

return a \* b;

}

}

const double = createMultiplier(2);

const triple = createMultiplier(3);

console.log(double(5)); // Output: 10

console.log(triple(5)); // Output: 15

**21. Mention some advantages of JavaScript.**

There are many advantages of JavaScript. Some of them are

1. JavaScript is executed on the client-side as well as server-side also. There are a variety of Frontend Frameworks that you may study and utilize. However, if you want to use JavaScript on the backend, you'll need to learn NodeJS. It is currently the only JavaScript framework that may be used on the backend.
2. JavaScript is a simple language to learn.
3. Web pages now have more functionality because of JavaScript.
4. To the end-user, JavaScript is quite quick.

**22. What are object prototypes?**

**A prototype is a blueprint of an object.** The prototype allows us to use properties and methods on an object even if the properties and methods do not exist on the current object.

**All JavaScript objects inherit properties from a prototype.** For example,

* Date objects inherit properties from the Date prototype
* Math objects inherit properties from the Math prototype
* Array objects inherit properties from the Array prototype.

### 23. What are callbacks?

A callback is a function that will be executed after another function gets executed. In JavaScript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function, and can be used as a property of an object.

function print(callback) {

callback();

}

**Functions that are used as an argument to another function are called callback functions.**

function greet(name, callback) {

console.log(`Hello, ${name}!`);// not mandatory

callback();

}

function sayGoodbye() {

console.log('Goodbye!');

}

greet('Alice', sayGoodbye);

### 24. What are the types of errors in JavaScript?

There are two types of errors in JavaScript.

1. **Syntax error**: Syntax errors are mistakes or spelling problems in the code that cause the program to not execute at all or to stop running halfway through. Error messages are usually supplied as well.
2. **Logical error**: Reasoning mistakes occur when the syntax is proper but the logic or program is incorrect. The application executes without problems in this case. However, the output findings are inaccurate. These are sometimes more difficult to correct than syntax issues since these applications do not display error signals for logic faults.

**25. What is memoization?**

Memoization is ***a form of caching*** where the ***return value of a function is cached*** based on its parameters. If the parameter of that function is not changed, the cached version of the function is returned.

*Ex: Closure and Higher order function*

*Note- Although using memoization saves time, it results in larger consumption of memory since we are storing all the computed results.*

**25. What is Recursion?**

Recursion is the ***process of repeating items*** in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

function recurse() {  
 // function code  
 recurse();  
 // function code  
}  
  
recurse();

Recursive functions are very useful to solve many mathematical problems.

**Example:**

// program to count down numbers to 1

function **countDown**(number) {

console.log(number); // display the number

const newNumber = number - 1; // decrease the number value

// base case

if (newNumber > 0) {

**countDown**(newNumber);

}

}

countDown(9);

**27. What is the use of a constructor function in JavaScript?**

Constructor functions are used to create objects in JavaScript.

When do we use constructor functions?

If we want ***to create multiple objects having similar properties and methods,*** constructor functions are used.

#### **Note- The name of a constructor function should always be written in Pascal Notation: every word should start with a capital letter.**

Example:

function person(name,age,gender){

this.name = name;

this.age = age;

this.gender = gender;

}

var person1 = new person("Vivek", 76, "male");

console.log(person1);

var person2 = new person("Courtney", 34, "female");

console.log(person2);

### 29. Which method is used to retrieve a character from a certain index?

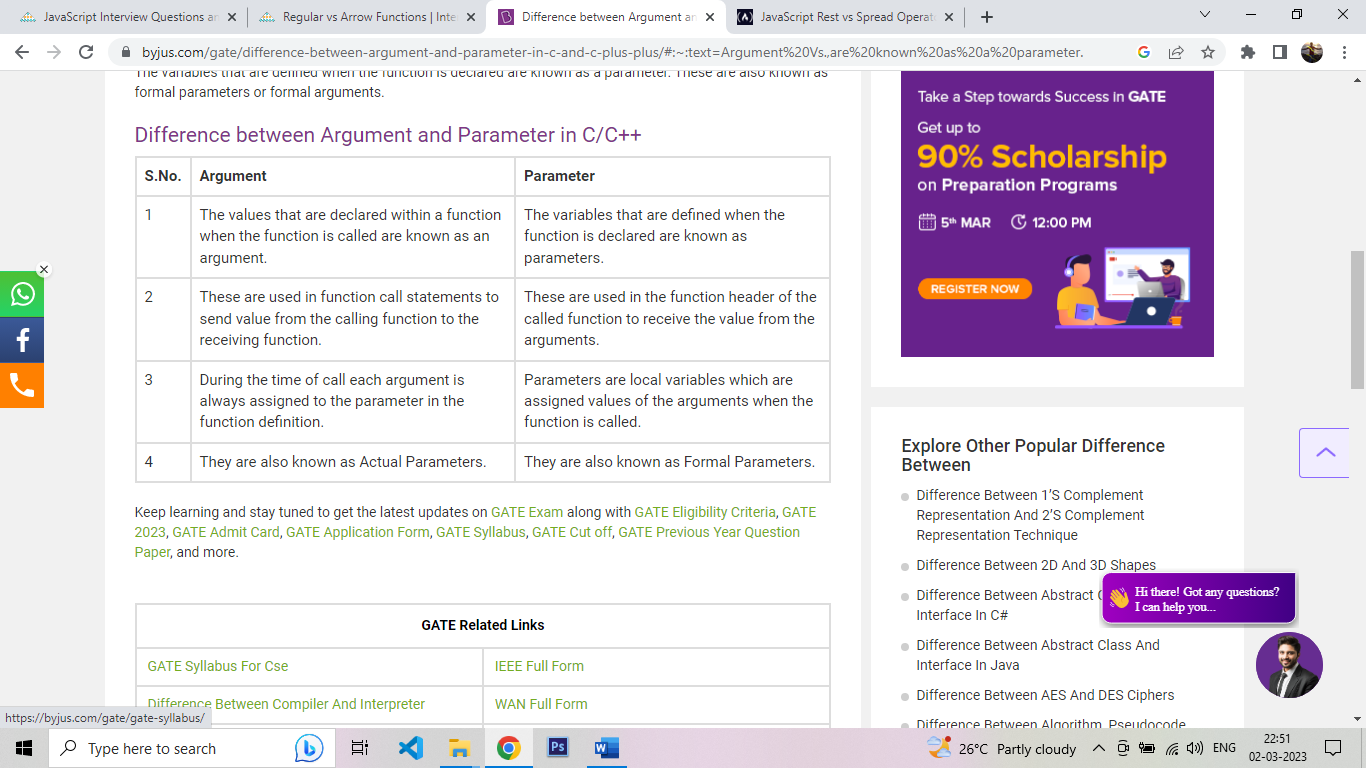
The **charAt()** function of the JavaScript string finds a char element at the supplied index. The index number begins at 0 and continues up to n-1, Here n is the string length. The index value must be positive, higher than, or the same as the string length.

### 31. What is the distinction between client-side and server-side JavaScript?

Client-side means that the processing takes place on the user's computer. It requires browsers to run the scripts on the client machine without involving any processing on the server.

Server-side means that the processing takes place on a web server.

### 32. Difference between Argument (Actual values) and Parameter (Variables)



function addNumbers(x, y) {

return x + y; //x,y - parameters

}

const result = addNumbers(3, 5); //3,5 - arguments

### 32. What are arrow functions?

Arrow functions were introduced in the ES6 version of JavaScript. They provide us with a **new and shorter syntax for declaring functions**. Arrow functions can only be used as a function expression.  
  
Let’s compare the normal function declaration and the arrow function declaration in detail:

// Traditional Function Expression

**var** add = **function**(a,b){

**return** a + b;

}

// Arrow Function Expression

const add = (a, b) => {

return a + b;

};

console.log(add(2, 3));

**console.log(add(2, 3));**

Arrow functions are declared without the function keyword. If there is only one returning expression then we don’t need to use the return keyword as well in an arrow function as shown in the example above. Also, for functions having just one line of code, curly braces { } can be omitted.

Regular Function

function helloName(name) {  
    return "Hello " + name;  
}

Arrow function

const helloName = (name) => "Hello " + name;

### 35. What is the rest parameter and spread operator?

Both rest parameter and spread operator were introduced in the ES6 version of JavaScript.  
  
Spread syntax looks exactly like rest syntax. In a way, spread syntax is the opposite of rest syntax.

**Spread syntax** "expands" an array into its elements.

**Rest syntax** collects multiple elements and "condenses" them into a single element.

**Spread Parameter:**

const q1 = ["Jan", "Feb", "Mar"];

const q2 = ["Apr", "May", "Jun"];

const q3 = ["Jul", "Aug", "Sep"];

const q4 = ["Oct", "Nov", "May"];

const year = [...q1, ...q2, ...q3, ...q4];

document.getElementById("demo").innerHTML = year;

**Rest Parameter:**

var colors = ["Violet", "Indigo", "Blue", "Green", "Yellow", "Orange", "Red"];

// destructuring assignment

var [a,b,...args] = colors;

console.log(a);  //  Violet

console.log(b);   //  Indigo

console.log(args);  // [Blue, Green, Yellow, Yellow, Red]

### 36. In JavaScript, how many different methods can you make an object?

In JavaScript, there are several ways to declare or construct an object.

1. Object.
2. using Class.
3. create Method.
4. Object Literals.
5. using Function.
6. Object Constructor.

**37. Callback Function:**

*"I will call back later!"*

A callback is a function passed as an argument to another function.

This technique allows a function to call another function.

A callback function can run after another function has finished.

# **38. JavaScript Asynchronous:**

*"I will finish later!"*

Functions running in parallel with other functions are called asynchronous

A good example is JavaScript setTimeout().

function myFunction(value) {

document.write(value);

}

setTimeout(function() {

myFunction("Hi");

}, 3000);

# **39. JavaScript Promises:**

*"I Promise a Result!"*

A promise is a JavaScript object that **represents the success or failure of an asynchronous operation and its resulting value**. It is a way to handle asynchronous code more elegantly and avoid the "callback hell" (nested callbacks) that can occur with deeply nested asynchronous operations. Promises provide a cleaner and more readable syntax for working with asynchronous code.

**Callback hell**, refers to a situation in programming where multiple nested callbacks are used within asynchronous code, making the code difficult to read, understand, and maintain. This occurs when you have several layers of nested callbacks, especially in scenarios involving multiple asynchronous operations or events.

A promise can be in one of **three states**:

**Pending**: The initial state. The promise is neither fulfilled nor rejected.

**Fulfilled**: The operation completed successfully, and the promise has a resulting value.

**Rejected**: The operation failed, and the promise has a reason for the failure.

The syntax for creating a promise is as follows:

const myPromise = new Promise((resolve, reject) => {

*// Asynchronous operation*

if (*/\* operation successful \*/*) {

resolve('Operation successful'); *// Fulfilled*

} else {

reject('Operation failed'); *// Rejected*

}

});

Once a promise is created, you can use the then and catch methods to handle the fulfilled and rejected states, respectively:

myPromise

.then((result) => {

console.log(result); *// Operation successful*

})

.catch((error) => {

console.error(error); *// Operation failed*

});

Promises are commonly used in scenarios such as:

**Fetching Data**: When making HTTP requests or fetching data from APIs.

**Timeouts and Intervals**: Managing asynchronous operations with timeouts and intervals.

**File Operations**: Reading or writing files asynchronously.

**Database Operations**: Performing asynchronous database queries.

**Event Handling**: Handling events that may or may not have occurred yet.

# **JavaScript Async**

*"async and await make promises easier to write"*

**async** makes a function return a Promise.

**await** makes a function wait for a Promise.

**Async Example:**

async function myFunction() {  
  return "Hello";  
}  
myFunction().then(  
  function(value) {myDisplayer(value);},  
  function(error) {myDisplayer(error);}  
);

**Await Example:**

The await keyword can only be used inside an async function.

The await keyword makes the function pause the execution and wait for a resolved promise before it continues:

async function myDisplay() {  
  let myPromise = new Promise(function(resolve, reject) {  
    resolve("I love You !!");  
  });  
  document.getElementById("demo").innerHTML = await myPromise;  
}  
  
myDisplay();

### 38. What are classes in JavaScript?

Classes are a template/blueprint for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on prototypes.

Unlike functions, classes are not hoisted. A class cannot be used before it is declared.

A class can inherit properties and methods from other classes by using the extend keyword.

All the syntaxes inside the class must follow the strict mode (‘use strict’) of JavaScript. An error will be thrown if the strict mode rules are not followed.

**Class Declaration**

To declare a class we can use a special keyword class followed by class name and then pair of curly braces which wraps the body of the class.

class Student{  
//Body of class  
}

One can also declare a class as:

const Student = class {  
 // Body of class  
};

**Instance of a class**

An instance is an object containing data and behaviour described by the class.

We can create an instance of a class (initiate the class), by using new keyword.

var student1 = new Student(); //student1 is now an instance of class Student

We can initialise the data that an instance may contain by defining a constructor function in the class body which is called whenever a class instance is created.

class Student{  
 constructor(name, age) {  
 this.name = name,  
 this.age = age  
 }  
}  
  
const student1 = new Student("max", 20)  
console.log(student1.name) //prints max  
console.log(student1.age) //prints 20

**static**

The static keyword defines a static method or property for a class. Neither static methods nor static properties can be called on instances of the class. Instead, they’re called on the class itself. Static methods are often utility functions.

We dont need to create any instance to call or access the static members.

class A{  
    static propA = "propA";  
    static methodA() {  
        console.log("methodA");  
    }  
}  
  
console.log(A.propA); //prints "propA"  
A.methodA();  //prints "methodA"  
const a1 = new A(); //creating an instance of class A  
console.log(a1.propA); //prints undefined (propA is not a member of instance a1)  
a1.methodA(); //gives an error

### 40. Explain WeakMap and WeakSet in JavaScript.

[WeakMap](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap) is Map-like collection that allows only objects as keys and removes them together with associated value once they become inaccessible by other means.

[WeakSet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet) is Set-like collection that stores only objects and removes them once they become inaccessible by other means.

Their main advantages are that they have weak reference to objects, so they can easily be removed by garbage collector.

[WeakMap](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap) and [WeakSet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakSet) are used as “secondary” data structures in addition to the “primary” object storage. Once the object is removed from the primary storage, if it is only found as the key of WeakMap or in a WeakSet, it will be cleaned up automatically.

### 43. What is Object Destructuring?

Object destructuring is a new way to extract elements from an object or an array.

* **Object destructuring:** Before ES6 version:

**const** classDetails = {

strength: 78,

benches: 39,

blackBoard:1

}

**const** classStrength = classDetails.strength;

**const** classBenches = classDetails.benches;

**const** classBlackBoard = classDetails.blackBoard;

The same example using object destructuring:

**const** classDetails = {

strength: 78,

benches: 39,

blackBoard:1

}

**const** {strength:classStrength, benches:classBenches, blackBoard:classBlackBoard} = classDetails;

console.log(classStrength); // Outputs 78

console.log(classBenches); // Outputs 39

console.log(classBlackBoard); // Outputs 1

* **Array destructuring:**Before ES6 version:

**const** arr = [1, 2, 3, 4];

**const** first = arr[0];

**const** second = arr[1];

**const** third = arr[2];

**const** fourth = arr[3];

The same example using object destructuring:

**const** arr = [1, 2, 3, 4];

**const** [first,second,third,fourth] = arr;

console.log(first); // Outputs 1

console.log(second); // Outputs 2

console.log(third); // Outputs 3

console.log(fourth); // Outputs 4

### 45. What is a Temporal Dead Zone?

Temporal Dead Zone is a behaviour that occurs with variables declared using **let**and **const**keywords. It is a behaviour where we try to access a variable before it is initialized. Examples of temporal dead zone:

x = 23; // Gives reference error

**let** x;

**function** **anotherRandomFunc**(){

message = "Hello"; // Throws a reference error

**let** message;

}

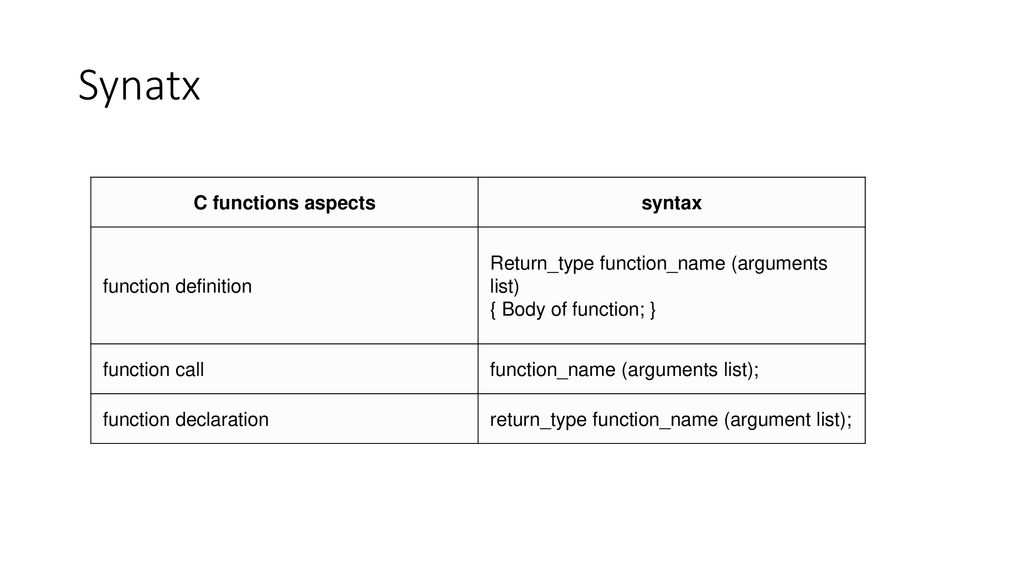
anotherRandomFunc();

In the code above, both in the global scope and functional scope, we are trying to access variables that have not been declared yet. This is called the **Temporal Dead Zone**.

### 46. Function Definition vs Function Declaration:

A function declaration tells the compiler about a function's name, return type, and parameters.

A function definition provides the actual body of the function.



**28. What is DOM?**

The Document Object Model is a programming interface for HTML and XML (Extensible markup language) documents.

DOM is a way to represent the webpage in a structured hierarchical way so that it will become easier for programmers and users to glide through the document. With DOM, we can easily access and manipulate tags, IDs, classes, Attributes, or Elements of HTML using commands or methods provided by the Document object. Using DOM, the JavaScript gets access to HTML as well as CSS of the web page and can also add behavior to the HTML elements. so basically, Document Object Model is an API that represents and interacts with HTML or XML documents.

### Why DOM is required?

HTML is used to **structure**the web pages and JavaScript is used to add **behavior**to our web pages. When an HTML file is loaded into the browser, the JavaScript can not understand the HTML document directly. So, a corresponding document is created (DOM). **DOM is basically the representation of the same HTML document but in a different format with the use of objects**. JavaScript interprets DOM easily i.e JavaScript can not understand the tags(<h1>H</h1>) in HTML document but can understand object h1 in DOM. Now, JavaScript can access each of the objects (h1, p, etc) by using different functions.

### Virtual DOM

VDOM is the ***virtual representation of Real DOM***

* React update the state changes in Virtual DOM first and then it syncs with Real DOM
* Virtual DOM is just like a ***blueprint of a machine***, can do the changes in the blueprint but those changes will not directly apply to the machine.
* Virtual DOM is a programming concept where a *virtual representation of a UI* is kept in memory synced with “Real DOM” by a library such as ReactDOM and this process is called reconciliation
* Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions.

### 30. What do you mean by BOM?

Browser Object Model is known as BOM. ***It allows users to interact with the browser.*** A browser's initial object is a window. As a result, you may call all of the window's functions directly or by referencing the window. The **document, history, screen, navigator, location,** and other attributes are available in the window object.

### 31. JavaScript Call Stack:

In JavaScript, the call stack is a mechanism for managing function calls. It keeps track of the functions that are currently being executed and the order in which they were called. When a function is called, it is added to the top of the call stack, and when the function completes, it is removed from the stack.

**Function Calls:**

When a function is called, it is pushed onto the call stack.

The function at the top of the stack is the one currently being executed.

**Execution:**

Each function is executed in a last-in, first-out (LIFO) order.

The function at the top of the stack must complete before the next one can start.

**Stack Overflow:**

If the call stack becomes too deep (due to too many nested function calls), it can result in a stack overflow error.

### Lazy Loading in Web Development:

Lazy loading is a technique in computer programming and web development where resources, such as images, scripts, or other assets, are loaded only when they are actually needed. The opposite of lazy loading is eager loading, where all resources are loaded upfront, regardless of whether they are immediately necessary.

**Images:**

In web development, one common use of lazy loading is with images. Instead of loading all images when the page loads, images below the fold (not initially visible) can be loaded only when the user scrolls down to view them. This can significantly improve initial page load times.

<img src="placeholder.jpg" data-src="image-to-load.jpg" class="lazyload" alt="Description">

In this example, the actual image source (data-src) is set as a data attribute. JavaScript is then used to load the image when it comes into the viewport.

**JavaScript Libraries:**

Lazy loading is also applicable to JavaScript files. For large web applications, you might have different sections or components that use specific JavaScript functionality. Instead of loading all scripts at once, you can load them dynamically when needed.

// Example using dynamic import for lazy loading JavaScript module

const button = document.getElementById('loadButton');

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

  const module = await import('./path/to/module.js');

  module.doSomething();

});

In this example, the JavaScript module is loaded only when the user clicks the button.

**Frameworks and Libraries:**

Some JavaScript frameworks and libraries support lazy loading of components or modules. For example, in React, you might use **React.lazy** for dynamically loading components.

const MyComponent = React.lazy(() => import('./MyComponent'));

**CSS:**

Lazy loading can also be applied to CSS. For example, in certain scenarios, you might want to load a specific stylesheet only when a particular component is rendered.

**Benefits of Lazy Loading:**

* Faster Initial Page Load:

Lazy loading reduces the initial payload of a webpage, leading to faster load times, especially for users with slower internet connections or on mobile devices.

* Optimized Resource Usage:

By loading resources only when needed, you optimize the use of network bandwidth and reduce the strain on the user's device.

* Improved User Experience:

Users get a faster and more responsive experience, as unnecessary resources are not loaded upfront.

* Reduced Server Load:

Lazy loading can help distribute server load over time rather than all at once, particularly for large applications with numerous resources.

### Diff b/w js and jsx

JavaScript (JS) and JSX are related but distinct technologies used in the context of React, a JavaScript library for building user interfaces. Here are the main differences between JavaScript and JSX:

**Syntax:**

**JavaScript (JS):** Standard JavaScript is a programming language with a specific syntax, and it's used for defining logic, data structures, and functionality.

**JSX:** JSX (JavaScript XML) is a syntax extension for JavaScript recommended by React. It looks similar to XML or HTML and is used to describe the structure of UI components in React.

**Usage:**

**JavaScript (JS):** Used for writing logic, functions, and managing the overall behavior of a web application. It can be used both on the client and server sides.

**JSX:** Primarily used in React for defining the structure of React components. JSX is not standalone JavaScript and needs to be transpiled into JavaScript before being executed in the browser.

### Shallow copy and Deep copy

**1. Shallow Copy:**

* A shallow copy creates a new object, but it does not create new copies of nested objects or arrays.
* It copies the references to the original nested objects, so changes to nested objects in the copied structure will affect the original object and vice versa.
* Shallow copying can be done using methods like `Object.assign()`, spread syntax (`...`), or the `Array.slice()` method.

Example of shallow copy using spread syntax:

const originalObject = { a: 1, b: { c: 2 } };

const shallowCopy = { ...originalObject };

console.log(shallowCopy === originalObject); // false (different objects)

console.log(shallowCopy.b === originalObject.b); // true (nested reference is shared)

**2. Deep Copy:**

* A deep copy creates a new object and recursively copies all nested objects and arrays.
* It results in a completely independent copy, so changes to the copied object or its nested structures do not affect the original object.
* Deep copying is not directly supported in JavaScript, so you may need to use custom functions or external libraries like `lodash` to achieve it.

const originalObject = { a: 1, b: { c: 2 } };

const deepCopy = JSON.parse(JSON.stringify(originalObject));

console.log(deepCopy === originalObject); // false (different objects)

console.log(deepCopy.b === originalObject.b); // false (nested reference is independent)

### Shallow Comparison and Deep Comparison:

Shallow comparison involves comparing only the immediate properties or values of objects, not their nested properties or values.

const obj1 = { a: 1, b: 2 };

const obj2 = { a: 1, b: 2 };

console.log(obj1 === obj2); // Shallow comparison

// Output: false (because obj1 and obj2 reference different objects in memory)

Deep comparison involves comparing not only the immediate properties or values but also the nested properties or values of objects.

const obj1 = { a: 1, b: { c: 2 } };

const obj2 = { a: 1, b: { c: 2 } };

const isEqual = deepEqual(obj1, obj2); // Custom deep equality check

console.log(isEqual);

// Output: true (because the nested properties are also equal)