**JAVASCRIPT 30-4-2023**

**1) What is the inheritance? How we can achieve?**

Inheritance refers to an object's ability to access methods and other properties from another object. Objects can inherit things from other objects. Inheritance in JavaScript works through something called prototypes and this form of inheritance is often called prototypal inheritance.

Coding:

function Bus(){

this.fw =2

this.totalWheels = function(){

console.log(this.fw+this.bw)

}

}

-under f2 console developer toos-

Bus.prototype.bw = 4

var obj = new Bus()

obj.totalWheels()

O/P = 6

What is Prototype?- What is Prototypal Inheritance?- What is Prototype Chain? - Why we call it \_proto\_ ?- What is inhertance in Javascript

----------------------------

protype chain

The prototype chain is a fundamental concept in JavaScript's object-oriented programming model. Each object in JavaScript has an internal property called [[Prototype]], which references another object called its prototype. This forms a chain of prototypes, known as the prototype chain.

let parent = {

greet: function() {

return 'Hello';

}

};

let child = Object.create(parent);

child.name = 'Sachin';

console.log(child.name);

console.log(child.greet());

---------------------------------------------------------------------------------------------------

**2) What is Prototype?**

A prototype is an existing inbuilt functionality in JavaScript. Whenever we create a JavaScript function, JavaScript adds a prototype property to that function. A prototype is an object, where it can add new variables and methods to the existing object. i.e., Prototype is a base class for all the objects, and it helps us to achieve the inheritance

Coding:

function Bus(){

this.fw =2

this.totalWheels = function(){

console.log(this.fw+this.bw)

}

}

-under f2 console developer toos-

Bus.prototype.bw = 4

var obj = new Bus()

obj.totalWheels()

O/P = 6

------------------------------------------------------------------------------------------------

**3) Do you know the different between call , bind and apply?**

1)call invokes the function and allows you to pass in arguments one by one.

2)apply invokes the function and allows you to pass in arguments as an array.

3)bind returns a new function, allowing you to pass in a this array and any number of arguments

---------

Syntax:

function f1(){

console.log("Called")

}

f1()

f1.call()

f1.apply()

f1.bind()()

o/p Called

---------

Coding:

function f1(a){

console.log(this.x+a)

}

// this

// Window {window: Window, self: Window, document: document, name: '', location: Location, …}

f1(10)

o/p NaN

var o={x:30}

f1.call(o,10)

f1.apply(o,[10])

f1.bind(o,10)()

f1.bind(o)(10)

-------------------------------------------------------------------------------------------------------

**4) What is async and await?**

Async/Await makes it easier to write promises. The keyword 'async' before a function makes the function return a promise, always. And the keyword await is used inside async functions, which makes the program wait until the Promise resolve.

coding:

function f1(){

return new Promise(function(resolve,reject){

setTimeout(function(){

resolve('success')

},5000)

})

}

// f1()

// .then(function(data){

// console.log("then cb",data);

// })

// .catch(function(data){

// console.log("then cb",data)

// })

async function getData(){

const data = await f1();

console.log(data)

}

getData()

----------------------

What is the use of then and catch in JavaScript?

then : when a promise is successful, you can then use the resolved data. catch : when a promise fails, you catch the error, and do something with the error information

------------------------------------------------------------------------------------------------------

**5) Can you explain me what are call back function?**

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

function f1(a,b,c){

a(1)

b(2)

c(3)

}

f1(

function(data){

console.log(data)

},

function(data){

console.log(data)

},

function(data){

console.log(data)

}

)

------------------------------------------------------------------------------------------------------

**6) Tell me Different between undefined and not defined?**

In JavaScript, undefined and not defined are two different things.

undefined means a variable has been declared, but has not yet been assigned a value. For example:

let num;

console.log(num); // Output: undefined

In this example, we declared the variable num but didn't assign any value to it. So, when we try to log the value of num, it returns undefined.

On the other hand, not defined means a variable has not been declared. For example:

console.log(num); // Output: Uncaught ReferenceError: num is not defined

In this example, we didn't declare the variable num, but still we try to log its value. So, it returns an error saying num is not defined.

So, the key difference between undefined and not defined is that undefined means a variable has been declared but has not yet been assigned a value, whereas not defined means a variable has not been declared at all.

**7) What is event?**

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page. When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc

Coding:

<html>

<head>

<script type = "text/javascript">

function sayHello() {

alert("Hello World")

}

</script>

</head>

<body>

<p>Click the following button and see result</p>

<form>

<input type = "button" onclick = "sayHello()" value = "Say Hello"/>

</form>

</body>

</html>

-------------------------------------------------------------------------------------------------------**8) What is difference between setTimeout and setInterval?**

setTimeout:

The setTimeout() method is used to execute a function after waiting for the specified time interval.

setTimeout(function(){

console.log('5sec completed')

},5000)

---------------

setInterval:

The setInterval() function is commonly used to set a delay for functions that are executed again and again

setInterval(function(){

console.log("calling ")

},1000)

1

clearInterval(1) (or) clearTimeout(1)

---------------------------

What is the difference between setTimeout and setInterval?

setTimeout allows us to run a function once after the interval of time. setInterval allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval

-------------------------------------------------------------------------------------------------------

**9) What is an event loop in JavaScript?**

Event loop, it's a job to monitor call stack and the callback queue.If the call stack is empty, Event loop will take the first event from the callback queue and will push it to the call stack, now call stack will starts it's execution

function f(){

console.log(1)

setTimeout(function(){

console.log(2)

},5000)

console.log(3)

}

f()

o/p

1

3

2

-----

function f(){

console.log(1)

setTimeout(function(){

console.log(2)

},0)

console.log(3)

}

f()

o/p

1

3

2

----------------------------------------------------------------------------------------------

**10) What is the difference between Debounce and throttle in JavaScript?**

Rewrite the code::: Debouncing used to improve the browser performance. There might be some functionality in a web page that requires time-consuming process. If such a method is invoked frequently, it might greatly affect the performance of the browser. So Debouncing is nothing but reducing unnecessary time consuming process and it increasing browser performance. For example of a search bar in an e-commerce website.

//before debounce

<p>

<b>Search:</b> <input onkeyup="fnSearch()" placeholder="Search..."/>

</p>

<script>

function fnSearch(){

console.log('AJAX req')

}

</script>

-------------------------------------------------------------

**Debounce:**

<p>

<b>Search:</b> <input onkeyup="debounce()" placeholder="Search..."/>

</p>

<script>

var timer;

function fnSearch(){

console.log('AJAX req')

}

function debounce(){

clearTimeout(timer)

timer=setTimeout(function(){

fnSearch();

},5000);

}

</script>

**Throttling:**

Throttling is used to call a function after every millisecond or a particular interval of time only the first click is executed immediately.

coding:

<p>

<b>Search:</b> <input onkeyup="fnThrottling()" placeholder="Search..."/>

</p>

<script>

var timer, flag=true;

function fnSearch(){

console.log("AJAX")

}

function fnThrottling(){

if(flag){

fnSearch();

flag=false;

}

clearTimeout(timer)

timer=setTimeout(function(){

fnSearch();

},5000);

}

</script>

-----------------------------------------------------------------------------------------------------

**11) Can you give me example of Closure?**

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Coding:

<script src="app.js">

function makeFunc() {

var name = 'Mozilla';

function displayName() {

alert(name);

}

return displayName;

}

var myFunc = makeFunc();

myFunc();

</script>

-------------------------------------------------------------------------------------------------------

**12) Lexical scope?**

Lexical scope is the ability for a function scope to access variables from the parent scope. We call the child function to be lexically bound by that of the parent function.

function f1() {

var a = 10

function f2() {

var b = 20

console.log(a);

console.log(b);

}

f2();

}

f1();

--------------------------------------------------------------------------------------------------

**13) Spread Operator?**

The spread operator in JavaScript is denoted by three dots (...) and is used to spread an iterable object, such as an array or string, into individual elements.

coding:1

const array1 = [1, 2, 3];

const array2 = [4, 5, 6];

const newArray = [...array1, ...array2];

console.log(newArray); // [1, 2, 3, 4, 5, 6]

coding:2

const myVehicle = {

brand: 'Ford',

model: 'Mustang',

color: 'red'

}

const updateMyVehicle = {

type: 'car',

year: 2021,

color: 'yellow'

}

const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle}

//Check the result object in the console:

console.log(myUpdatedVehicle);

-------------------------------------------------------------------------------------------------------

**14) Event looping in javascript?**

Event loop, it's a job to monitor call stack and the callback queue.

If the call stack is empty, Event loop will take the first event from the callback queue and will push it to the call stack, now call stack will starts it's execution

function f1(){

console.log(1)

setTimeout(function(){

console.log(2)

},1000)

console.log(3)

}

------------------------------------------------------------------------------------------------------

**15) What is different between var, let and const ? Have you worked on ES6?**

In JavaScript, there are three ways to declare a variable: var, let, and const. The main differences between them are in terms of scope, hoisting, and reassignment. Here are some examples:

var:

javascript

// example with var

function exampleVar() {

var x = 10;

if (true) {

var x = 20;

console.log(x); // Output: 20

}

console.log(x); // Output: 20

}

The variable x is function-scoped and can be redeclared and reassigned anywhere in the function, even inside the if block.

let:

javascript

// example with let

function exampleLet() {

let x = 10;

if (true) {

let x = 20;

console.log(x); // Output: 20

}

console.log(x); // Output: 10

}

The variable x is block-scoped and can be reassigned, but not redeclared, within the block it was declared in.

const:

javascript

// example with const

function exampleConst() {

const x = 10;

if (true) {

const x = 20;

console.log(x); // Output: 20

}

console.log(x); // Output: 10

}

The variable x is also block-scoped like let, but cannot be reassigned or redeclared after it has been initialized.

In general, it is recommended to use const for variables that won't be reassigned, let for variables that will be reassigned, and avoid using var due to its global or function scope and the potential for hoisting issues.

--------------------------------------------------------------------------

**16) What is ‘ this’ keyword javascript?**

‘ this’ keyword javascript with example

In JavaScript, the this keyword refers to the current execution context of a function or method. The value of this depends on how the function or method is called, and it can change depending on the context.

Here's an example to demonstrate how this works:

javascript

let person = {

name: 'John',

sayHello: function() {

console.log(`Hello, my name is ${this.name}`);

}

};

person.sayHello(); // output: "Hello, my name is John"

In this example, person is an object with a name property and a sayHello method. When the sayHello method is called on the person object using person.sayHello(), this refers to the person object. So when the sayHello method logs the name property of this, it logs the name of the person object.

Here's another example to show how this can change depending on how a function is called:

javascript

let name = 'Jane';

function sayHello() {

console.log(`Hello, my name is ${this.name}`);

}

let person = {

name: 'John',

sayHello: sayHello

};

sayHello(); // output: "Hello, my name is Jane"

person.sayHello(); // output: "Hello, my name is John"

In this example, sayHello is a function that logs the name property of this. When sayHello is called directly without an object context, this refers to the global object (window in a browser environment or global in Node.js) because the function is not called as a method of an object.

When person.sayHello is called, this refers to the person object, which has a name property of 'John'. So when the sayHello method logs the name property of this, it logs the name of the person object.

--------------------------------------------------------------------------

**17) What is Hoisting?**

Hoisting is a JavaScript mechanism that allows variables and function declarations to be used before they are actually declared in the code. In other words, JavaScript moves variable and function declarations to the top of their respective scopes during the compilation phase, before the code is executed.

Here's an example to demonstrate hoisting in JavaScript:

javascript

console.log(x); // Output: undefined

var x = 10;

In this example, the variable x is declared and assigned a value of 10. However, when we try to log the value of x before its declaration, we get undefined instead of a ReferenceError. This is because the declaration of x is hoisted to the top of its scope during compilation, but its assignment remains in place. So the code is effectively interpreted like this:

javascript

var x;

console.log(x); // Output: undefined

x = 10;

Note that hoisting only applies to declarations and not to initializations. If you try to use a variable before it has been declared and initialized, you will get a ReferenceError.

Here's an example that demonstrates hoisting with function declarations:

javascript

sayHello(); // Output: "Hello, World!"

function sayHello() {

console.log("Hello, World!");

}

In this example, we call the sayHello function before its declaration, but the code still works and logs the expected output. This is because the function declaration is hoisted to the top of its scope during compilation, just like variable declarations. So the code is effectively interpreted like this:

javascript

function sayHello() {

console.log("Hello, World!");

}

sayHello(); // Output: "Hello, World!"

It's important to note that hoisting can make your code harder to read and understand, so it's generally recommended to declare your variables and functions at the top of their respective scopes to avoid confusion.

--------------------------------------------------------------------------

**18) What arrow function?**

Arrow functions were introduced in ES6 as a more concise way to write function expressions in JavaScript. Arrow functions have a shorter syntax compared to regular function expressions, and they also lexically bind this, meaning that they inherit the this value from the surrounding code rather than having their own this value.

Here's an example of a regular function expression:

javascript

let square = function(x) {

return x \* x;

};

console.log(square(5)); // Output: 25

And here's the equivalent arrow function:

javascript

let square = (x) => {

return x \* x;

};

console.log(square(5)); // Output: 25

As you can see, the arrow function has a shorter syntax compared to the regular function expression. We can further simplify this arrow function by omitting the curly braces and return keyword for single-line expressions:

javascript

let square = x => x \* x;

console.log(square(5)); // Output: 25

In addition to their concise syntax, arrow functions also lexically bind this, which can make them easier to use in certain situations. Here's an example to illustrate this:

javascript

let person = {

name: 'John',

sayHello: function() {

setTimeout(() => {

console.log(`Hello, my name is ${this.name}`);

}, 1000);

}

};

person.sayHello(); // Output: "Hello, my name is John" (after 1 second)

In this example, we have an object person with a name property and a sayHello method. Inside the sayHello method, we use setTimeout to log a message after a 1 second delay. We use an arrow function for the callback function passed to setTimeout so that it can access the this value of the person object. If we used a regular function expression for the callback function, this would refer to the global object instead of the person object.

--------------------------------------------------------------------------

**19) What is arrow function? How it is beneficial compare to normal function?**

Arrow functions are a new way of defining functions in JavaScript that were introduced in ES6. Arrow functions have a concise syntax, they don't bind their own this value, and they're generally easier to read and write than traditional function expressions.

Here's an example of a traditional function expression:

javascript

let multiply = function(x, y) {

return x \* y;

};

console.log(multiply(5, 10)); // Output: 50

And here's the equivalent arrow function:

javascript

let multiply = (x, y) => x \* y;

console.log(multiply(5, 10)); // Output: 50

As you can see, the arrow function has a more concise syntax, and we don't need to use the return keyword or curly braces when we have a single expression in the function body.

Arrow functions are also useful because they don't bind their own this value, unlike traditional function expressions. When you use this inside an arrow function, it refers to the this value of the surrounding lexical context, which is usually the enclosing function or the global scope. This makes arrow functions more predictable and easier to reason about than traditional functions, which can have unexpected this values depending on how they are called.

Another benefit of arrow functions is that they're generally easier to read and write than traditional functions. The concise syntax and lack of explicit return and curly braces can make the code more readable and easier to understand, especially for simple functions.

However, it's worth noting that arrow functions aren't always the best choice for every situation. They don't have a prototype property, so they can't be used as constructors for creating new objects. They also can't be used with the arguments object or as a method definition in an object literal. So it's important to understand the limitations of arrow functions and choose the right tool for the job.

--------------------------------------------------------------------------

**20) What is JavaScript bind function?**

The bind() method is a built-in function in JavaScript that allows you to create a new function that has a specified this value and can optionally have one or more arguments pre-set. When you call bind() on a function, it returns a new function that has the same code as the original function but with a bound this value and possibly some pre-set arguments.

Here's an example of how to use bind():

javascript

let person = {

name: 'John',

sayHello: function() {

console.log(`Hello, my name is ${this.name}`);

}

};

let sayHelloToMary = person.sayHello.bind({name: 'Mary'});

sayHelloToMary(); // Output: "Hello, my name is Mary"

In this example, we have an object person with a name property and a sayHello method that logs a message using this.name. We use the bind() method to create a new function sayHelloToMary that has a this value of {name: 'Mary'}. When we call sayHelloToMary(), it logs the message "Hello, my name is Mary" instead of "Hello, my name is John".

The bind() method can also be used to pre-set arguments for a function. Here's an example:

javascript

let multiply = function(x, y) {

return x \* y;

};

let double = multiply.bind(null, 2);

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

In this example, we have a function multiply that takes two arguments and returns their product. We use bind() to create a new function double that has a pre-set first argument of 2. When we call double(5), it returns the product of 2 and 5, which is 10.

The bind() method is a powerful tool for creating new functions with a bound this value or pre-set arguments. It can be especially useful in situations where you need to pass a function as a callback and want to ensure that it has the correct this value or pre-set arguments.

--------------------------------------------------------------------------

**21) What is Map method ? What is different between for loop and Map method?**

The map() method is a built-in function in JavaScript that allows you to create a new array by applying a function to each element of an existing array. The map() method takes a callback function as an argument and returns a new array with the same length as the original array, where each element is the result of applying the callback function to the corresponding element of the original array.

Here's an example of how to use the map() method:

javascript

let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = numbers.map(function(number) {

return number \* 2;

});

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

In this example, we have an array numbers with 5 elements. We use the map() method to create a new array doubledNumbers that contains the doubled values of each element of numbers. The callback function takes one argument number and returns the result of multiplying it by 2. The resulting doubledNumbers array contains the values [2, 4, 6, 8, 10].

The map() method is very versatile and can be used to transform an array in many different ways, depending on the function you pass as the callback. You can use it to:

Transform an array of objects into an array of values from a specific property

Generate a new array of random values

Create a new array with only certain elements from the original array

Generate a new array by combining values from multiple arrays

Here's an example of using map() to extract a specific property from an array of objects:

javascript

let users = [

{name: 'John', age: 30},

{name: 'Mary', age: 25},

{name: 'Peter', age: 40}

];

let userNames = users.map(function(user) {

return user.name;

});

console.log(userNames); // Output: ["John", "Mary", "Peter"]

In this example, we have an array users with three objects, each containing a name and an age property. We use the map() method to create a new array userNames that contains only the name property of each object. The resulting userNames array contains the values ["John", "Mary", "Peter"]

-------------------

different between for loop and Map method

Both for loops and the map() method can be used to iterate over arrays in JavaScript, but they have some differences in terms of syntax and functionality.

The main difference between for loops and the map() method is that for loops provide more flexibility and control over the iteration process, while the map() method provides a more concise syntax for transforming arrays.

Here are some key differences:

Syntax: The syntax for a for loop is more verbose and requires more lines of code than the map() method. Here's an example:

javascript

// Using a for loop

let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = [];

for(let i = 0; i < numbers.length; i++) {

doubledNumbers.push(numbers[i] \* 2);

}

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

// Using map

let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = numbers.map(function(number) {

return number \* 2;

});

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

Functionality: for loops provide more control over the iteration process, allowing you to break out of the loop or skip certain elements based on conditions. The map() method, on the other hand, always iterates over every element of the array and applies the function to each element.

javascript

// Using a for loop to skip even numbers

let numbers = [1, 2, 3, 4, 5];

let doubledOddNumbers = [];

for(let i = 0; i < numbers.length; i++) {

if(numbers[i] % 2 === 0) {

continue; // skip even numbers

}

doubledOddNumbers.push(numbers[i] \* 2);

}

console.log(doubledOddNumbers); // Output: [2, 6, 10]

// Using map to skip even numbers

let numbers = [1, 2, 3, 4, 5];

let doubledOddNumbers = numbers

.filter(function(number) {

return number % 2 !== 0; // filter out even numbers

})

.map(function(number) {

return number \* 2; // double remaining odd numbers

});

console.log(doubledOddNumbers); // Output: [2, 6, 10]

In this example, we have an array numbers with 5 elements. In the for loop, we skip even numbers and only double the odd numbers. In the map() method, we use the filter() method to remove even numbers from the array before applying the map() method to the remaining odd numbers.

Overall, for loops are more flexible and provide more control over the iteration process, but the map() method can provide a more concise and readable syntax for transforming arrays. The choice between for loops and the map() method depends on the specific requirements of your code and the type of iteration you need to perform

------------------------------------------

different between forEach and Map method

In JavaScript, both the forEach() and map() methods are used to iterate over arrays, but they have some differences in terms of their functionality and the way they handle return values.

Here are some key differences:

Return Value: The forEach() method returns undefined, while the map() method returns a new array with the same number of elements as the original array.

javascript

// Using forEach

let numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(number) {

console.log(number \* 2);

});

// Output: 2 4 6 8 10

// Using map

let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = numbers.map(function(number) {

return number \* 2;

});

console.log(doubledNumbers);

// Output: [2, 4, 6, 8, 10]

In this example, the forEach() method is used to double each number in the array, but it does not return a new array. The map() method is also used to double each number, but it returns a new array with the doubled values.

Mutability: The forEach() method modifies the original array, while the map() method creates a new array without modifying the original array.

javascript

// Using forEach

let numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(number, index) {

numbers[index] = number \* 2;

});

console.log(numbers); // Output: [2, 4, 6, 8, 10]

// Using map

let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = numbers.map(function(number) {

return number \* 2;

});

console.log(numbers); // Output: [1, 2, 3, 4, 5]

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

In this example, the forEach() method is used to modify the original array by doubling each number. The map() method is used to create a new array without modifying the original array.

Overall, the forEach() method is useful when you need to perform a task for each element in an array without modifying the array, while the map() method is useful when you need to transform an array into a new array of the same length.

--------------------------------------------------------------------------

**22) Can be used for loop in array?**

Yes, a for loop can be used to iterate over an array in JavaScript. Here is an example:

javascript

let numbers = [1, 2, 3, 4, 5];

for (let i = 0; i < numbers.length; i++) {

console.log(numbers[i]);

}

In this example, the for loop is used to iterate over the numbers array. The loop starts at index 0, which is the first element of the array, and continues until the index is less than the length of the array. On each iteration, the loop prints the current element of the array using the index i.

However, in modern JavaScript, the forEach() method or other array methods such as map(), filter(), reduce(), etc. are often used instead of for loops to iterate over arrays because they provide a more concise and readable way of performing common operations on arrays.

--------------------------------------------------------------------------

**23) Have you heard about ternary operator?What is ternary operator?**

The ternary operator, also known as the conditional operator, is a shorthand way of writing an if...else statement in JavaScript. It allows you to write a single line of code that evaluates a condition and returns one of two values based on the result of the evaluation.

Here is an example of using the ternary operator to find the largest of two numbers:

javascript

let a = 5;

let b = 10;

let largest = a > b ? a : b;

console.log(largest); // Output: 10

In this example, the ternary operator is used to compare the values of a and b. If a is greater than b, the value of a is assigned to largest. Otherwise, the value of b is assigned to largest. The result of the evaluation is then printed to the console.

The ternary operator is written using the following syntax:

javascript

condition ? value1 : value2

If the condition is true, value1 is returned. Otherwise, value2 is returned.

--------------------------------------------------------------------------

**24) What is nullish coalescing operator?**

The nullish coalescing operator (??) is a new operator introduced in JavaScript ES2020. It allows you to provide a default value for a variable if its value is null or undefined. It is similar to the logical OR operator (||), but it only returns the default value if the variable is null or undefined, not for other falsy values such as false, 0, or an empty string.

Here is an example of using the nullish coalescing operator:

javascript

let name;

let defaultName = 'John';

console.log(name ?? defaultName); // Output: John

name = 'Alice';

console.log(name ?? defaultName); // Output: Alice

name = '';

console.log(name ?? defaultName); // Output: ''

name = 0;

console.log(name ?? defaultName); // Output: 0

name = null;

console.log(name ?? defaultName); // Output: John

name = undefined;

console.log(name ?? defaultName); // Output: John

In this example, the name variable is assigned different values, including null, undefined, an empty string, and 0. The nullish coalescing operator is used to provide a default value of defaultName if the value of name is null or undefined. In the first example, since name is undefined, the default value of John is returned. In the second example, since name is 'Alice', its value is returned instead of the default value. In the third example, since name is an empty string, it is not considered null or undefined, so an empty string is returned. In the fourth example, since name is 0, it is not considered null or undefined, so 0 is returned. Finally, in the last two examples, since name is null or undefined, the default value of John is returned.

The nullish coalescing operator is a useful addition to JavaScript that allows for more concise and readable code when providing default values.

-------------------------------------------------------------------------

**25) What is optional chaining?**

The optional chaining operator (?.) is a new operator introduced in JavaScript ES2020. It allows you to safely access properties of an object or elements of an array, even if one or more of the intermediate properties or elements do not exist. It helps to avoid errors that can occur when trying to access properties or elements that are null or undefined.

Here is an example of using the optional chaining operator:

javascript

let user = {

name: 'Alice',

address: {

street: '123 Main St.',

city: 'New York',

country: 'USA'

}

};

console.log(user.address?.zipCode); // Output: undefined

user = {

name: 'Bob'

};

console.log(user.address?.zipCode); // Output: undefined

In this example, the user object has an address property that contains nested properties such as street, city, and country. However, it does not have a zipCode property. The optional chaining operator is used to access the zipCode property of the address property. If the address property is null or undefined, the optional chaining operator returns undefined instead of throwing an error.

If we try to access the zipCode property without the optional chaining operator like this:

javascript

console.log(user.address.zipCode); // Output: Uncaught TypeError: Cannot read property 'zipCode' of undefined

we would get a TypeError because the address property is undefined on the user object.

The optional chaining operator can be used with other operators such as the nullish coalescing operator (??) to provide a default value if the property or element is null or undefined.

--------------------------------------------------------------------------

**26) What is OR operator ?**

The OR (||) operator is a logical operator in JavaScript that returns the first truthy value it encounters, or the last falsy value if none are found. It is often used to provide default values for variables or parameters.

Here is an example of using the OR operator:

javascript

let name;

let defaultName = 'John';

console.log(name || defaultName); // Output: John

name = 'Alice';

console.log(name || defaultName); // Output: Alice

name = '';

console.log(name || defaultName); // Output: John

name = 0;

console.log(name || defaultName); // Output: John

In this example, the name variable is assigned different values, including an empty string and 0, which are falsy values. The OR operator is used to provide a default value of defaultName if the value of name is falsy. In the first example, since name is falsy, the default value of John is returned. In the second example, since name is truthy (the string 'Alice'), its value is returned instead of the default value. In the third and fourth examples, since name is falsy, the default value of John is returned.

The OR operator can also be used to check if a variable or expression exists, like this:

javascript

let foo = null;

let bar = 'Hello';

console.log(foo || 'default value'); // Output: default value

console.log(bar || 'default value'); // Output: Hello

In this example, the OR operator is used to check if foo and bar are truthy. Since foo is falsy (null), the default value of 'default value' is returned. Since bar is truthy ('Hello'), its value is returned instead of the default value

--------------------------------------------------------------------------

**27) Do you know about Promises? What are Promises?**

1 promises:

Promises in JavaScript are a way to handle asynchronous operations. A promise represents a value that may not be available yet, but will be resolved at some point in the future. Once the promise is resolved, it may either be fulfilled with a value or rejected with a reason for failure.

Here's an example of creating a promise in JavaScript:

javascript

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

// Some asynchronous operation

const data = fetch('https://api.example.com/data'); //https://jsonplaceholder.typicode.com/posts

if (data) {

resolve(data);

} else {

reject(new Error('Data not found'));

}

});

// Consuming the promise

promise

.then((data) => {

console.log(data);

})

.catch((error) => {

console.error(error);

});

In this example, we create a new Promise instance with a callback function that takes two arguments: resolve and reject. This callback function represents an asynchronous operation that may either resolve with a value (in this case, the data fetched from an API endpoint) or reject with an error (in this case, an error message). Once the asynchronous operation is completed, we check if the data is available and either resolve the promise with the data or reject it with an error.

To consume the promise, we use the .then() method to handle the fulfilled value and the .catch() method to handle the rejected reason. If the promise is fulfilled, the .then() method will be called with the resolved value as its argument. If the promise is rejected, the .catch() method will be called with the rejected reason as its argument.

Promises are a powerful tool for handling asynchronous code in JavaScript, and they can be used in many different scenarios, such as making network requests, handling user input, and processing large amounts of data.

-------------------------

**2) promise in javascript with example**

In JavaScript, a Promise is an object representing the eventual completion or failure of an asynchronous operation and its resulting value. Promises provide a more readable and manageable way to handle asynchronous operations compared to traditional callback functions. Here's an example of using a Promise to load data asynchronously:

javascript

function loadData() {

return new Promise(function(resolve, reject) {

// Make an asynchronous request to load data

let xhr = new XMLHttpRequest();

xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1');

xhr.onload = function() {

if (xhr.status === 200) {

// Resolve the promise with the loaded data

resolve(xhr.responseText);

} else {

// Reject the promise with an error message

reject('Error loading data');

}

};

xhr.onerror = function() {

// Reject the promise with an error message

reject('Error loading data');

};

xhr.send();

});

}

// Call the loadData function and handle the promise's resolved or rejected states

loadData().then(function(data) {

console.log('Data loaded:', data);

}).catch(function(error) {

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

});

In this example, the loadData function returns a new Promise object. Inside the Promise constructor, an asynchronous request is made to load data using the XMLHttpRequest object. If the request is successful, the Promise is resolved with the loaded data using the resolve function. If the request fails or encounters an error, the Promise is rejected with an error message using the reject function.

The loadData function is then called and its Promise is handled using the then method, which takes a callback function to handle the resolved state of the Promise. The catch method is used to handle any errors or rejected states of the Promise.

Note that Promises can also be used to chain multiple asynchronous operations together, or to combine multiple Promises using methods like Promise.all.

--------------------------------------------------------------------------

**28) What is promise? what is promise.race()? what is promise.all?**

A promise is an object in JavaScript that represents the eventual completion or failure of an asynchronous operation, and its resulting value. Promises are a way to handle asynchronous code that can simplify error handling and make it easier to write readable code.

The Promise.race() method returns a promise that resolves or rejects as soon as one of the promises in an iterable resolves or rejects, with the value or reason from that promise. Here's an example:

javascript

const promise1 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('promise1');

}, 1000);

});

const promise2 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('promise2');

}, 2000);

});

Promise.race([promise1, promise2])

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

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

In this example, two promises are created using the Promise constructor. The setTimeout function is used to simulate an asynchronous operation that resolves with a string value after a specified delay. The Promise.race() method is called with an array of promises as an argument, and it returns a new promise that resolves or rejects as soon as one of the promises in the array resolves or rejects. In this case, promise1 resolves after 1 second, and promise2 resolves after 2 seconds. Since promise1 resolves first, the then callback is called with the value 'promise1'.

The Promise.all() method returns a promise that resolves with an array of values when all of the promises in an iterable have resolved, or rejects with the reason of the first promise that rejects. Here's an example:

javascript

const promise1 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('promise1');

}, 1000);

});

const promise2 = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('promise2');

}, 2000);

});

Promise.all([promise1, promise2])

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

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

In this example, two promises are created using the Promise constructor, and the setTimeout function is used to simulate an asynchronous operation that resolves with a string value after a specified delay. The Promise.all() method is called with an array of promises as an argument, and it returns a new promise that resolves with an array of values when all of the promises in the array have resolved. In this case, both promises resolve successfully, so the then callback is called with an array containing the values 'promise1' and 'promise2'.

Overall, Promise.race() and Promise.all() are two useful methods for working with promises in JavaScript.

--------------------------------------------------------------------------

**29) Can you explain me what call back function and promises and async and await?**

Callbacks, Promises, and async/await are three ways to handle asynchronous code in JavaScript.

A callback is a function that is passed as an argument to another function and is called when the parent function completes. Here's an example:

javascript

function fetchData(callback) {

setTimeout(() => {

callback('data');

}, 1000);

}

fetchData(data => {

console.log(data);

});

In this example, the fetchData function takes a callback function as an argument and uses the setTimeout function to simulate an asynchronous operation. When the operation completes after 1 second, the callback function is called with a string value, which is logged to the console.

Promises are objects that represent the eventual completion or failure of an asynchronous operation, and its resulting value. Here's an example:

javascript

function fetchData() {

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

setTimeout(() => {

resolve('data');

}, 1000);

});

}

fetchData()

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

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

In this example, the fetchData function returns a new Promise that resolves with a string value after 1 second. The then method is called on the Promise object to handle the successful completion of the operation, and the catch method is called to handle any errors that occur.

async/await is a syntactic sugar for working with Promises that was introduced in ES2017. It allows you to write asynchronous code in a synchronous style using the async and await keywords. Here's an example:

javascript

function fetchData() {

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

setTimeout(() => {

resolve('data');

}, 1000);

});

}

async function getData() {

try {

const data = await fetchData();

console.log(data);

} catch (error) {

console.log(error);

}

}

getData();

In this example, the fetchData function returns a new Promise that resolves with a string value after 1 second. The getData function is declared as an async function and uses the await keyword to wait for the Promise to resolve before logging the value to the console. The try/catch block is used to handle any errors that occur.

Overall, callbacks, Promises, and async/await are three powerful ways to handle asynchronous code in JavaScript, and the choice of which one to use depends on the specific use case and personal preference

--------------------------------------------------------------------------

**30) Do you know default parameter value which is introduce ES6?**

Default parameter values are a feature introduced in ES6 (ECMAScript 2015) that allows us to set default values for function parameters if no value or undefined is passed during the function call. Here is an example:

javascript

function greet(name = 'World') {

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

}

greet(); // Output: Hello, World!

greet('Alice'); // Output: Hello, Alice!

In this example, the greet() function takes a name parameter with a default value of 'World'. If no value or undefined is passed during the function call, the default value of 'World' is used instead. If a value is passed, it is used instead of the default value.

This feature is particularly useful for functions with optional parameters, where it can be cumbersome to check for undefined values inside the function body. Here is an example of a function that accepts an optional parameter using default parameter values:

javascript

function multiply(a, b = 1) {

return a \* b;

}

console.log(multiply(5)); // Output: 5

console.log(multiply(5, 2)); // Output: 10

In this example, the multiply() function takes two parameters, a and b, with a default value of 1 for b. If no value is passed for b, the default value of 1 is used. If a value is passed, it is used instead of the default value.

Default parameter values can also be used in arrow functions:

javascript

const greet = (name = 'World') => {

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

};

greet(); // Output: Hello, World!

greet('Alice'); // Output: Hello, Alice!

In this example, the arrow function greet() takes a name parameter with a default value of 'World'. If no value or undefined is passed during the function call, the default value of 'World' is used instead. If a value is passed, it is used instead of the default value.

--------------------------------------------------------------------------

**31) What about rest and spread operator?**

The rest operator and spread operator are two related features in JavaScript that allow you to work with arrays and objects in a more flexible way.

The Rest Operator:

The rest operator is denoted by three dots ... and is used to capture multiple arguments or elements into an array. It is used in function declarations and function expressions to capture any number of arguments passed to the function. Here's an example:

javascript

function sum(...numbers) {

let result = 0;

numbers.forEach((num) => {

result += num;

});

return result;

}

console.log(sum(1, 2, 3, 4)); // Output: 10

In this example, the sum() function takes any number of arguments and the rest operator ... captures them into an array called numbers. We can then use the forEach() method to loop through the array and calculate the sum.

The Spread Operator:

The spread operator is also denoted by three dots ... and is used to expand an array or an object into individual elements. It is commonly used to concatenate or copy arrays, or to pass an array as individual arguments to a function. Here's an example:

javascript

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const arr3 = [...arr1, ...arr2];

console.log(arr3); // Output: [1, 2, 3, 4, 5, 6]

In this example, we use the spread operator ... to concatenate two arrays arr1 and arr2 into a new array arr3.

Here's an example of using the spread operator to pass an array as individual arguments to a function:

javascript

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers)); // Output: 6

In this example, we use the spread operator ... to pass an array numbers as individual arguments to the sum() function. The spread operator ... expands the array into three separate arguments 1, 2, and 3.

In summary, the rest operator allows you to capture multiple arguments into an array, while the spread operator allows you to expand an array or an object into individual elements. They are both powerful features that make working with arrays and objects in JavaScript more flexible and concise

--------------------------------------------------------------------------

**32) What do is Array includes() method?**

The includes() method in JavaScript is a built-in method of an array that is used to check if an array includes a specific element or not. It returns a boolean value true if the element is present in the array, otherwise, it returns false. The syntax of the includes() method is as follows:

javascript

array.includes(element, start)

where element is the element that you want to check for, and start is an optional parameter that specifies the position in the array where the search will start (the default value of start is 0).

Here's an example:

javascript

const fruits = ["apple", "banana", "mango", "orange"];

console.log(fruits.includes("apple")); // Output: true

console.log(fruits.includes("kiwi")); // Output: false

console.log(fruits.includes("mango", 2)); // Output: false

console.log(fruits.includes("orange", 2)); // Output: true

In this example, the includes() method is used to check if the array fruits includes the elements "apple", "kiwi", "mango", and "orange".

When we call fruits.includes("apple"), it returns true because "apple" is present in the array. Similarly, when we call fruits.includes("kiwi"), it returns false because "kiwi" is not present in the array.

The includes() method can also take an optional start parameter that specifies the position in the array where the search will start. In the above example, when we call fruits.includes("mango", 2), it returns false because the search for "mango" starts from the third element of the array (position 2) and "mango" is actually present at position 2. Similarly, when we call fruits.includes("orange", 2), it returns true because the search for "orange" starts from position 2 and "orange" is present in the array starting from position 3.

--------------------------------------------------------------------------

**33) typeof null?**

The typeof operator in JavaScript is used to determine the data type of a value. However, when the value being checked is null, the typeof operator returns 'object'. This is because null is considered to be an empty object reference. Here's an example:

javascript

const x = null;

console.log(typeof x); // Output: object

In this example, the typeof operator is used to check the data type of the variable x, which has a value of null. The output of the typeof operator is 'object'.

It is important to note that this behavior of the typeof operator for null values is considered to be a bug in JavaScript, and it is not expected to change for backwards compatibility reasons. If you want to specifically check for null values, you can use the === operator, like this:

javascript

const x = null;

console.log(x === null); // Output: true

In this example, the === operator is used to check if the value of x is null. The output of the expression is true because x is indeed null.

--------------------------------------------------------------------------

**34) Can we update the value of const?**

No, we cannot update the value of a const variable in JavaScript once it has been initialized. A const variable is a read-only variable that cannot be reassigned or modified after it has been declared. Here's an example:

javascript

const x = 5;

x = 10; // This will throw a TypeError: Assignment to constant variable.

In this example, the const keyword is used to declare a variable x with a value of 5. When we try to update the value of x to 10, we get a TypeError because we are trying to modify the value of a const variable.

However, it is important to note that if the const variable is an object or an array, the properties or elements of the object or array can be modified. This is because the const keyword only prevents the variable from being reassigned or modified as a whole, but not the individual properties or elements. Here's an example:

javascript

const person = {

name: "John",

age: 30,

};

person.age = 40; // This is allowed.

console.log(person); // Output: {name: "John", age: 40}

In this example, the const keyword is used to declare a variable person that is an object with properties name and age. Although we cannot reassign the value of person to a new object, we can modify the properties of the object. When we modify the value of the age property, it is allowed and the value of person is updated to {name: "John", age: 40}.

--------------------------------------------------------------------------

**35) Do know about event delegation and event bubbling? And how we can resolve it?**

Event delegation and event bubbling are two related concepts in JavaScript that deal with how events are handled by the browser.

Event delegation is a technique where instead of attaching an event listener to each individual element, you attach a single event listener to a parent element that will handle events for all its descendants. This can be useful for performance reasons, as well as for dynamically created content that may not exist when the page loads.

Event bubbling is the natural behavior of events in the browser, where an event that is triggered on a child element will "bubble" up to its parent elements, triggering any event listeners on those elements along the way.

Here's an example to illustrate how event delegation works:

HTML:

php

<ul id="myList">

<li>Item 1</li>

<li>Item 2</li>

<li>Item 3</li>

</ul>

JavaScript using event delegation:

javascript

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

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

if (event.target.tagName === 'LI') {

console.log('You clicked on item:', event.target.textContent);

}

});

In this example, instead of adding a click event listener to each li element, we added a single event listener to the ul element. When a li element is clicked, the event bubbles up to the ul element, which checks whether the clicked element is an li. If it is, the click event is handled by the callback function, which logs the clicked item's text content to the console.

To prevent event bubbling, you can use the event.stopPropagation() method in your event listener. For example:

javascript

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

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

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

event.stopPropagation();

console.log('Button clicked');

});

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

console.log('Div clicked');

});

In this example, if you click the button, only the button's click event handler will be called, and the div's click event handler will not be called due to event.stopPropagation()

-----------------------

event bubbling?

Event bubbling is a natural behavior of events in the browser, where an event that is triggered on a child element will "bubble" up to its parent elements, triggering any event listeners on those elements along the way.

Here's an example to illustrate event bubbling in JavaScript:

HTML:

bash

<div id="outer">

<div id="inner">

Click me!

</div>

</div>

JavaScript:

javascript

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

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

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

console.log('Outer element clicked');

});

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

console.log('Inner element clicked');

});

In this example, if you click on the "Click me!" text, both the inner and outer event handlers will be called, in that order. This is because the click event first triggers on the inner element, and then "bubbles" up to the outer element.

You can use the event.stopPropagation() method to prevent event bubbling if you only want to handle the event on the inner element. For example:

javascript

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

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

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

console.log('Outer element clicked');

});

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

console.log('Inner element clicked');

event.stopPropagation();

});

In this modified example, if you click on the "Click me!" text, only the inner event handler will be called, as event.stopPropagation() prevents the event from bubbling up to the outer element

-------------------------------------------------------------------------------------------------------

**36) What is =, ==,===, in JavaScript?**

What is = in JavaScript?

Equal to (=) is an assignment operator, which sets the variable on the left of the = to the value of the expression that is on its right. This operator assigns lvalue to rvalue.

Coding:

In the below program, there are two variables “a” and “b”. We are adding and printing their values using a third variable, “c”. The sum of the value of variable “a” and “b” is 7. Therefore, the output is 7.

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Operators</h2>

<p>a = 2, b = 5, calculate c = a + b, and display c:</p>

<p id="demonstration"></p>

<script>

var a = 2;

var b = 5;

var c= a + b;

document.getElementById("demonstration").innerHTML = c;

</script>

</body>

</html>

----------------------------------------

What is == in JavaScript?

Double equals (==) is a comparison operator, which transforms the operands having the same type before comparison.

So, when you compare string with a number, JavaScript converts any string to a number. An empty string is always converts to zero. A string with no numeric value is converts to NaN (Not a Number), which returns false.

Coding;

In the below program, we have declared one variable “a” having value 10. Lastly, the statement a == 20 returns false as the value of a is 10.

<!DOCTYPE html>

<html>

<body>

<p id="demonstration"></p>

<script>

var a = 10;

document.getElementById("demonstration").innerHTML = (a == 20);

</script>

</body>

</html>

--------------------------------------------------------------

What is === in JavaScript?

=== (Triple equals) is a strict equality comparison operator in JavaScript, which returns false for the values which are not of a similar type. This operator performs type casting for equality. If we compare 2 with “2” using ===, then it will return a false value.

Coding:

In the below program, the value of variable x is 10. It is compared to 10 written in double-quotes, which is considered as a string, and therefore, the values are not strictly the same. The output of the program is false.

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var x = 10;

document.getElementById("demo").innerHTML = (x === "10");

</script>

</body>

</html>

--------------------------------------------------------------------------

**37) What is a window object?(My)**

In JavaScript, the window object is a global object that represents the browser window. It contains properties and methods that are available globally in the browser's JavaScript environment.

Here's an example of how to use the window object to create a new browser window:

javascript

// Open a new browser window

const newWindow = window.open('https://www.example.com', '\_blank', 'width=600,height=400');

// Check if the window is closed

if (newWindow.closed) {

console.log('The window was closed');

} else {

console.log('The window is open');

}

// Close the window

newWindow.close();

In this example, we use the window object to open a new browser window using the open() method. The first parameter is the URL of the page to open, the second parameter is the name of the window, and the third parameter is a string of window features such as the width and height.

We then use the closed property of the window object to check if the window has been closed. If it is closed, we log a message to the console. If it is still open, we log a different message and then close the window using the close() method.

Note that the window object is not available in non-browser environments such as Node.js, as it is specific to the browser environment.

--------------------------------------------------------------------------

**38) Can you tell me some of ES6 latest features?**

ES6 (ECMAScript 2015) introduced several new features and enhancements to JavaScript. Here are some of the notable features introduced in ES6:

1)let and const: ES6 introduced two new variable declaration keywords, let and const. let allows block-scoped variables that can be reassigned, while const allows the declaration of variables with block scope that cannot be reassigned.

2)Arrow Functions: Arrow functions provide a concise syntax for writing anonymous functions. They have implicit returns and lexical scoping of this, making them useful for shorter function expressions.

3)Template Literals: Template literals allow for easy string interpolation and multiline strings. They use backticks () instead of single or double quotes and support placeholders for variable substitution using ${}`.

4)Destructuring Assignment: Destructuring assignment allows for extracting values from arrays or objects into individual variables. It provides a concise way to assign values and access nested data structures.

5)Default Parameters: ES6 introduced default parameter values for function parameters. Default parameter values can be assigned when a parameter is undefined or not provided.

6)Rest and Spread Operators: The rest and spread operators (...) enable working with variable-length argument lists and arrays. The rest operator is used to collect multiple function arguments into an array, while the spread operator is used to unpack elements of an array or object into another array or object.

7)Promises: Promises provide a standardized way to handle asynchronous operations. They allow for better handling of callbacks and simplify the code structure when working with asynchronous tasks.

These are just a few of the many features introduced in ES6. The subsequent versions of ECMAScript have continued to add more features and enhancements to the JavaScript language.

--------------------------------------------------------------------------

**39) What is Generator function?**

A generator function is a special type of function in JavaScript that allows you to pause and resume its execution at any point. This is achieved by using the yield keyword to yield a value and pause the execution, and the next() method to resume the execution and continue with the next value.

Here is an example of a generator function in JavaScript:

javascript

function\* generateNumbers() {

let num = 0;

while (true) {

yield num++;

}

}

const numbers = generateNumbers();

console.log(numbers.next().value); // Output: 0

console.log(numbers.next().value); // Output: 1

console.log(numbers.next().value); // Output: 2

console.log(numbers.next().value); // Output: 3

In this example, the generateNumbers function is a generator function that contains a while loop that runs indefinitely. Inside the loop, the yield keyword is used to yield the current value of num and pause the execution.

When the numbers generator is created, it is initially paused at the first yield statement. When the next() method is called on the generator, the execution resumes from the point where it was paused and continues until the next yield statement is encountered, at which point it pauses again and returns the yielded value.

In this way, the generateNumbers generator can be used to generate an infinite sequence of numbers, with each call to next() returning the next value in the sequence.

--------------------------------------------------------------------------

**40) What is shallow copy and deep copy?**

In JavaScript, objects and arrays are reference types, which means that when you assign a variable to an object or an array, the variable is actually pointing to the same object or array in memory. This can cause unexpected behavior if you need to create a copy of an object or an array, because changes to the copy may also affect the original object or array.

There are two types of copying methods: shallow copy and deep copy.

Shallow copy creates a new object or array that references the same properties or elements as the original object or array. This means that changes to the copied object or array will affect the original object or array as well. Shallow copy can be achieved using the spread operator or the Object.assign() method.

Here is an example of shallow copying an array using the spread operator:

scss

const arr1 = [1, 2, 3];

const arr2 = [...arr1];

console.log(arr1); // Output: [1, 2, 3]

console.log(arr2); // Output: [1, 2, 3]

arr2.push(4);

console.log(arr1); // Output: [1, 2, 3, 4]

console.log(arr2); // Output: [1, 2, 3, 4]

In this example, the spread operator ... is used to create a new array arr2 that contains the same elements as arr1. However, arr1 and arr2 are still referencing the same underlying array in memory, so changes to arr2 will also affect arr1.

Deep copy creates a new object or array that has its own copy of all the properties or elements of the original object or array. This means that changes to the copied object or array will not affect the original object or array. Deep copy can be achieved using the JSON.parse() and JSON.stringify() methods, or third-party libraries such as lodash.

Here is an example of deep copying an object using JSON.parse() and JSON.stringify():

css

const obj1 = { name: 'John', age: 30 };

const obj2 = JSON.parse(JSON.stringify(obj1));

console.log(obj1); // Output: { name: 'John', age: 30 }

console.log(obj2); // Output: { name: 'John', age: 30 }

obj2.age = 40;

console.log(obj1); // Output: { name: 'John', age: 30 }

console.log(obj2); // Output: { name: 'John', age: 40 }

In this example, the JSON.stringify() method is used to serialize the obj1 object into a JSON string, and then the JSON.parse() method is used to parse the JSON string into a new object obj2. The resulting obj2 object is a deep copy of obj1, so changes to obj2 will not affect obj1.

--------------------------------------------------------------------------

**41) Why JavaScript is called as interpreted language?**

JavaScript is often referred to as an interpreted language because it is executed line by line, or statement by statement, without the need for a separate compilation step. Here are a few reasons why JavaScript is considered an interpreted language:

1)Execution at runtime: JavaScript code is executed directly by an interpreter or a JavaScript engine, such as V8 (used in Chrome) or SpiderMonkey (used in Firefox). The interpreter reads and executes the code line by line, translating it into machine code or bytecode on-the-fly as it encounters each statement.

2)No explicit compilation step: Unlike languages such as C++ or Java, where code is compiled into machine code or bytecode before execution, JavaScript does not require an explicit compilation step. The code can be written and executed directly without the need for compiling into an intermediate form.

3)Dynamic typing: JavaScript is a dynamically typed language, which means variable types are determined and checked at runtime rather than during compilation. This flexibility allows for interpreting and executing code dynamically based on the runtime values and operations.

4)Just-in-time (JIT) compilation: While JavaScript is often referred to as an interpreted language, modern JavaScript engines utilize JIT compilation techniques to optimize performance. These engines analyze the code at runtime and may compile frequently executed parts of the code into highly optimized machine code to improve execution speed.

It's important to note that the term "interpreted language" can be a bit misleading since modern JavaScript engines use a combination of interpretation and just-in-time compilation. This hybrid approach allows for dynamic execution and performance optimization, making JavaScript a versatile and efficient language for web development.

---------------------------------------------------------------------------------------------

**42) JavaScript is single thread language or multi thread language? Then how can manage multithread are synchronous operation?**

JavaScript is a single-threaded language, meaning it can only perform one task at a time. However, JavaScript is capable of managing multiple threads through asynchronous programming.

In JavaScript, synchronous operations are performed in a blocking manner, meaning the program will wait for each operation to complete before moving on to the next one. Asynchronous operations, on the other hand, are performed in a non-blocking manner, allowing the program to continue executing while waiting for a task to complete.

JavaScript manages multithreading through the use of the event loop. The event loop is a mechanism that allows the program to handle asynchronous operations by adding them to a queue and executing them when the thread is idle.

For example, in the case of a web browser, when a user clicks a button that triggers an AJAX request, the request is added to the event queue, and the program continues to execute. When the AJAX response is ready, it is added to the event queue, and the program executes the corresponding callback function.

This way, JavaScript can manage multiple threads of execution without actually creating additional threads, allowing it to perform concurrent tasks while still maintaining a single-threaded mode

--------------------------------------------------------------------------

**43) How to handle asynchronous operator in JavaScript?**

In JavaScript, there are multiple ways to handle asynchronous operations. Here are some common approaches:

Callbacks: Callbacks are functions passed as arguments to asynchronous functions. They are called once the asynchronous operation is complete. Callbacks can be used to handle the result or error of the operation. However, callback-based code can lead to callback hell and make code harder to read and maintain.

Promises: Promises provide a more structured and composable way to handle asynchronous operations. Promises represent the eventual completion (or failure) of an asynchronous operation and allow you to attach success and error handlers using the then() and catch() methods respectively. Promises can be chained together and offer better error handling and code organization compared to callbacks.

Async/await: Async/await is a more recent addition to JavaScript and provides a syntax that makes asynchronous code look more like synchronous code. The async keyword is used to declare an asynchronous function, and the await keyword is used to pause the execution of an asynchronous function until a Promise is resolved. Async/await builds on top of Promises and provides a cleaner and more readable way to handle asynchronous operations.

Here's an example of how to handle asynchronous operations using Promises and async/await:

Using Promises:

javascript

function fetchData() {

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

// Perform asynchronous operation

// Call resolve(value) if successful or reject(error) if there's an error

});

}

fetchData()

.then(result => {

// Handle successful result

})

.catch(error => {

// Handle error

});

Using async/await:

javascript

async function fetchData() {

try {

// Perform asynchronous operation

// Return the result if successful

} catch (error) {

// Throw an error if there's an error

}

}

(async () => {

try {

const result = await fetchData();

// Handle successful result

} catch (error) {

// Handle error

}

})();

Both Promises and async/await provide ways to handle asynchronous operations in a more organized and readable manner compared to traditional callbacks. Choose the approach that suits your project and coding style best.

--------------------------------------------------------------------------

**44) Why we need arrow function?**

Arrow functions were introduced in ES6 as a shorthand way to write function expressions. They are especially useful for writing smaller, simpler functions, and they have a number of advantages over traditional function expressions.

One reason why arrow functions are so useful is that they have a more concise syntax compared to traditional functions. For example, an arrow function to calculate the square of a number can be written like this:

javascript

const square = (num) => num \* num;

This is much shorter and more readable than the equivalent function expression:

javascript

const square = function(num) {

return num \* num;

}

Another advantage of arrow functions is that they have a lexical this binding, which means that they don't create their own this value. This can be especially useful in event handlers and callbacks, where the value of this can often be unpredictable. Here's an example:

javascript

class Button {

constructor() {

this.element = document.querySelector('button');

this.element.addEventListener('click', () => {

this.handleClick();

});

}

handleClick() {

console.log('Button clicked');

}

}

In this example, the arrow function passed to addEventListener is used to preserve the value of this in the Button class instance, rather than creating a new this value.

Overall, arrow functions can make your code shorter, more concise, and easier to read, while also providing useful features like lexical this binding.

--------------------------------------------------------------------------

**45) Can hoisting work for function?**

Yes, hoisting works for function declarations in JavaScript. When a function is declared using the function keyword, its entire definition is hoisted to the top of its containing scope. This means that the function can be called from anywhere in the code, even before its actual definition.

Here's an example:

scss

foo();

function foo() {

console.log('Hello, world!');

}

In this example, the foo function is called before its definition, but because the function declaration is hoisted to the top of the scope, the code will run without any errors.

It's worth noting that hoisting only applies to function declarations, not function expressions. Function expressions are only evaluated when the program reaches the line where they are defined, so they cannot be called before they are defined. Here's an example:

javascript

bar(); // Error: bar is not a function

const bar = function() {

console.log('Hello, world!');

}

In this example, the bar function is defined using a function expression, which means that it cannot be called before it is defined. If we try to call bar before its definition, we'll get an error.

--------------------------------------------------------------------------

**46) What is the use Strict in javascript?**

In JavaScript, 'use strict' is a directive that can be used to enable a stricter mode of interpreting and executing the code. It enforces stricter parsing and error handling rules, disallows certain language features that could be considered dangerous or error-prone, and generally makes the code more reliable and easier to debug.

Here's an example of how to use 'use strict':

javascript

'use strict';

function doSomething() {

x = 10; // This will throw an error in strict mode, because "x" is not declared

}

doSomething();

In this example, the 'use strict' directive is used at the beginning of the script to enable strict mode. Then, we define a function called doSomething that tries to assign a value to a variable called x without declaring it first. In non-strict mode, this would silently create a global variable x, but in strict mode, it throws an error because undeclared variables are not allowed.

By using 'use strict', we can catch and prevent certain types of errors that would otherwise go unnoticed, and write more reliable and robust code

--------------------------------------------------------------------------

**47) What is different between json and object?**

In JavaScript, an object is a collection of key-value pairs, where the keys are strings and the values can be of any type, including other objects. Objects are a fundamental data structure in JavaScript and can be created and manipulated using various syntax and methods.

JSON (JavaScript Object Notation), on the other hand, is a lightweight data interchange format that is used to serialize and transmit structured data over a network. It is based on a subset of the JavaScript object literal syntax, but it is more restricted and standardized to ensure compatibility between different systems.

Here's an example of an object in JavaScript:

css

let person = {

name: 'John',

age: 30,

address: {

street: '123 Main St',

city: 'Anytown',

state: 'CA'

}

};

In this example, we define an object called person with three properties: name, age, and address. The address property is itself an object with three sub-properties: street, city, and state.

Here's an example of JSON data:

json

{

"name": "John",

"age": 30,

"address": {

"street": "123 Main St",

"city": "Anytown",

"state": "CA"

}

}

In this example, we have the same data as before, but in JSON format. The syntax is similar to JavaScript object literals, but with some key differences, such as the use of double quotes for property names and the absence of functions or methods.

One key difference between objects and JSON is that JSON is a string-based format that can be easily transmitted and parsed by different systems, while objects are a native data structure in JavaScript that can be used directly in code. Additionally, JSON has stricter syntax rules and does not support certain features of JavaScript objects, such as functions or circular references

--------------------------------------------------------------------------

**48) What is the javascript primitive and non primitive?**

In JavaScript, there are two types of data types: primitive and non-primitive.

Primitive data types are immutable data types and they are stored directly in the memory. They include:

Number: represents numerical data.

Example:

bash

let num = 10;

String: represents textual data.

Example:

bash

let str = "Hello World";

Boolean: represents true or false values.

Example:

csharp

let bool = true;

Undefined: represents undefined values.

Example:

javascript

let x;

console.log(x); // output: undefined

Null: represents null or non-existent values.

Example:

javascript

let y = null;

console.log(y); // output: null

Symbol: represents a unique identifier.

Example:

javascript

let sym = Symbol("foo");

Non-primitive data types are mutable data types and they are not stored directly in the memory. They include:

Object: represents a collection of key-value pairs.

Example:

bash

let obj = { name: "John", age: 30 };

Array: represents a collection of values stored in a specific order.

Example:

bash

let arr = [1, 2, 3, 4];

Function: represents a block of code that can be called at any time.

Example:

javascript

function greet(name) {

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

}

greet("John"); // output: Hello, John!

--------------------------------------------------------------------------

**49) Do you know Armstrong number?**

An Armstrong number is a number whose sum of its own digits raised to the power of the number of digits is equal to the original number itself. For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153.

Here's an example of how to check if a number is an Armstrong number in JavaScript:

javascript

function isArmstrongNumber(num) {

// Convert number to string to calculate length

const numStr = num.toString();

// Calculate the sum of each digit raised to the power of the length

let sum = 0;

for (let i = 0; i < numStr.length; i++) {

sum += Math.pow(parseInt(numStr[i]), numStr.length);

}

// Check if the sum is equal to the original number

return sum === num;

}

// Example usage

console.log(isArmstrongNumber(153)); // Output: true

console.log(isArmstrongNumber(371)); // Output: true

console.log(isArmstrongNumber(9474)); // Output: true

console.log(isArmstrongNumber(8208)); // Output: true

console.log(isArmstrongNumber(1634)); // Output: true

console.log(isArmstrongNumber(123)); // Output: false

console.log(isArmstrongNumber(163)); // Output: false

--------------------------------------------------------------------------

**50) Destructuring in ES6?**

Destructuring is a feature introduced in ES6 that allows developers to extract properties or elements from objects and arrays and assign them to variables. Here is an example of destructuring an object:

arduino

const person = {

name: 'John Doe',

age: 30,

address: {

street: '123 Main St',

city: 'Anytown',

state: 'CA',

zip: '12345'

}

};

const { name, age, address: { city } } = person;

console.log(name); // Output: John Doe

console.log(age); // Output: 30

console.log(city); // Output: Anytown

In this example, we have an object person with properties name, age, and address. Using destructuring, we extract the values of the name, age, and city properties and assign them to variables with the same name. We also use destructuring to extract the city property from the address object.

We can also use destructuring with arrays. Here is an example:

arduino

const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log(first); // Output: 1

console.log(second); // Output: 2

console.log(rest); // Output: [3, 4, 5]

In this example, we have an array numbers with five elements. Using destructuring, we extract the first two elements and assign them to variables first and second. We also use destructuring to extract the rest of the elements and assign them to a variable rest using the rest operator (...).

--------------------------------------------------------------------------

**51) Can you tell me about what is the different between value type and reference type in javascript?**

In JavaScript, variables can hold two different types of values: primitive types and reference types.

Primitive types include:

numbers

strings

booleans

null

undefined

symbols (added in ES6)

Reference types include:

objects

arrays

functions

The main difference between value types and reference types is how they are stored in memory and how they are compared.

Value types are stored in memory by value. When you assign a value type to a variable, the value is copied into the variable. So if you assign a value type to another variable, a new copy of the value is created. When you compare value types, you are comparing their values.

Here's an example:

javascript

let x = 10;

let y = x; // y is a copy of x

y = 20;

console.log(x); // 10

console.log(y); // 20

Reference types, on the other hand, are stored in memory by reference. When you assign a reference type to a variable, the variable holds a reference to the memory location where the object is stored. So if you assign a reference type to another variable, both variables reference the same object in memory. When you compare reference types, you are comparing their references.

Here's an example:

javascript

let arr1 = [1, 2, 3];

let arr2 = arr1; // arr2 references the same memory location as arr1

arr2.push(4);

console.log(arr1); // [1, 2, 3, 4]

console.log(arr2); // [1, 2, 3, 4]

In this example, arr2 references the same memory location as arr1. When we push a value to arr2, it also affects arr1, because they both reference the same memory location.

It's important to keep the difference between value types and reference types in mind when working with JavaScript, as it can affect how you write your code and how you approach certain problems.

--------------------------------------------------------------------------

**52) Primitive type and reference type?**

In JavaScript, there are two types of data types: primitive types and reference types.

Primitive types include:

String

Number

Boolean

Undefined

Null

Symbol (added in ECMAScript 6)

Reference types include:

Object

Array

Function

Date

RegExp

The main difference between these two types is how they are stored and accessed in memory.

Primitive types are stored directly in the variable's location in memory, whereas reference types are stored as a reference to the location in memory where the actual value is stored.

Here are some examples to illustrate the difference:

javascript

// Primitive type (string)

let name = "John";

let nameCopy = name; // Copy the value of name to nameCopy

name = "Jane";

console.log(name); // "Jane"

console.log(nameCopy); // "John"

// Reference type (array)

let numbers = [1, 2, 3];

let numbersCopy = numbers; // Copy the reference to the numbers array

numbers.push(4);

console.log(numbers); // [1, 2, 3, 4]

console.log(numbersCopy); // [1, 2, 3, 4]

In the first example, the name variable is a primitive type, so when its value is copied to nameCopy, the two variables are completely independent of each other. Changing the value of name does not affect nameCopy.

In the second example, the numbers variable is a reference type (an array), so when its value is copied to numbersCopy, only the reference to the array is copied. Both variables are pointing to the same array in memory. Therefore, when the array is modified (by pushing a new value onto it), both variables reflect the changes.

--------------------------------------------------------------------------

**53) What is higher order function in javascript?**

In JavaScript, a higher-order function is a function that takes one or more functions as arguments or returns a function as its result. Higher-order functions are used extensively in functional programming, where they are used to create abstractions and compose programs from smaller building blocks.

Here is an example of a higher-order function that takes a function as an argument:

javascript

function applyFunction(fn, x) {

return fn(x);

}

function square(x) {

return x \* x;

}

let result = applyFunction(square, 4); // result will be 16

In this example, applyFunction is a higher-order function that takes a function (fn) and a value (x) as arguments. It applies the function to the value and returns the result. The square function is passed as an argument to applyFunction, which then applies it to the value 4.

Here's another example of a higher-order function that returns a function:

javascript

function createAdder(x) {

return function(y) {

return x + y;

};

}

let add5 = createAdder(5); // add5 is now a function that adds 5 to its argument

let result = add5(3); // result will be 8

In this example, createAdder is a higher-order function that takes a value (x) as an argument and returns a new function that takes another value (y) as an argument and returns the sum of x and y. createAdder is used to create a new function add5, which adds 5 to its argument. The add5 function is then called with the argument 3, which returns the result 8

--------------------------------------------------------------------------

**54) What are the differences between null and undefined?**

In JavaScript, null and undefined are two different things.

undefined means a variable has been declared but has not been assigned a value, or it hasn't been declared at all. For example:

javascript

let num;

console.log(num); // Output: undefined

console.log(age); // Output: Uncaught ReferenceError: age is not defined

In the first example, we declared a variable num but didn't assign any value to it. So, when we try to log the value of num, it returns undefined.

In the second example, we tried to log the value of a variable age that was never declared. This throws a reference error, saying that age is not defined.

null is a value that represents the intentional absence of any object value. It is often used to indicate that a variable has no value or that a function should return no value. For example:

javascript

let num = null;

console.log(num); // Output: null

In this example, we assigned null to the variable num. So, when we try to log the value of num, it returns null.

So, the key difference between null and undefined is that null is an intentional absence of any object value, whereas undefined means a variable has been declared but has not been assigned a value, or it hasn't been declared at all.

--------------------------------------------------------------------------

**55) What is Default parameters?**

In JavaScript, default parameters allow us to specify default values for function parameters in case they are not provided during the function call. This is a feature introduced in ECMAScript 2015 (ES6).

Here's an example that demonstrates the use of default parameters:

javascript

function greet(name = "World") {

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

}

greet(); // Output: Hello, World!

greet("John"); // Output: Hello, John!

In this example, we have a function greet that takes a name parameter. We have set the default value of name to "World" using the = operator.

When we call the greet function without any arguments, it uses the default value and prints "Hello, World!" to the console. If we call the function with an argument, it uses the passed argument and prints "Hello, John!" to the console.

We can also use expressions as default parameters:

css

function multiply(a, b = a \* 2) {

return a \* b;

}

console.log(multiply(5)); // Output: 50

console.log(multiply(5, 10)); // Output: 50

In this example, the multiply function takes two parameters a and b. The default value of b is a \* 2.

When we call the multiply function with only one argument, the default value of b is used and the function returns a \* (a \* 2), which is 50. When we call the function with two arguments, the second argument overrides the default value and the function returns a \* b, which is also 50 in this case.

--------------------------------------------------------------------------

**56) Destructuring assignment?**

Destructuring assignment is a feature introduced in ES6 that allows you to extract values from objects or arrays and assign them to variables in a more concise and readable way. Here are some examples:

Destructuring an object:

arduino

const person = { name: "John", age: 30, city: "New York" };

const { name, age, city } = person;

console.log(name); // "John"

console.log(age); // 30

console.log(city); // "New York"

Destructuring an array:

arduino

const numbers = [1, 2, 3, 4, 5];

const [first, second, ...rest] = numbers;

console.log(first); // 1

console.log(second); // 2

console.log(rest); // [3, 4, 5]

Destructuring function parameters:

javascript

function getFullName({ firstName, lastName }) {

return `${firstName} ${lastName}`;

}

const person = { firstName: "John", lastName: "Doe" };

console.log(getFullName(person)); // "John Doe"

In the above example, we are destructuring the person object parameter to get its firstName and lastName properties.

Destructuring can also be used to assign default values to variables if the value being destructured is undefined:

arduino

const person = { name: "John", age: 30 };

const { name, age, city = "New York" } = person;

console.log(city); // "New York"

In the above example, since the city property is not present in the person object, the default value of "New York" is assigned to the city variable.

-------------------------------------------------------------------------------------------------------**>>>>>>>>>>>>>>>>>>>>>>> End >>>>>>>>>>>>>>>>>>>>>>>>>**

-------------------------------------------------------------------------------------------------------