What is javaScript .

JavaScript is a single-threaded, synchronous programming and scripting language widely used in web development.

The **event loop** in JavaScript is a mechanism that allows JavaScript to perform non-blocking, asynchronous operations despite being single-threaded (executing one piece of code at a time). It helps JavaScript handle multiple tasks, like responding to user actions or making network requests, without getting stuck waiting for one operation to complete before moving to the next.

Call stack:

The call stack is where JavaScript keeps track of function executions. Whenever a function is invoked, it gets pushed onto the call stack, and when the function completes, it is popped off the stack.

Web APIs:

- For tasks that take time (like timers, network requests, or reading files), JavaScript uses Web APIs provided by the browser or the environment (like Node.js). These APIs handle the task in the background.

Callback Queue (or Task Queue): Once an asynchronous operation is complete, its callback function is moved to the callback queue. This is a queue of functions that are waiting to be executed.

Event Loop:

- The event loop continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first function from the callback queue and pushes it onto the call stack for execution.

Key Points

- **Microtasks** have a higher priority than macrotasks. They run immediately after the current script execution completes and before any macrotasks.
- **Microtasks** can be created using Promises, **queueMicrotask()**, or APIs like **MutationObserver.**
- The event loop ensures all microtasks are completed before moving on to the next macrotask, which helps maintain a responsive and smooth-running application.

In Simple Terms

- **Microtasks** are quick, high-priority tasks that run after the current code finishes but before any other larger tasks.
- If both a microtask and a macrotask are waiting, the microtask will always run first.

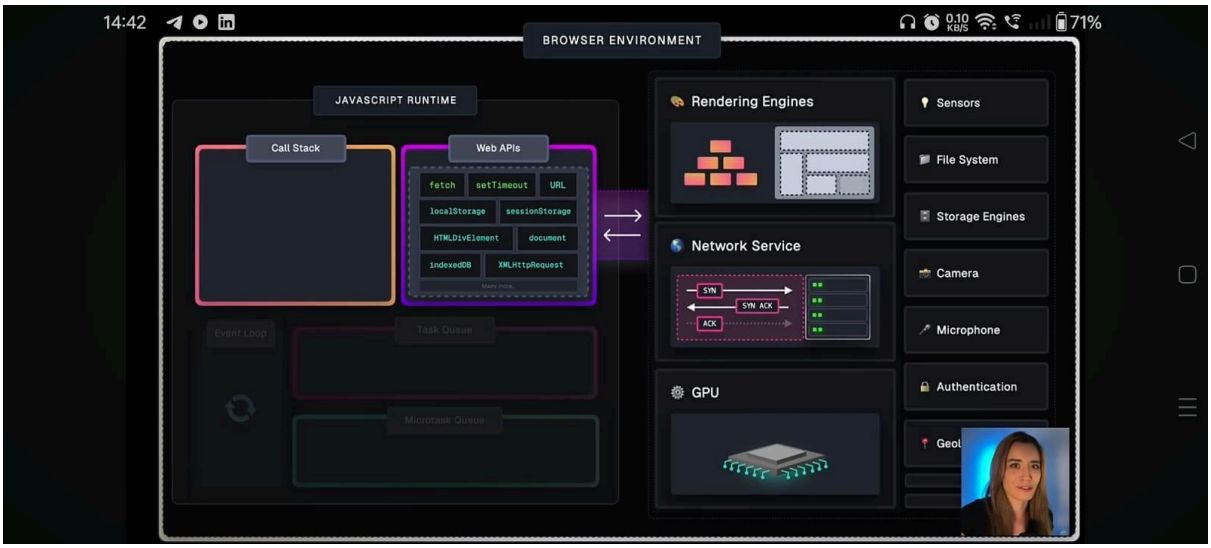
```
fetch("...")
  .then(res => ...)
```

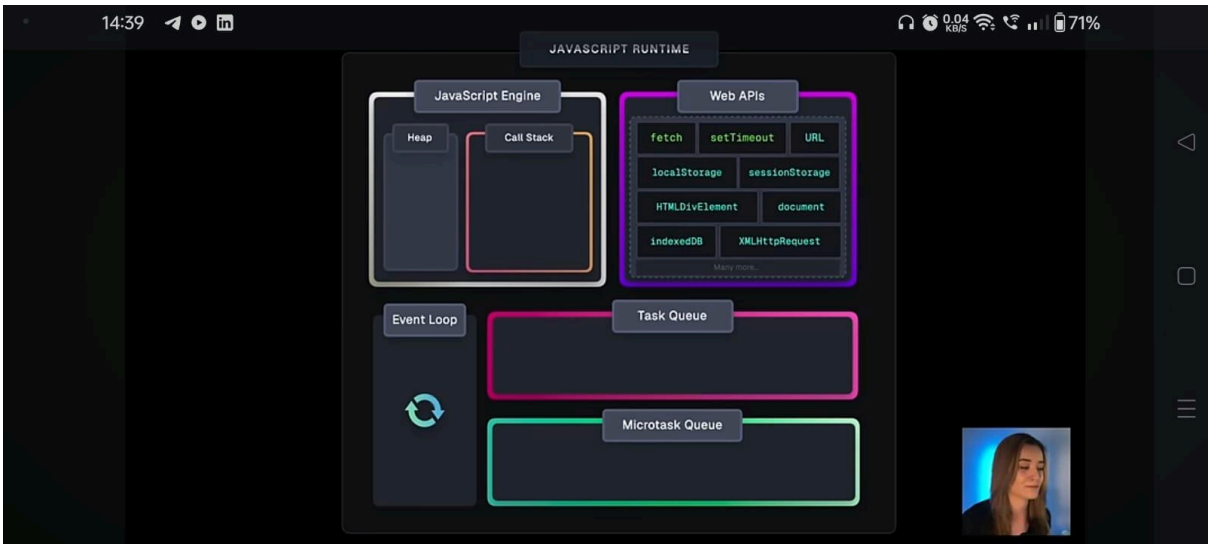
```
const [fileHandle] = await window.showOpenFilePicker();
const file = await fileHandle.getFile();
```

```
navigator.geolocation.getCurrentPosition(
  position => console.log(position),
  error => console.error(error)
)
```

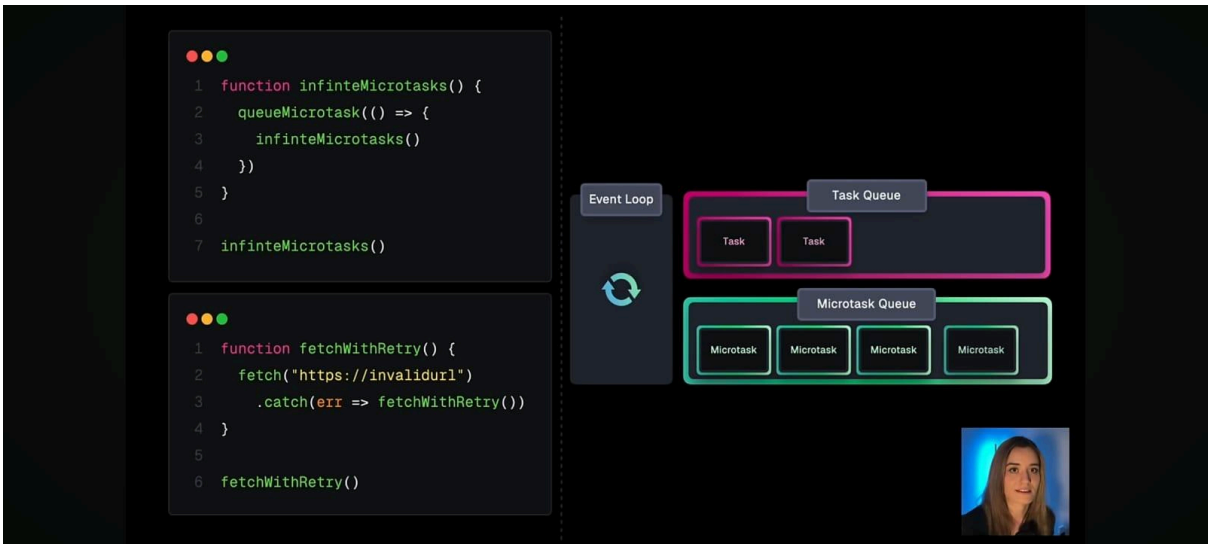
```
const request = indexedDB.open("myDb");

request.onsuccess = event => {
  console.log(event)
}

request.onerror = error => {
  console.log(error)
}
```

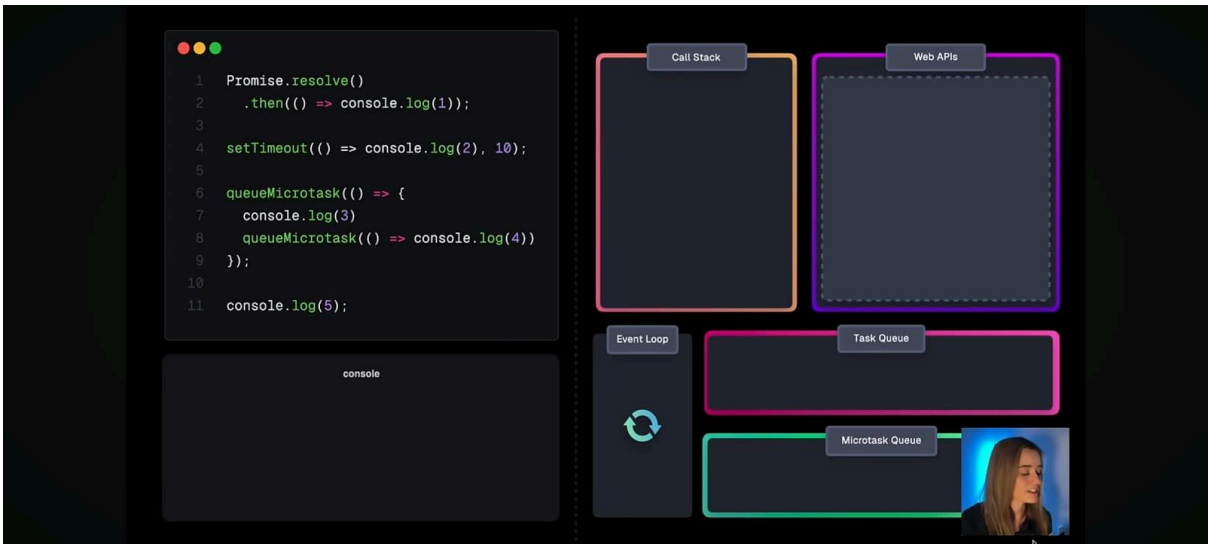
```
setTimeout(() => console.log("Done"), 2000)
```

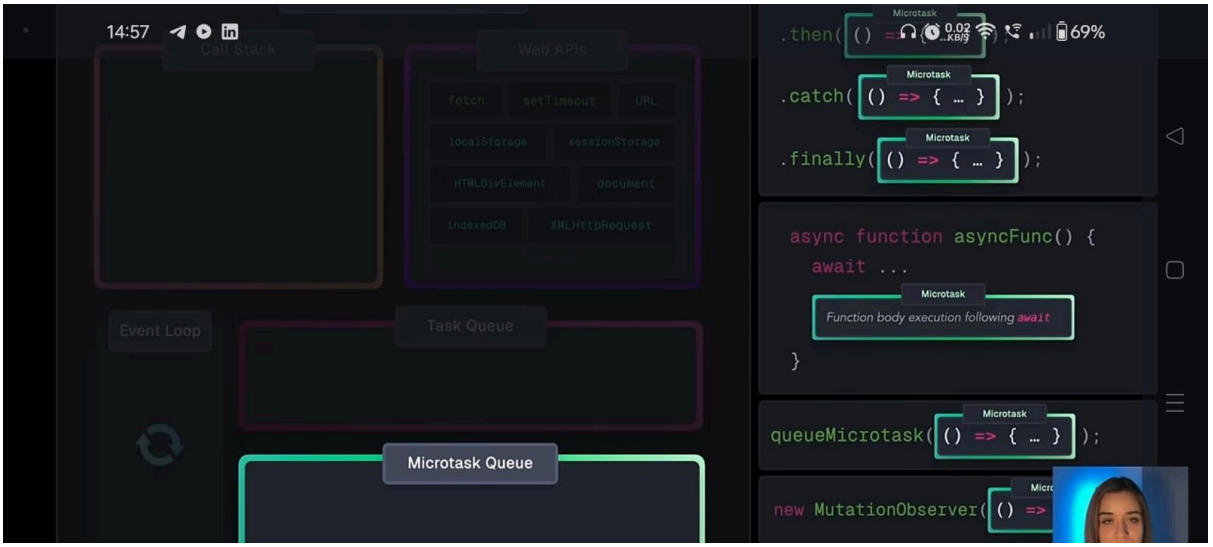




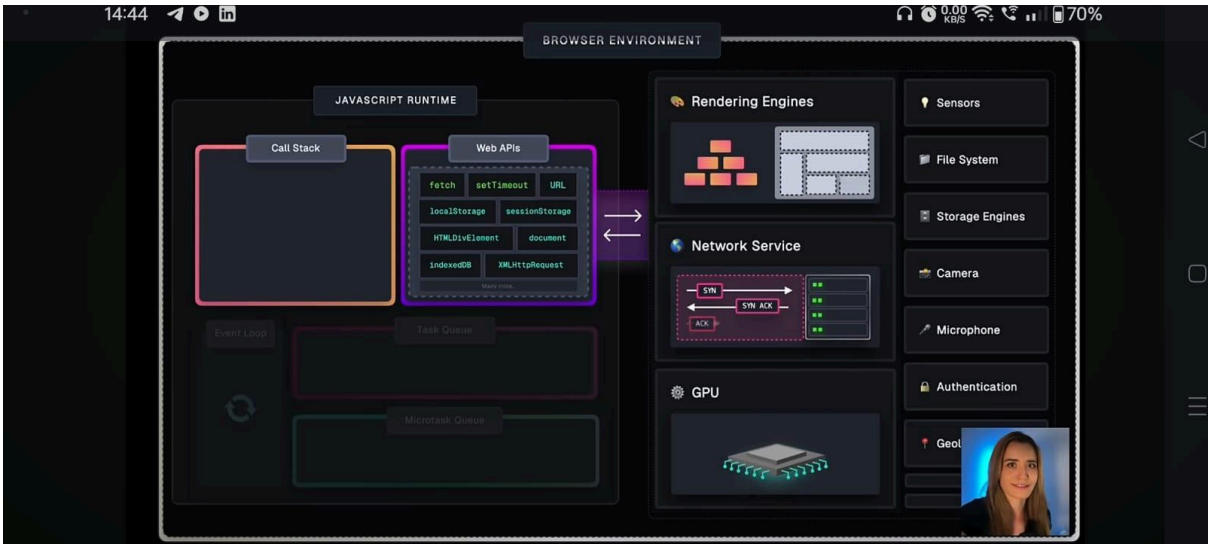

Call Stack
Web APIs
fetch
setTimeout
URL
localStorage
sessionStorage
HTMLDivElement
document
indexedDB
XMLHttpRequest
Event Loop
Task Queue
Microtask Queue


function infinteMicrotasks() {
queueMicrotask(() => {
infinteMicrotasks()
})
}
infinteMicrotasks()
function fetchWithRetry() {
fetch("https://invalidurl")
.catch(err => fetchWithRetry())
}
fetchWithRetry()
Event Loop
Task Queue
Task
Task
Microtask Queue
Microtask
Microtask
Microtask
Microtask


Promise.resolve()
.then(() => console.log(1));
setTimeout(() => console.log(2), 10);
queueMicrotask(() => {
console.log(3)
queueMicrotask(() => console.log(4))
});
console.log(5);
console
Call Stack
Web APIs
Event Loop
Task Queue
Microtask Queue

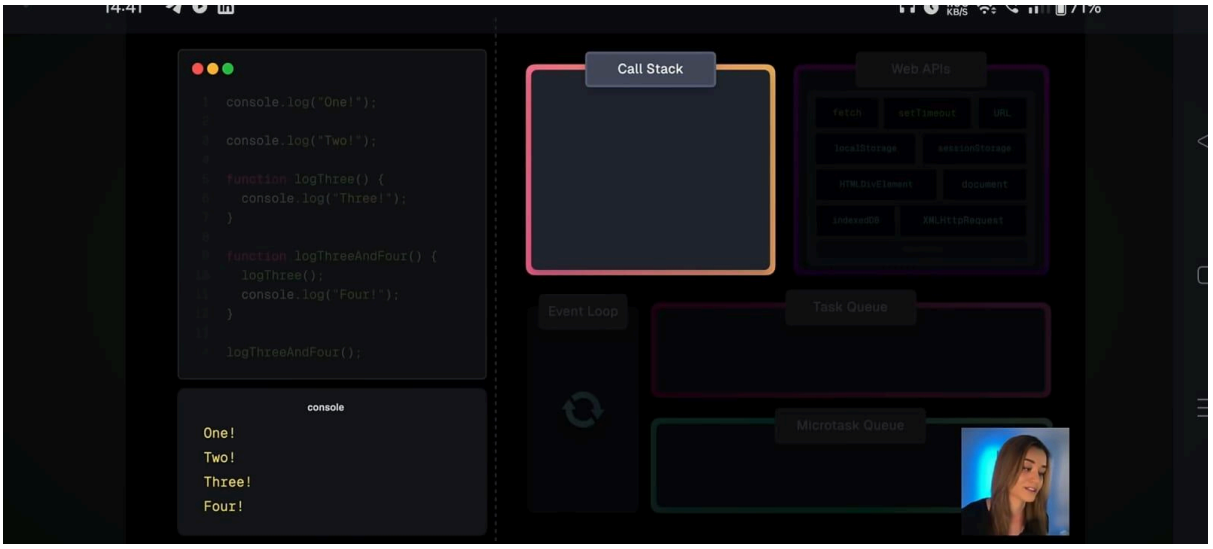

14:57
Call Stack
Web APIs
fetch
setTimeout
URL
localStorage
sessionStorage
HTMLDivElement
document
indexedDB
XMLHttpRequest
Event Loop
Task Queue
Microtask Queue
.then(() => { … });
Microtask
.catch(() => { … });
.finally(() => { … });
async function asyncFunc() {
await ...
Function body execution following await
}
queueMicrotask(() => { … });
new MutationObserver(() =>


JavaScript Visualized
CALLBACKS, FOR EXAMPLE:
navigator.geolocation.getCurrentPosition(
position => console.log(position),
error => console.error(error)
)
setTimeout(() => console.log("Done"), 2000)
const request = indexedDB.open("myDb");
request.onsuccess = event => {
console.log(event)
}
request.onerror = error => {
console.log(error)
}
PROMISES, FOR EXAMPLE:
fetch("...")
.then(res => ...)
const [fileHandle] = await window.showOpenFilePicker();
const file = await fileHandle.getFile();


14:44
70%
BROWSER ENVIRONMENT
JAVASCRIPT RUNTIME
Call Stack
Web APIs
fetch
setTimeout
URL
localStorage
sessionStorage
HTMLDivElement
document
indexedDB
XMLHttpRequest
Event Loop
Task Queue
Microtask Queue
Rendering Engines
Network Service
SYN
SYN ACK
ACK
GPU
Sensors
File System
Storage Engines
Camera
Microphone
Authentication


14:44
70%
Web APIs
Fetch API
Enables fetching resources
fetch
Timers API
Enables scheduling of timed or repeating tasks
setTimeout
setInterval
...
Console API
Enables logging output
console
Geolocation API
Allows the user to provide their location to web applications
Geolocation
GeolocationPosition
...
Web Storage API
Provides mechanisms to store key/value pairs
localStorage
sessionStorage
...
File API
Facilitates reading and interacting with files
Blob
File
FileList
...
Performance API
Standards used to measure the performance of web applications
Performance
PerformanceMeasure
...
HTML DOM
Interfaces that define the functionality of each of the elements in HTML
HTMLElement
HTMLLIElement
HTMLDivElement
...
URL API
Enables URL handling
URL
...
Many more...


14:43
70%
Geolocation API
Allows the user to provide their location to web applications
navigator.geolocation.getCurrentPosition(
successCallback
position => console.log(position)
,
errorCallback
error => console.error(error)
)


console.log("One!");
console.log("Two!");
function logThree() {
console.log("Three!");
}
function logThreeAndFour() {
logThree();
console.log("Four!");
}
logThreeAndFour();
console
One!
Two!
Three!
Four!
Call Stack
Web APIs
Event Loop
Task Queue
Microtask Queue

Prototypes

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

__proto__ Property: Every JavaScript object has an internal **[[Prototype]]** (which is often accessible via the **__proto__** property). This property points to the prototype of the object, which is usually the prototype of the constructor function used to create the object. For example:

```
const obj = {};
console.log(obj.__proto__ === Object.prototype);
// output  true
```

Here, obj is an instance of Object, and its __proto__ points to Object.prototype.

Prototypes are the mechanism by which JavaScript objects inherit features from one another. In this article, we explain what a prototype is, how prototype chains work, and how a prototype for an object can be set.

Prerequisites:	Understanding JavaScript functions, familiarity with JavaScript basics (see First steps and Building blocks), and OOJS basics (see Introduction to objects).
Objective:	To understand JavaScript object prototypes, how prototype chains work, and how to set the prototype of an object.

The prototype chain

In the browser's console, try creating an object literal:

```
const myObject = {
    city: "Madrid",
    greet() {
      console.log(`Greetings from ${this.city}`);
    },
  };
  myObject.greet(); // Greetings from Madrid
```

This is an object with one data property, city, and one method, greet(). If you type the object's name *followed by a period* into the console, like myObject., then the console will pop up a list of all the properties available to this object. You'll see that as well as city and greet, there are lots of other properties!

__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
city
constructor
greet
hasOwnProperty
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf

Try accessing one of them:

myObject.toString(); // "[object Object]"

It works (even if it's not obvious what toString() does).

What are these extra properties, and where do they come from?

Every object in JavaScript has a built-in property, which is called its prototype. The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain. The chain ends when we reach a prototype that has null for its own prototype.

Yes, you're right! The super keyword in JavaScript is used in two primary contexts:

1. **Accessing properties or methods on an object's [[Prototype]] (in classes, the superclass).**
2. **Invoking a superclass's constructor.**

1. Using super in Classes:

In JavaScript classes, the super keyword is used to call functions on an object's parent class. This is especially useful when extending a class, allowing a subclass to access or override the properties and methods of its superclass.

Example: Calling a Superclass's Constructor

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        return `${this.name} makes a noise.`;
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Calls the constructor of the superclass (Animal)
        this.breed = breed;
    }

    speak() {
        return `${super.speak()} ${this.name} barks.`; // Calls the `speak` method from Animal
    }
}

const myDog = new Dog('Buddy', 'Golden Retriever');
console.log(myDog.speak());
// Output: "Buddy makes a noise. Buddy barks."
```

- In this example:
    - The Dog class extends the Animal class.
    - The super keyword is used in the Dog constructor to call the constructor of the superclass (Animal) to initialize the name property.
    - In the speak method, super.speak() calls the speak method of the Animal class.

2. Using super in Object Literals:

You can also use super in object literals to access methods from an object's prototype. This allows you to call methods from an object's prototype in a more concise way.

Example: Using super in Object Literals

```
const parent = {
    greet() {
        return 'Hello from the parent!';
    }
};

const child = {
    greet() {
        return `${super.greet()} Hello from the child!`;
    }
};

Object.setPrototypeOf(child, parent);

console.log(child.greet());
// Output: "Hello from the parent! Hello from the child!"
```

- Here:
    - parent has a greet method.
    - child uses super.greet() to call the greet method of its prototype (parent).

Summary:

- super is used to:
    - **Invoke a superclass's constructor**: super(arguments) in the constructor of a subclass.
    - **Access superclass methods or properties**: super.methodName() to call a method on the superclass.
- super is particularly useful for extending classes and achieving inheritance in JavaScript, and it provides a clean way to access and override methods and properties of a parent class.

Would you like more examples or details on how to use super?

Constructor.

In JavaScript, a constructor is a special function used to create and initialize objects when you create an instance of a class or a function. Constructors are mainly used in object-oriented programming to set up properties and perform any setup work needed when an object is created.

How Constructors Work

1. Constructor in a Function-Based Approach:

Before JavaScript had the class syntax (introduced in ES6), constructors were typically created using regular functions. When these functions are invoked with the new keyword, they act as constructors.

Example:

```
function Person(name, age) {
    this.name = name; // `this` refers to the new instance being created
    this.age = age;
}

const person1 = new Person('Alice', 30); // Creates a new instance of Person
console.log(person1.name); // Output: "Alice"
console.log(person1.age);  // Output: 30
```

Explanation:

Person is a constructor function.

When new Person('Alice', 30) is called:

A new empty object is created.

The this keyword inside the constructor refers to that new object.

The properties name and age are assigned to the new object.

The newly created object is returned by default.

2. Constructor in a Class-Based Approach:

With the introduction of ES6, JavaScript provides a class syntax, where a constructor is a special method used for creating and initializing an object created with that class.

Example:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hello, my name is ${this.name} and I am ${this.age} years old.`;
    }
}

const person1 = new Person('Bob', 25);
console.log(person1.greet());
```

// Output: "Hello, my name is Bob and I am 25 years old."

Explanation:

The constructor method is a special method for creating and initializing an instance of the Person class.

When new Person('Bob', 25) is called, the constructor is invoked, and this refers to the new instance being created.

The greet method is a regular method of the Person class and can be called on any instance of the class.

Key Characteristics of Constructors:

Used with the new Keyword:

Constructors are designed to be called with the new keyword to create an instance. When called with new, JavaScript does the following:

Creates a new empty object.

Sets the prototype of the new object to the constructor's prototype.

Executes the constructor function with this bound to the new object.

Returns the new object (if the constructor does not explicitly return another object).

Initializes Object Properties:

A constructor is often used to initialize the properties of an object with specific values, as seen in the Person examples above.

Return Behavior:

By default, constructors return the new instance object created. However, if a constructor function explicitly returns an object, that object will be returned instead of the newly created instance.

Why Use Constructors?

Reusability: Constructors provide a reusable way to create multiple instances of objects with the same structure and behavior.

Encapsulation: Constructors can encapsulate initialization logic, ensuring that every object created from the constructor is properly set up.

Consistency: Using constructors ensures a consistent way of creating objects with specific properties and methods.

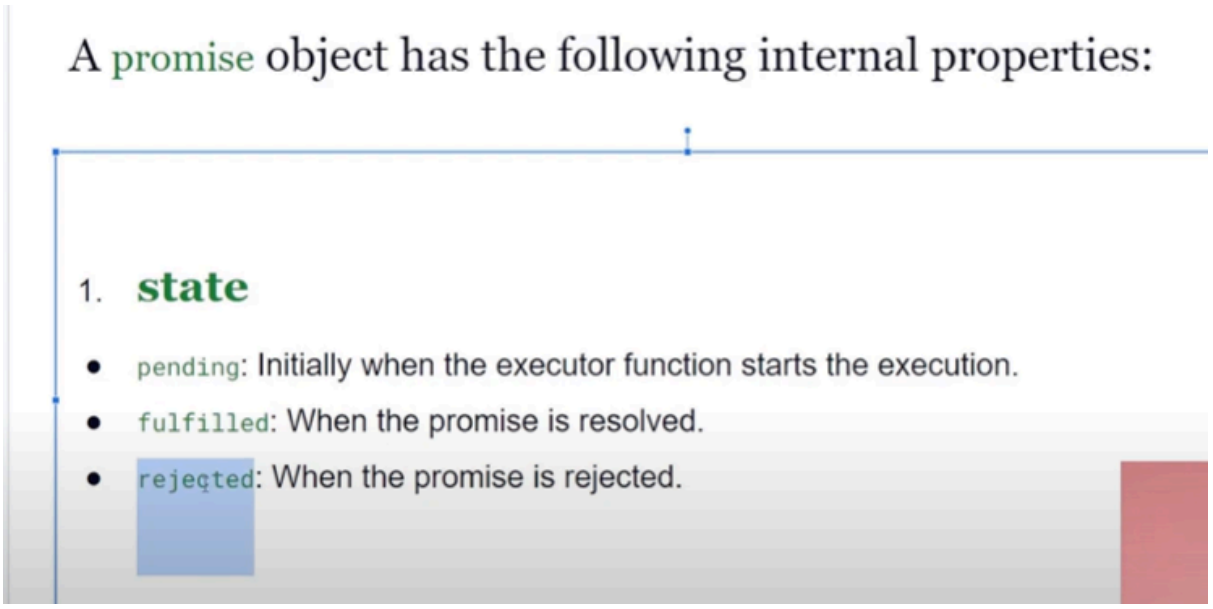
What is Promise In Javascript?

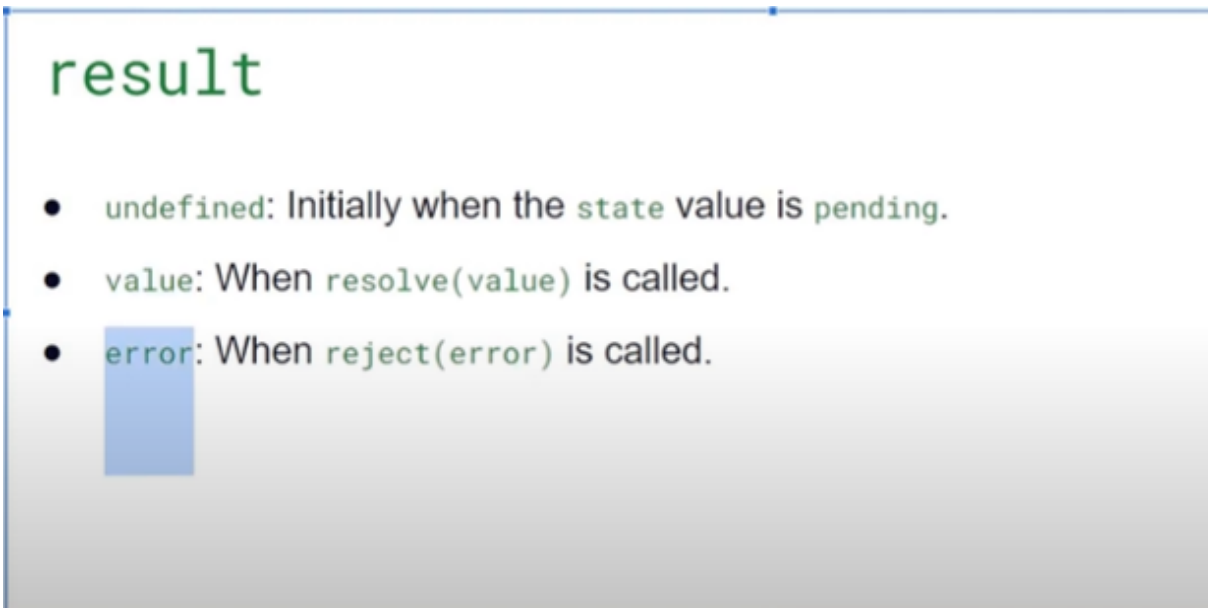
A **Promise** in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises are a way to handle asynchronous operations in a more flexible and readable way, avoiding the so-called "callback hell" (nested callbacks that can make code difficult to understand and maintain).

Key Characteristics of Promises

1. **Three States of a Promise:**
    - **Pending:** The initial state of a promise. The asynchronous operation is not yet completed.
    - **Fulfilled (Resolved):** The state when the asynchronous operation has completed successfully, and the promise has a value.
    - **Rejected:** The state when the asynchronous operation has failed, and the promise has a reason (usually an error).
2. **Immutable State Transitions:**
    - Once a promise is resolved (fulfilled or rejected), it cannot change to another state. For example, a promise that is fulfilled cannot transition to a rejected state.
3. **Chaining Promises:**

- Promises can be chained using .then() for handling resolved values and .catch() for handling errors. This allows for writing sequential asynchronous operations in a clear and readable manner.





let promise = new Promise(function (resolve,reject) {

```
    let result = true;
    if ( result ) {
        setTimeout(() => {
      resolve("You are success");
    })
    }else{
        reject("you are fail")
    }
});

console.log(promise,"promise")

 promise.then((successMessage) => {
    console.log(successMessage); // Output after 1 second: "You are success"
  })
  .catch((errorMessage) => {
    console.error(errorMessage); // Will not run in this case
  });
```

Hindi Explanation

Promise JavaScript mein ek object hai jo asynchronous operations ko manage karne ke liye istemal hota hai. Ye aapko ek aisa mechanism provide karta hai jisse aap asynchronous code ko more manageable aur readable bana sakte hain.

Promise Ka Structure

Promise ke 3 states hote hain:

1. **Pending**: Initial state, jab promise abhi complete nahi hua hai.
2. **Fulfilled**: Jab promise successful ho jata hai.
3. **Rejected**: Jab promise fail ho jata hai.

Promise Kaise Kaam Karta Hai

Promise ko create karne ke liye, aap Promise constructor ka istemal karte hain:

```
let myPromise = new Promise((resolve, reject) => {
    // Asynchronous operation
    const success = true; // ya false

    if (success) {
        resolve("Operation was successful!");
    } else {
        reject("Operation failed!");
    }
});
```

then, catch, aur finally Methods

Promises ke saath aap then, catch, aur finally methods ka istemal karte hain:

then: Ye method promise ke fulfill hone par call hota hai. Ye ek callback function ko accept karta hai jo success value ko process karta hai.
javascript
Copy code

```
myPromise
    .then(result => {
        console.log(result); // "Operation was successful!"
    });
```

1.

catch: Ye method promise ke reject hone par call hota hai. Ye error handling ke liye istemal hota hai.

```
myPromise
    .catch(error => {
        console.error(error); // "Operation failed!"
    });
```

2.

finally: Ye method promise ke fulfill ya reject hone ke baad call hota hai, chahe result kuch bhi ho. Ye cleanup actions ya final steps ke liye istemal hota hai.

```
myPromise
    .finally(() => {
        console.log("This runs after the promise is settled, regardless of its outcome.");
    });
```

3.

Complete Example

```
let myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        const success = true; // Change this to false to test rejection
        if (success) {
            resolve("Operation was successful!");
        } else {
            reject("Operation failed!");
        }
    }, 1000);
});

myPromise
    .then(result => {
        console.log(result); // "Operation was successful!"
    })
    .catch(error => {
        console.error(error); // "Operation failed!"
    })
    .finally(() => {
        console.log("Cleanup or final steps can go here.");
    });
```

Summary

- **Promise**: Ek object jo asynchronous operations ko manage karta hai.
- **States**: Pending, Fulfilled, Rejected.
- **Methods**:
    - then: Success value handle karta hai.
    - catch: Error handle karta hai.
    - finally: Promise settle hone ke baad run hota hai.

Promises ko istemal karke aap asynchronous code ko zyada structured aur readable bana sakte hain!

CallBack Function

1. What is a Callback Function?

A **callback function** is a function that is passed as an argument to another function and is intended to be executed after the completion of some operation. In JavaScript, callbacks are commonly used to handle asynchronous tasks like reading files, making network requests, or interacting with databases.

2. Why Use Callback Functions?

- **Handle Asynchronous Operations**: JavaScript is single-threaded and non-blocking, meaning it doesn't wait for long-running tasks to complete before moving on to the next task. Callbacks provide a way to run code after an asynchronous task is finished, ensuring that you don't block the main thread.
- **Maintain Control**: Callbacks give you control over the order of execution, allowing you to specify precisely what to do once an asynchronous operation completes.
- **Modular Code**: Using callbacks helps separate the "what to do next" part of your code from the "how to do it" part, making your code more modular and easier to maintain.

3. When to Use Callback Functions?

- **Asynchronous Operations**: Use callbacks whenever you deal with tasks that take time to complete, such as:
    - **Making HTTP requests** (e.g., using fetch or XMLHttpRequest).
    - **Reading or writing files** (e.g., in Node.js).
    - **Timers and intervals** (e.g., setTimeout or setInterval).
    - **Event handling** (e.g., click events, form submissions).
- **Delegating Execution**: When you want to delegate a specific operation to another function and decide what to do with the result afterward.

Alternatives to Callbacks

To address the issues of callback hell and improve code readability, JavaScript introduced **Promises** and later, **async/await** syntax.

- **Promises**: Promises allow chaining of asynchronous operations in a more readable way.
- **Async/Await**: This syntax provides an even more readable and synchronous-looking way to handle asynchronous code.

. How to Use Callback Functions?

Here's how you can use callback functions in JavaScript:

Define the Function That Takes a Callback:
Create a function that accepts another function (the callback) as a parameter:
javascript
Copy code

```
function performOperation(callback) {

    console.log("Performing some operation...");

    callback(); // Call the callback function after the operation

}
```

1.

Pass the Callback Function:
Call the function and pass another function as an argument:
javascript
Copy code

```
performOperation(function() {

    console.log("Callback function executed after the operation.");

});
```

Output:

```
Performing some operation...

Callback function executed after the operation.
```

2.

Using Callbacks in Asynchronous Functions:
Here's an example of using a callback with an asynchronous operation like setTimeout:

```
function delayedGreeting(callback) {

    setTimeout(() => {

        console.log("Hello after 2 seconds!");
```

```
        callback(); // Call the callback after the delay
    }, 2000);
}

delayedGreeting(function() {
    console.log("This runs after the greeting.");
});
```

Output:
Hello after 2 seconds!

This runs after the greeting.

3.

Summary: Callback Functions

- **What**: A function passed to another function to be executed later.
- **Why**: To handle asynchronous operations and maintain control over the order of execution.
- **When**: For any operation that is time-consuming or requires asynchronous handling.
- **How**: Define a function that takes a callback, and pass a function to be called when the operation completes.

Basic Example of a Callback Function

Here's a simple example where a callback function is used to print a message after a certain action is completed:

```javascript
function greet(name, callback) {
    console.log(`Hello, ${name}!`);
    callback(); // Call the callback function after greeting
}

function sayGoodbye() {
    console.log("Goodbye!");
}
```

Callback Hell

A common problem with callbacks is known as "callback hell," which occurs when multiple asynchronous operations are nested within each other, leading to deeply nested code that is hard to read and maintain.

Example of Callback Hell:

```javascript
asyncOperation1(function(result1) {
    asyncOperation2(result1, function(result2) {
        asyncOperation3(result2, function(result3) {
            asyncOperation4(result3, function(result4) {
                // Continue with more operations...
            });
        });
    });
});
```

- This nested structure makes the code hard to read and maintain.

Currying.

Currying is a technique where a function with multiple parameters is transformed into a sequence of functions, each accepting a single argument.

```
function multiply(a) {
  return function(b) {
      return a * b;
  };
}

const double = multiply(2);
console.log(double(5)); // 10
```

Other way

```
multiply(2)(5)
```

Why is it used?

Improved Code Reusability

Flexibility: Currying allows for partial application, making it easy to create specialized functions.
Readability: Breaking functions into simpler, smaller pieces improves code clarity.
Modularity: Facilitates function composition and reuse.
Configurability: Supports deferred execution for adaptable function use.

Explanation in Hindi

Currying ek functional programming technique hai jo JavaScript mein use hoti hai. Currying ka matlab hai ek function ko aise transform karna ki woh multiple arguments ko ek ek karke accept kare, rather than taking all arguments at once.

Concept:

- **Function Transformation:** Currying transforms a function that takes multiple arguments into a series of functions, each taking a single argument.
- **Partial Application:** It allows partial application of function arguments, meaning you can provide some arguments and get a new function that expects the remaining arguments.

How It Works:

1. **Basic Function:** A function that normally takes multiple arguments.
2. **Curried Function:** Transforms this function into a series of functions where each function takes one argument.

Example:

```
// Normal function
function multiply(a, b) {
  return a * b;
}

// Curried function
function curriedMultiply(a) {
  return function(b) {
```

```
    return a * b;
  };
}

const multiplyBy2 = curriedMultiply(2);
console.log(multiplyBy2(5)); // 10
```

Explanation:

1. **Normal Function:** multiply takes two arguments a and b and returns their product.
2. **Curried Function:** curriedMultiply takes one argument a and returns a new function that takes another argument b and returns the product of a and b.
3. **Partial Application:** multiplyBy2 is a partially applied function where a is fixed to 2. It can now be used to multiply any number by 2.

Benefits:

- **Modularity:** Currying allows for more modular and reusable code.
- **Function Composition:** It helps in composing functions and creating more specific functions from general ones.
- **Lazy Evaluation:** It supports lazy evaluation, where functions can be applied incrementally.

Currying is widely used in functional programming and can be very powerful in JavaScript for creating flexible and reusable functions.

Why Use?

1. Function Reusability:

Currying se aap ek general function ko specific functions mein convert kar sakte hain, jahan kuch arguments ko fix karke naye functions create kiye ja sakte hain.

Example:

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}

const double = multiply(2);

console.log(double(5)); // 10
```

Yahaan, double ek specialized function hai jo multiply se bana hai aur a ko 2 fix kiya gaya hai. Aap triple jaise aur functions bhi is pattern se create kar sakte hain.

2. Partial Application:

Currying partial application ki suvidha deta hai, jahan aap kuch arguments ko ab provide kar sakte hain aur baaki arguments ko baad mein de sakte hain. Yeh configuration ko incrementally allow karta hai.

Example:

```
function add(a, b, c) {
  return a + b + c;
}

const addFive = add.bind(null, 5);
console.log(addFive(10, 15)); // 30
```

Is case mein, addFive ek partially applied function hai jahan pehla argument 5 fix hai. Currying bhi similar approach provide karta hai, lekin zyada versatile hota hai.

3. Readability Improve Karna:

Curried functions se code zyada clean aur readable ban sakta hai, especially jab functions ke multiple parameters hote hain. Yeh complex functions ko simpler, single-argument functions mein break kar deta hai.

Example:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}

const add5 = x => x + 5;
const multiply2 = x => x * 2;
```

```
const addThenMultiply = compose(multiply2, add5);

console.log(addThenMultiply(3)); // 16
```

Yahaan compose ek curried function hai jo add5 aur multiply2 ko combine karta hai, jis se code modular aur readable ho jata hai.

4. Function Composition Enhance Karna:

Currying function composition ko simplify karta hai, jahan aap complex functions ko basic building blocks se create kar sakte hain. Yeh functional programming mein commonly use hota hai.

Example:

javascript

Copy code

```
function filter(predicate) {
  return function(array) {
    return array.filter(predicate);
  };
}

const isEven = x => x % 2 === 0;
const filterEvens = filter(isEven);

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

Yahaan filter curried function hai jo filterEvens banata hai jo array se even numbers filter karta hai.

5. Flexibility Enhance Karna:

Currying aapko zyada flexibility deta hai function calls mein. Aap highly specialized functions create kar sakte hain jo specific configurations ke sath use kiye ja sakte hain.

Example:

```
function greet(greeting) {
  return function(name) {
    return `${greeting}, ${name}!`;
  };
}

const sayHello = greet('Hello');
const sayHi = greet('Hi');

console.log(sayHello('Alice')); // Hello, Alice!
console.log(sayHi('Bob'));      // Hi, Bob!
```

Yahaan greet se different greeting functions create kiye gaye hain, jo currying ki flexibility ko showcase karte hain.

Summary mein, currying se aap modular, reusable, aur readable code likh sakte hain. Yeh function composition, partial application, aur function calls ki flexibility ko enhance karta hai.

Call Apply Bind method

```
const obj = {
    fname:"Naveen",
    lastName: "Yadav"
}

function greeting( profession ){
    return this.fname + " " + this.lastName + " " + profession
}
```

Call: The call() method calls a function with a given value and arguments provided individually.

```
function getName(city1, city2, city3){
   console.log(this.fname + " " + this.lname + " has lived in " + city1 + " " + city2 + " " + city3);
}

let Person = {
   fname : 'John',
   lname : 'Doe'
}

getName.call(Person, 'delhi','mumbai','bangalore');
```

Example

```
console.log(greeting.call(obj,"Software Engineer"))
```

Apply : The call() method calls a function with a given value and arguments provided array.

```
console.log(greeting.apply(obj,["Software Engineer"]))
```

Bind: The bind() method creates a new function that, when called, has its keyword set to the provided value.

Example

```
const newFunc = greeting.bind(obj)

console.log(newFunc("Software Engineer"),"Bind method")
```

Closure

English

A **closure** is a programming concept where an inner function retains access to the variables of its outer function even after the outer function has finished executing. This allows the inner function to remember and use the outer function's variables.

In Hindi:-

Closure ek programming concept hai jo JavaScript aur kai dusre programming languages mein use hota hai. Closure tab banta hai jab ek function apni outer function ke scope ke variables ko access karta hai, even though outer function apni execution complete kar chuki hoti hai.

Conceptual Explanation:

1. **Inner Function Access:** Closure ke zariye, ek inner function outer function ke variables ko access kar sakta hai, jo outer function ke scope ke part hote hain.

2. **Persistent Scope:** Closure ke zariye, variables jo outer function ke scope ka part hain, unki values ko "remember" kiya ja sakta hai, even after the outer function has finished executing.

Example:

```
function outerFunction(outerVariable) {
  return function innerFunction(innerVariable) {
    console.log("Outer Variable: " + outerVariable);
    console.log("Inner Variable: " + innerVariable);
  };
}

const closureFunction = outerFunction("I'm outside!");
closureFunction("I'm inside!");
```

Explanation:

- outerFunction ek outer function hai jo ek outerVariable ko accept karta hai aur ek inner function innerFunction return karta hai.
- innerFunction outerVariable aur innerVariable dono ko access kar sakta hai.
- Jab closureFunction ko call kiya jata hai, to outerVariable ki value ko bhi access kiya ja sakta hai, jo closure ke zariye possible hota hai.

Key Points:

- **Enclosing Scope:** Closure ensures that the inner function retains access to the variables of its enclosing (outer) function.
- **Data Encapsulation:** Closures can be used to create private variables or methods, thereby encapsulating data.

Closures are a powerful feature in JavaScript that enable various programming patterns and can be used for data encapsulation, function factories, and more.

Closures ka use programming mein kai tarikon se hota hai. Yeh powerful feature hai jo flexibility aur functionality provide karta hai. Yahan kuch key reasons hain jinke liye closures ka use kiya jata hai:

Why use?.

1. Data Encapsulation (Private Variables):

Closures ka use private variables aur methods create karne ke liye hota hai. JavaScript mein, closures ko use karke aap data ko encapsulate kar sakte hain aur variables ko private rakh sakte hain jo directly access nahi kiye ja sakte.

Example:

```
function createCounter() {
  let count = 0; // Private variable
  return function() {
    count += 1;
    return count;
  };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

Is example mein, count variable createCounter function ke scope ke andar encapsulated hai aur counter function ke through hi access aur modify kiya ja sakta hai.

2. Function Factories:

Closures ko function factories banane ke liye use kiya jata hai. Aap ek function create kar sakte hain jo specific configuration ke saath multiple functions create karta hai.

Example:

```
function makeMultiplier(multiplier) {
  return function(value) {
    return value * multiplier;
  };
}
```

```
const double = makeMultiplier(2);

const triple = makeMultiplier(3);


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

console.log(triple(5)); // 15
```

Yahaan, makeMultiplier function ek multiplier value ko accept karta hai aur ek function return karta hai jo is multiplier ke saath value ko multiply karta hai.

3. Maintaining State:

Closures ko use karke aap state ko maintain kar sakte hain across multiple function calls. Yeh especially useful hota hai jab aapko function ke state ko preserve karna hota hai.

Example:

```
function createGreeting(greeting) {
  return function(name) {
    return greeting + ', ' + name;
  };
}


const sayHello = createGreeting('Hello');

console.log(sayHello('Alice')); // Hello, Alice

console.log(sayHello('Bob')); // Hello, Bob
```

Is example mein, greeting variable createGreeting function ke closure ke part hai aur isko sayHello function ke through access kiya ja sakta hai.

4. Partial Application and Currying:

Closures ko partial application aur currying ke techniques mein bhi use kiya jata hai, jo functions ko partially apply karke re-useable functions create karne mein help karte hain.

Example:

```
function add(a) {
  return function(b) {
    return a + b;
  };
}

const addFive = add(5);
console.log(addFive(10)); // 15
```

Yahaan, add function ek partial application create karta hai jahan a value fix hoti hai aur b value ko dynamically provide kiya jata hai.

5. Asynchronous Programming:

Closures ka use asynchronous programming mein bhi hota hai, jahan functions callbacks aur promises ke saath deal karte hain aur closures ke zariye state aur variables ko manage karte hain.

Example:

```
function asyncOperation() {
  let result = 0;
  setTimeout(function() {
    result = 42;
    console.log('Async operation result:', result);
  }, 1000);
}

asyncOperation();
```

Is example mein, result variable ko setTimeout callback function ke zariye access kiya jata hai, jo closure ki madad se possible hota hai.

Closures ka use karke aap code ko modular, reusable, aur maintainable bana sakte hain, aur complex programming patterns ko simplify kar sakte hain.

Lexical Scope

Lexical scope JavaScript mein ek fundamental concept hai jo variables aur functions ke visibility aur lifetime ko define karta hai. Lexical scope ka matlab hai ki ek variable ka scope uski position ke basis par determine hota hai code ke structure mein. Iska matlab hai ki functions apne enclosing (outer) scopes ke variables ko access kar sakti hain, lekin outer functions apne inner (nested) scopes ke variables ko access nahi kar sakti.

Concept:

1. **Scope Chain:**
   - Lexical scope ek scope chain create karta hai, jahan ek function apne outer (enclosing) function ke variables ko access kar sakti hai. Yeh chain outer function se le kar global scope tak extend hoti hai.
2. **Nested Functions:**
   - Agar aap ek function ke andar ek aur function define karte hain, to inner function apne outer function ke variables ko access kar sakta hai.

Example:
javascript
Copy code

```
function outerFunction() {

  let outerVariable = 'I am outside!';



  function innerFunction() {

    console.log(outerVariable); // 'I am outside!'

  }



  innerFunction();

}



outerFunction();
```

3. Yahaan, innerFunction outerFunction ke outerVariable ko access kar sakti hai, kyunki inner function lexical scope ke basis par outer function ke scope mein exist karta hai.
4. **Block Scope with let and const:**
   - let aur const ke saath block scope create hota hai, jo lexical scoping ke principles ko follow karta hai. Variables jo block ke andar defined hote hain, woh block ke bahar accessible nahi hote.

Example:

```
if (true) {
```

```
  let blockVariable = 'I am in block!';

  console.log(blockVariable); // 'I am in block!'

}



console.log(blockVariable); // ReferenceError: blockVariable is not defined
```

5. Yahaan, blockVariable sirf if block ke andar accessible hai, kyunki let block scope create karta hai.
6. **Closures and Lexical Scope:**
    - Lexical scope aur closures closely related hain. Closures allow functions to maintain access to variables from their enclosing (outer) scope even after the outer function has finished executing.

Example:
javascript
Copy code

```
function createCounter() {

  let count = 0;

  return function() {

    count += 1;

    return count;

  };

}



const counter = createCounter();

console.log(counter()); // 1

console.log(counter()); // 2
```

7. Yahaan, counter function createCounter ke count variable ko access karta hai, kyunki lexical scoping allow karta hai inner function ko outer function ke variables access karne ki.

Why It Matters:

1. **Predictability:**
    - Lexical scope ensures that variables are accessible only within their defined scopes, making code more predictable and easier to understand.
2. **Encapsulation:**
    - It helps in creating encapsulated code blocks where variables are hidden from the global scope and can only be accessed within their intended scope.
3. **Function Closures:**

- Lexical scope is fundamental for understanding closures, which are used for creating private variables, function factories, and other advanced programming patterns.

Summary:

Lexical scope in JavaScript refers to the visibility of variables based on their position in the code. Variables are accessible within their defined scope and any nested scopes, but not outside of them. This concept is crucial for understanding how functions access variables from their outer scopes and is fundamental to concepts like closures and block scope with let and const.

Variable shadowing.

Variable shadowing JavaScript mein tab hota hai jab ek variable jo kisi outer scope (jaise function ya block) mein declare hota hai, usi naam ka ek variable inner scope mein declare kiya jata hai. Jab aisa hota hai, to inner variable outer variable ko "shadow" kar deta hai, matlab inner scope mein inner variable ka istemal hota hai, aur outer variable accessible nahi hota.

Key Points:

1. **Inner Scope Ka Precedence:**
   - Jab aap inner scope mein variable ka naam refer karte hain, to JavaScript inner variable ko use karta hai, outer variable ko nahi.
2. **Scope Ka Mahatvapurn Hona:**
   - Shadowing usually functions ya blocks (jaise let ya const ke saath defined) mein hota hai.

Variable Shadowing Ka Example:

```
let x = 10; // Outer variable

function exampleFunction() {
  let x = 5; // Inner variable (shadows the outer variable)
  console.log(x); // Outputs: 5 (inner variable)
}

exampleFunction();
console.log(x); // Outputs: 10 (outer variable)
```

Explanation:

- Yahaan outer variable x ka value 10 hai.
- exampleFunction ke andar, ek naya variable x declare kiya gaya hai jiska value 5 hai.

- Jab function ke andar console.log(x) call hota hai, to yeh inner variable ko refer karta hai, isliye output 5 hota hai.
- Function ke bahar, console.log(x) outer variable ko refer karta hai, isliye output 10 hota hai.

Block Scope Ke Saath Shadowing:

Variable shadowing block statements (jaise if ya for blocks) ke andar bhi ho sakta hai:

```
let y = 20; // Outer variable


if (true) {
  let y = 15; // Inner variable (shadows the outer variable)
  console.log(y); // Outputs: 15 (inner variable)
}


console.log(y); // Outputs: 20 (outer variable)
```

Explanation:

- Yahaan, inner y outer y ko if block ke andar shadow karta hai.
- Inner y sirf us block ke andar accessible hai.

Kyun Yeh Important Hai:

1. **Code Clarity:**
    - Shadowing se confusion ho sakta hai agar aap isse dhyan se nahi sambhalte, especially larger codebases mein, kyunki yeh samajhna mushkil ho sakta hai ki kaunsa variable refer kiya ja raha hai.
2. **Best Practices:**
    - Inner scope mein outer scope ke variables ke saath same naam nahi rakhna behtar hai, taaki unintentional shadowing se bacha ja sake.
3. **Debugging:**
    - Shadowing ko samajhna debugging aur variable scopes ko samajhne mein madad karta hai, khas taur par jab complex functions ya nested blocks ka deal karte hain.

Summary:

Variable shadowing tab hota hai jab inner scope mein ek variable ka naam outer scope ke variable ke naam se milta hai. Inner variable ka precedence hota hai, aur outer variable us scope ke andar accessible nahi hota. Yeh concept variable scope ko samajhne aur code mein clarity rakhne ke liye important hai.

Hoisting

Hoisting is an important concept in JavaScript that explains the behavior of variable and function declarations. When JavaScript code executes, variable and function declarations are "hoisted" to the top of their scope. This means that variables and functions can be used before they are actually declared in the code.

Concept:

1. **Variable Hoisting:**
    - Variables declared with var are hoisted to the top of their scope (function or global), but their initialization stays in the same place. Initially, they are assigned a value of undefined.

Example:

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

var x = 5;

console.log(x); // 5
```

2. Here, var x is hoisted to the top, but x is initially undefined, so the first console.log(x) outputs undefined.
3. **Function Hoisting:**
    - Function declarations are fully hoisted, meaning the entire function definition is available throughout the scope.

Example:

```
console.log(greet()); // "Hello"

function greet() {

  return "Hello";

}
```

4. Here, the greet function is hoisted along with its definition, so you can call greet before its definition in the code.
5. **let and const Variables:**
    - Variables declared with let and const are also hoisted but are not initialized. They exist in a "temporal dead zone" from the start of the block until the declaration is encountered. Accessing them before their declaration results in a ReferenceError.

Example:

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 10;
```

6. Here, a is hoisted but cannot be accessed before its declaration, resulting in a ReferenceError.

Why It Matters:

1. **Predictability:**
    - Understanding hoisting helps avoid unexpected behavior in code when variables or functions are used before their declarations.
2. **Debugging:**
    - Knowing how hoisting works makes debugging easier by understanding why certain errors occur or why certain values are undefined.
3. **Best Practices:**
    - Using let and const instead of var helps avoid issues with hoisting and makes the code clearer. Declaring variables at the top of their scope is a best practice to prevent hoisting issues.

Summary:

Hoisting means that variable and function declarations are moved to the top of their scope during the compilation phase, allowing you to use them before their actual declaration in the code. However, variable initialization is not hoisted, which can lead to undefined values or errors if accessed too early. Understanding hoisting helps in writing more maintainable and predictable code.

Hindi Explanation.

Hoisting JavaScript mein ek important concept hai jo variable aur function declarations ke behavior ko explain karta hai. Jab JavaScript code execute hota hai, to variable aur function declarations ko unke scope ke top par "hoist" kar diya jata hai. Iska matlab hai ki variables aur functions ko declare karne se pehle bhi use kiya ja sakta hai.

Concept:

1. **Variable Hoisting:**
    - var se declared variables ko unke scope (function ya global) ke top par hoist kiya jata hai, lekin unka initialization wahi rehta hai jahan par wo defined hai. Initially, unko undefined value milti hai.

Example:

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

var x = 5;

console.log(x); // 5
```

2. Yahaan var x ko top par hoist kiya gaya hai, lekin x initially undefined hota hai, isliye pehla console.log(x) undefined print karta hai.
3. **Function Hoisting:**
    - Function declarations poori tarah se hoist ho jati hain. Matlab function ki definition pura scope ke throughout available hoti hai.

Example:
javascript
Copy code
```
console.log(greet()); // "Hello"
```

```
function greet() {

  return "Hello";

}
```

4. Yahaan greet function ko uski definition ke sath hoist kiya gaya hai, isliye aap greet ko uske definition se pehle bhi call kar sakte hain.
5. **let aur const Variables:**
    - let aur const se declared variables bhi hoist kiye jate hain, lekin unko initialize nahi kiya jata. Yeh variables ek "temporal dead zone" mein hote hain jab tak declaration nahi milti. Inko declaration se pehle access karne par ReferenceError milta hai.

Example:

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 10;
```

6. Yahaan a hoist hota hai lekin initialization ke pehle access nahi kiya ja sakta, isliye ReferenceError aata hai.

Kyun Important Hai:

1. **Predictability:**
    - Hoisting ko samajhna code ke unexpected behavior ko avoid karne mein help karta hai, jaise ki variables ya functions ko unke declaration se pehle use karna.
2. **Debugging:**
    - Hoisting ka knowledge debugging ko asaan banata hai, kyunki aap samajh sakte hain ki errors kyun aa rahe hain ya values undefined kyun ho rahi hain.
3. **Best Practices:**
    - var ke bajaye let aur const ka use karke hoisting ke issues se bach sakte hain aur code ko zyada clear aur predictable bana sakte hain. Variables ko unke scope ke top par declare karna best practice hai.

Summary:

Hoisting ka matlab hai ki JavaScript mein variables aur functions ke declarations ko unke scope ke top par le jaya jata hai, jis se aap unhe unki actual declaration se pehle bhi use kar sakte hain. Lekin, variable initialization hoisting ke saath nahi aati, isliye early access par undefined values ya errors mil sakte hain. Hoisting ko samajh kar aap apne code ko zyada maintainable aur predictable bana sakte hain.

Temporal DeadZone

"Temporal Dead Zone" (TDZ) ek programming term hai jo JavaScript ke context mein use hota hai, khas kar ES6 (ECMAScript 2015) ke "let" aur "const" keywords ke sath. TDZ tab hota hai jab ek variable ko declare kiya gaya hai, lekin uski value ko initialize nahi kiya gaya hai.

Iska matlab hai ki jab tak aap variable ko declare nahi kar lete, aap us variable ko access nahi kar sakte. Agar aap access karne ki koshish karte hain, to aapko "ReferenceError" milta hai.

Example:

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization
let a = 5;
```

Yahaan, a ko declare kiya gaya hai let keyword ke sath, lekin jab aap console.log(a) ko call karte hain, tab a TDZ mein hota hai, isliye aapko error milta hai. TDZ tab tak continue hoti hai jab tak variable ko declare aur initialize nahi kar diya jata.

Yeh behavior var keyword ke sath nahi hota. var ke sath, variables hoisting ke through accessible hote hain, lekin unki values undefined hoti hain agar initialization se pehle access kiya jaye.

TDZ ka primary purpose variable declaration aur initialization ko ensure karna hai, taaki code ki readability aur debugging process behtar ho.

English Explanation.

The "Temporal Dead Zone" (TDZ) is a programming term used in the context of JavaScript, particularly with the let and const keywords introduced in ES6 (ECMAScript 2015). TDZ occurs when a variable has been declared but not yet initialized.

This means that until you declare and initialize the variable, you cannot access it. If you try to access it before the declaration, you'll get a ReferenceError.

Example:

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization

let a = 5;
```

Here, a is declared with the let keyword, but when you try to console.log(a), a is in the TDZ, and hence you get an error. The TDZ persists until the variable is declared and initialized.

This behavior does not apply to the var keyword. With var, variables are hoisted and accessible, but they are undefined if accessed before initialization.

The primary purpose of TDZ is to ensure that variables are declared and initialized properly before use, which improves code readability and debugging.

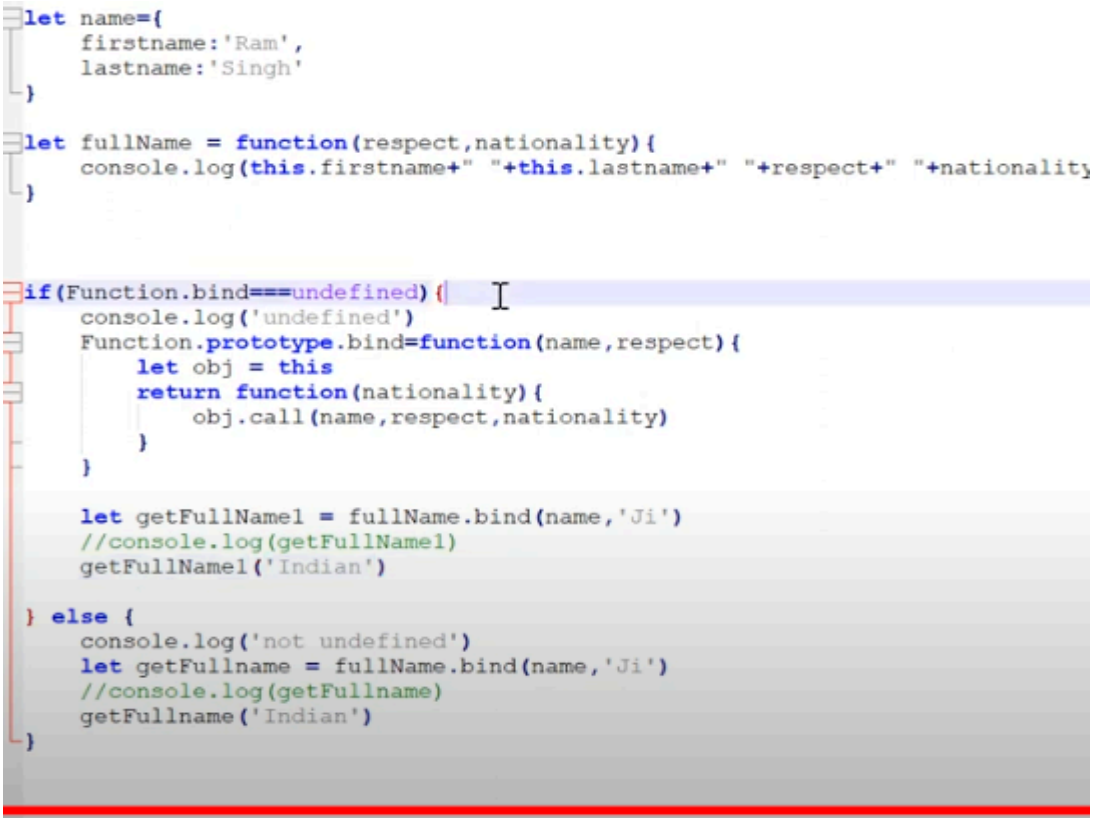
Polyfill

A **polyfill** in JavaScript is a piece of code (usually JavaScript code) that provides functionality that is not natively supported in some web browsers. It essentially "fills in" the gaps, allowing developers to use newer features or APIs that may not be available in older browser versions.

Agar koi function browser support nahi karta h . To ham ese implement karne ke liye polyfill ka use karte h

Key Points About Polyfills:

1. **Backward Compatibility**: Polyfills enable developers to use modern web standards and features without worrying about browser compatibility. For example, if you want to use Array.prototype.includes(), which is not available in Internet Explorer, you can use a polyfill that defines this method if it is not already present.
2. **Implementation of New Features**: Polyfills often mimic new JavaScript APIs or methods using older JavaScript syntax and constructs that are more widely supported. This allows developers to write code using the latest standards while maintaining compatibility with older browsers.
3. **Examples of Polyfills**:
    - Promise: For browsers that do not support Promises, a polyfill like the es6-promise library can be used.
    - fetch: Polyfills like whatwg-fetch provide the fetch API in environments that do not support it.
    - Object.assign: Provides a polyfill for environments that do not support Object.assign natively.
4. **Usage**: Polyfills are often included conditionally in web applications. Tools like Babel can also automatically add polyfills for the specific environments targeted by your codebase, ensuring that only the necessary polyfills are included.



```
let name={
    firstname:'Ram',
    lastname:'Singh'
}

let fullName = function(respect,nationality){
    console.log(this.firstname+" "+this.lastname+" "+respect+" "+nationality
}


if(Function.bind===undefined){
    console.log('undefined')
    Function.prototype.bind=function(name,respect){
        let obj = this
        return function(nationality){
            obj.call(name,respect,nationality)
        }
    }

    let getFullName1 = fullName.bind(name,'Ji')
    //console.log(getFullName1)
    getFullName1('Indian')

} else {
    console.log('not undefined')
    let getFullname = fullName.bind(name,'Ji')
    //console.log(getFullname)
    getFullname('Indian')
}
```

Hindi Explanation

Polyfill JavaScript mein ek code snippet hota hai jo kisi feature ki functionality ko provide karta hai jo purane browsers ya environments mein nahi hoti. Yani, agar aapko modern JavaScript features (jaise ES6+ ki features) use karna hai lekin aapke target users ke browsers unhe support nahi karte, to aap polyfills ka istemal karte hain.

Key Points:

1. **Compatibility:**
    - Polyfills ensure karte hain ki aapka code un browsers mein bhi kaam kare jo latest features ko support nahi karte. Yeh aapke application ko zyada accessible banata hai.
2. **Functionality Add Karna:**
    - Polyfills specific methods ya objects ko define karte hain agar wo pehle se exist nahi karte. Yeh kuch functions ya classes ko recreate karne ka kaam karte hain.
3. **Common Examples:**
    - Array methods jaise Array.prototype.includes, Promise, ya fetch function ko polyfill ki zaroorat ho sakti hai agar users purane browsers use kar rahe hain.

Example:

Yeh ek simple polyfill ka example hai jo Array.prototype.includes method ko define karta hai agar wo browser mein available nahi hai:

```
if (!Array.prototype.includes) {
  Array.prototype.includes = function(searchElement, fromIndex) {
    if (this == null) {
      throw new TypeError('"this" is null or not defined');
    }

    const O = Object(this);
    const len = O.length >>> 0;

    if (len === 0) {
      return false;
    }

    const n = fromIndex | 0;
    let k = Math.max(n >= 0 ? n : len - Math.abs(n), 0);

    while (k < len) {
      if (O[k] === searchElement) {
        return true;
      }
      k++;
    }
    return false;
  };
}
```

Explanation:

- Is polyfill mein pehle check kiya gaya hai ki Array.prototype.includes method exist karta hai ya nahi.
- Agar nahi karta, to yeh method ko define kar diya gaya hai jisse purane browsers mein bhi yeh method kaam karega.

Kyun Important Hai:

1. **Backward Compatibility:**
   - Polyfills developers ko allow karte hain ki wo modern features ka istemal karein bina unke users ko break kiye.
2. **Code Reusability:**
   - Aap ek polyfill ko apne projects mein reusable bana sakte hain, taaki aapko alag-alag browsers ke liye alag code nahi likhna pade.
3. **Progressive Enhancement:**
   - Polyfills aapke applications ko progressive enhancement ka approach follow karne mein madad karte hain, jahan pehle basic functionality implement hoti hai aur phir modern features add kiye jaate hain.

Summary:

Polyfills JavaScript mein ek tarika hai kisi modern feature ki functionality ko purane browsers mein implement karne ka. Yeh ensure karte hain ki aapka code wider audience ke liye accessible rahe, chahe unke browsers kitne bhi outdated kyun na ho.

Debounce and Throttle

Debounce aur **Throttling** JavaScript mein performance optimization techniques hain jo event handling ko manage karne ke liye use ki jaati hain, khaas taur par jab aapko frequent events, jaise scroll, resize, ya keypress, handle karna ho.

Debounce

Debounce ka matlab hai ki aap kisi function ko ek certain time delay ke saath invoke karte hain. Agar event baar-baar trigger hota hai, to function ko tab tak nahi chalega jab tak ki specified time period complete nahi ho jata.

Example:

```
function debounce(func, delay) {

  let timeout;

  return function(...args) {

    clearTimeout(timeout);

    timeout = setTimeout(() => {

      func.apply(this, args);

    }, delay);
```

```
  };
}

// Usage
const handleResize = debounce(() => {
  console.log('Window resized!');
}, 500);

window.addEventListener('resize', handleResize);
```

Explanation:

- Is example mein, handleResize function ko resize event par call kiya ja raha hai.
- Agar window resize hota hai, to event trigger hota hai, lekin function ko sirf 500 milliseconds ke baad call kiya jayega, jab aur koi resize event nahi hota.

Throttling

Throttling ka matlab hai ki aap kisi function ko ek fixed interval par execute karte hain, chahe event kitni bhi baar trigger ho. Yeh function ko limit karta hai ki wo kitni baar execute ho sakta hai ek specified time duration mein.

Example:

```
function throttle(func, limit) {
  let lastFunc;
  let lastRan;

  return function(...args) {
    const context = this;
    if (!lastRan) {
      func.apply(context, args);
      lastRan = Date.now();
    } else {
      clearTimeout(lastFunc);
```

```
      lastFunc = setTimeout(() => {
        if ((Date.now() - lastRan) >= limit) {
          func.apply(context, args);
          lastRan = Date.now();
        }
      }, limit - (Date.now() - lastRan));
    }
  };
}

// Usage
const handleScroll = throttle(() => {
  console.log('Scrolling...');
}, 1000);

window.addEventListener('scroll', handleScroll);
```

Explanation:

- Is example mein, handleScroll function ko scroll event par call kiya ja raha hai.
- Function ko sirf 1000 milliseconds (1 second) ke interval par hi execute kiya jayega, chahe scroll event kitni bhi baar trigger ho.

Kyun Important Hai:

1. **Performance Optimization:**
   - Dono techniques aapke applications ki performance ko improve karte hain, especially jab aapko frequent events handle karne hote hain. Yeh unnecessary function calls ko reduce karte hain.
2. **Resource Management:**
   - Debouncing aur throttling se aap server requests aur CPU resources ko effectively manage kar sakte hain, isse application ki responsiveness improve hoti hai.
3. **User Experience:**
   - Users ko smoother experience milta hai jab aap event handling ko efficiently manage karte hain, jaise scroll ya input events.

Summary:

- **Debounce**: Function ko ek specified delay ke baad call karta hai, aur agar event baar-baar trigger hota hai to function ko nahi chalega jab tak delay period complete nahi hota.
- **Throttling**: Function ko ek fixed interval par execute karta hai, chahe event kitni baar trigger ho. Yeh function calls ko limit karta hai taaki performance improve ho.

Dono techniques aapko event handling mein better control aur performance dete hain.

Shallow Copy and Deep Copy.

Shallow Copy and **Deep Copy** are two different ways of copying objects or arrays in JavaScript. Understanding these concepts is important because they can create differences in data manipulation and state management.

Shallow Copy

A **shallow copy** means that you create a new object or array that copies only the first-level properties or elements. If the original object or array contains nested objects or arrays, it copies those references, not the actual data.

Example:

```
const original = {
  name: 'John',
  age: 30,
  nested: {
    hobby: 'reading'
  }
};

// Shallow copy using Object.assign
const shallowCopy = Object.assign({}, original);

// Modifying the nested object
shallowCopy.nested.hobby = 'gaming';
```

```
console.log(original.nested.hobby); // Outputs: 'gaming'
```

Explanation:

- Here, shallowCopy contains copies of the properties of original.
- However, the nested object is a reference, so when we modify the nested object in shallowCopy, it also affects original.

Deep Copy

A **deep copy** means that you create a new object or array that contains a complete copy of the original object or array, including all nested objects or arrays. This means that the new object is completely independent of the original.

Example:

```
const original = {
  name: 'John',
  age: 30,
  nested: {
    hobby: 'reading'
  }
};

// Deep copy using JSON methods
const deepCopy = JSON.parse(JSON.stringify(original));

// Modifying the nested object
deepCopy.nested.hobby = 'gaming';

console.log(original.nested.hobby); // Outputs: 'reading'
```

Explanation:

- Here, deepCopy is a complete independent copy of original, including nested objects.
- Therefore, when we modify the nested object in deepCopy, original remains unaffected.

Why It Matters:

1. **Data Integrity:**
    - Deep copy helps maintain data integrity when working with complex objects or arrays.
2. **Avoiding Side Effects:**
    - Shallow copy can create unexpected side effects if you modify nested objects. Deep copy avoids this issue.
3. **Performance Considerations:**
    - Deep copy operations (especially using methods like JSON.stringify and JSON.parse) may be slower and may not support all data types (e.g., functions, undefined, or symbols). Shallow copy operations are typically faster but can lead to issues with nested data.

Summary:

- **Shallow Copy**: Creates a new object or array that copies only first-level properties or elements. Nested objects are copied by reference.
- **Deep Copy**: Creates a new object or array that contains a complete copy of the original object or array, including nested objects.

These concepts provide clarity in data handling and help avoid unexpected behavior.

Hindi Explanation

Shallow copy aur **deep copy** JavaScript mein objects ya arrays ki copying ke do alag tarike hain. Yeh samajhna important hai kyunki yeh data manipulation aur state management mein differences create kar sakte hain.

Shallow Copy

Shallow copy ka matlab hai ki aap ek naya object ya array create karte hain, lekin yeh sirf first-level properties ya elements ki copy karta hai. Agar original object ya array ke andar nested objects ya arrays hain, to yeh nested references ko copy karta hai, na ki unka actual data.

Example:

```
const original = {
  name: 'John',
  age: 30,
  nested: {
    hobby: 'reading'
  }
};
```

```
// Shallow copy using Object.assign

const shallowCopy = Object.assign({}, original);


// Modifying the nested object

shallowCopy.nested.hobby = 'gaming';


console.log(original.nested.hobby); // Outputs: 'gaming'
```

Explanation:

- Yahaan, shallowCopy mein original ki properties ki copy hoti hai.
- Lekin, nested object ka reference copy hota hai, isliye jab hum shallowCopy ke nested object ko modify karte hain, to original bhi affect hota hai.

Deep Copy

Deep copy ka matlab hai ki aap ek naya object ya array create karte hain, jisme original object ya array ke saath saath uske nested objects ya arrays ki bhi complete copy hoti hai. Iska matlab hai ki aapka naya object original se completely independent hota hai.

Example:

```
const original = {

  name: 'John',

  age: 30,

  nested: {

    hobby: 'reading'

  }

};


// Deep copy using JSON methods

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


// Modifying the nested object

deepCopy.nested.hobby = 'gaming';
```

```
console.log(original.nested.hobby); // Outputs: 'reading'
```

Explanation:

- Yahaan, deepCopy mein original ka complete independent copy hota hai, including nested objects.
- Isliye jab hum deepCopy ke nested object ko modify karte hain, to original unaffected rehta hai.

Kyun Important Hai:

1. **Data Integrity:**
   - Deep copy aapko data integrity maintain karne mein madad karta hai, jab aap complex objects ya arrays ke saath kaam kar rahe hote hain.
2. **Avoiding Side Effects:**
   - Shallow copy mein agar aap nested objects ko modify karte hain to yeh unexpected side effects create kar sakta hai. Deep copy is problem ko avoid karta hai.
3. **Performance Considerations:**
   - Deep copy operations (especially using methods like JSON.stringify and JSON.parse) may be slower and may not support all data types (e.g., functions, undefined, or symbols). Shallow copy operations are typically faster but can lead to issues with nested data.

Summary:

- **Shallow Copy**: Naya object ya array create karta hai jo sirf first-level properties ya elements ki copy karta hai. Nested objects ke references copy hote hain.
- **Deep Copy**: Naya object ya array create karta hai jo original object ya array ke saath saath unke nested objects ki bhi complete copy karta hai.

Yeh concepts aapko data handling mein clarity dete hain aur unexpected behavior se bacha sakte hain.

In JavaScript, map and forEach are both array methods used to iterate over elements, but they serve different purposes and have different behaviors. Here's a breakdown of their differences:

1. Purpose

- **map**:
  - Used to create a new array by applying a function to each element of the original array.
  - It transforms each element and returns the new array.
- **forEach**:

    - Used to execute a provided function once for each array element.
    - It performs a side effect but does not return a new array.

2. Return Value

- **map**:
    - Returns a new array containing the results of applying the callback function to each element.
- **forEach**:
    - Returns undefined. It's primarily used for its side effects, not for creating a new array.

3. Chaining

- **map**:
    - Can be chained with other array methods since it returns a new array.
- **forEach**:
    - Cannot be chained for further array processing since it returns undefined.

4. Mutability

- **map**:
    - Does not mutate the original array. It creates a new array based on the transformation.
- **forEach**:
    - Can mutate the original array if the callback function modifies the elements.

5. Use Cases

- **map**:
    - Use it when you need to transform data and create a new array.
- **forEach**:
    - Use it when you want to execute a function for each element without needing a new array.

Examples

Using map:

javascript

Copy code

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

const doubled = numbers.map(num => num * 2);



console.log(doubled); // Outputs: [2, 4, 6, 8]

console.log(numbers); // Outputs: [1, 2, 3, 4] (original array is unchanged)
```

Using forEach:

javascript

Copy code

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

numbers.forEach(num => {
  console.log(num * 2);
});


// Outputs:
// 2
// 4
// 6
// 8


console.log(numbers); // Outputs: [1, 2, 3, 4] (original array is unchanged)
```

Summary

- **map**: Creates a new array by transforming each element of the original array. It returns a new array.
- **forEach**: Executes a function on each element of the array without returning a new array. It's used for side effects.

Hindi Explanation

JavaScript mein map aur forEach dono array methods hain jo elements par iterate karne ke liye use hoti hain, lekin inka purpose aur behavior alag hota hai. Yahaan par in dono ke beech ke differences diye gaye hain:

1. Purpose

- **map**:
    - Yeh ek nayi array create karne ke liye use hota hai, jisme original array ke har element par ek function apply kiya jata hai.
    - Yeh har element ko transform karta hai aur nayi array return karta hai.
- **forEach**:

    - Yeh provided function ko har array element ke liye ek baar execute karne ke liye use hota hai.
    - Yeh sirf side effect perform karta hai aur nayi array return nahi karta.

2. Return Value

- **map**:
    - Yeh ek nayi array return karta hai, jo callback function ke results ko contain karti hai.
- **forEach**:
    - Yeh undefined return karta hai. Iska main purpose side effects hai, nayi array create karna nahi.

3. Chaining

- **map**:
    - Iska result nayi array hota hai, isliye aap ise dusre array methods ke saath chain kar sakte hain.
- **forEach**:
    - Yeh further array processing ke liye chain nahi kiya ja sakta kyunki yeh undefined return karta hai.

4. Mutability

- **map**:
    - Yeh original array ko mutate nahi karta. Yeh ek nayi array create karta hai transformation ke basis par.
- **forEach**:
    - Agar callback function elements ko modify kare, to yeh original array ko mutate kar sakta hai.

5. Use Cases

- **map**:
    - Jab aapko data transform karna ho aur nayi array create karni ho, tab isse use karein.
- **forEach**:
    - Jab aapko har element par function execute karna ho bina nayi array ke, tab isse use karein.

Examples

map ka istemal:

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

const doubled = numbers.map(num => num * 2);


console.log(doubled); // Outputs: [2, 4, 6, 8]
```

```
console.log(numbers); // Outputs: [1, 2, 3, 4] (original array unchanged)
```

forEach ka istemal:

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

numbers.forEach(num => {
  console.log(num * 2);
});

// Outputs:
// 2
// 4
// 6
// 8

console.log(numbers); // Outputs: [1, 2, 3, 4] (original array unchanged)
```

Summary

- **map**: Original array ke har element ko transform karke ek nayi array create karta hai. Yeh ek nayi array return karta hai.
- **forEach**: Har element par function execute karta hai bina nayi array ke. Yeh sirf side effects ke liye use hota hai.

Feature	`forEach()`	`map()`

Return Value	`undefined` (side-effect based)	**New Array** (transformed values)
Purpose	Iterate over array, apply side effects	Transform array values to a new array
Modification	Original array ko modify kar sakte hain	Original array change nahi hota
Break	`Break` nahi kar sakte loop ko	`Break` nahi kar sakte loop ko
Async Handling	Asynchronous operations ke liye complicated	Suitable for asynchronous operations, but synchronous execution

Reduce Method.

Reduce method JavaScript mein ek powerful array method hai jo ek single value ko generate karne ke liye array ke elements par iterate karta hai. Yeh method array ko reduce karke ek single output value (jaise sum, product, ya kisi aur type ka aggregate) banata hai.

Syntax

```
array.reduce(callback, initialValue);
```

- **callback**: Yeh ek function hai jo har element ke liye execute hota hai. Is function ko 4 parameters milte hain:

    - **accumulator**: Yeh woh value hai jo function return karta hai, jo agle iteration mein use hota hai.
    - **currentValue**: Yeh current element hai jo array ke iteration ke waqt process hota hai.
    - **currentIndex** (optional): Yeh current element ka index hai.
    - **array** (optional): Yeh original array hai.
- **initialValue** (optional): Yeh wo value hai jisse accumulator shuru hota hai. Agar yeh nahi diya gaya, to pehla element accumulator ke liye use hota hai, aur iteration second element se shuru hoti hai.

Example

Summing an Array of Numbers

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



// Using reduce to sum the numbers

const sum = numbers.reduce((accumulator, currentValue) => {

  return accumulator + currentValue;

}, 0); // Initial value is 0



console.log(sum); // Outputs: 15
```

Explanation:

- Is example mein, reduce method numbers array ke har element ko iterate karta hai.
- accumulator pehle iteration ke liye 0 hota hai (initial value).
- Har iteration mein currentValue ko accumulator ke saath add kiya jata hai, aur nayi value accumulator ban jaati hai.
- Final output 15 hota hai.

Finding the Maximum Value

```
const numbers = [5, 3, 8, 1, 4];



// Using reduce to find the maximum value

const max = numbers.reduce((accumulator, currentValue) => {

  return Math.max(accumulator, currentValue);

});
```

```
console.log(max); // Outputs: 8
```

Explanation:

- Yahaan, reduce maximum value nikalta hai.
- Har iteration mein Math.max function use hota hai, jo accumulator aur currentValue mein se bada value return karta hai.

Why Use Reduce?

1. **Simplifies Code:**
   - Multiple operations ko ek single method mein combine kar sakte hain, jisse code cleaner aur more readable hota hai.
2. **Powerful Aggregation:**
   - Aap sum, average, product, maximum, minimum, ya kisi bhi custom aggregation operation ko easily perform kar sakte hain.
3. **Functional Programming:**
   - Yeh functional programming paradigm ko follow karta hai, jo state ko mutate nahi karta, balki nayi values produce karta hai.

Summary

Reduce method ek array ke elements par iterate karke ek single value generate karta hai. Yeh powerful hai aggregation operations ke liye, jisse aapko complex calculations ko simple aur clean tarike se implement karne ki flexibility milti hai.

Difference Arrow and normal Function.

JavaScript mein **arrow functions** aur **normal functions** ke beech kuch key differences hain. Yahaan par in dono ke differences ko detail mein samjhaya gaya hai:

1. Syntax

Normal Function:

```
function normalFunction(param1, param2) {
    return param1 + param2;
}
```

Arrow Function:

```
const arrowFunction = (param1, param2) => {
    return param1 + param2;
```

```
};
```

2. this Binding

- **Normal Function:**
  - Normal functions mein this ka context function ke call hone ke tareeke par depend karta hai. Agar function ko kisi object ke method ke roop mein call kiya jata hai, to this us object ko refer karta hai.
- **Arrow Function:**
  - Arrow functions mein this lexical hota hai, matlab yeh parent scope se this ko inherit karta hai. Isliye, arrow functions mein this ki value kabhi change nahi hoti.

3. arguments Object

- **Normal Function:**
  - Normal functions mein arguments object available hota hai, jo function ke paas diye gaye arguments ki list ko represent karta hai.
- **Arrow Function:**
  - Arrow functions mein arguments object nahi hota. Agar aapko arguments ki zaroorat hai, to aapko normal function ya rest parameters (...args) ka istemal karna padega.

4. Constructor Behavior

- **Normal Function:**
  - Normal functions ko new keyword ke saath instantiate kiya ja sakta hai, yani yeh constructors ban sakte hain.
- **Arrow Function:**
  - Arrow functions ko new keyword ke saath istemal nahi kiya ja sakta, isliye yeh constructors nahi ban sakte.

5. Return Behavior

- **Normal Function:**
  - Normal functions mein aapko return statement use karna hota hai agar aap koi value return karna chahte hain.
- **Arrow Function:**
  - Agar arrow function ek hi expression hai, to aap return keyword ke bina bhi value return kar sakte hain.

```
const add = (a, b) => a + b; // Implicit return
```

-

Examples

Normal Function Example:

```
function add(a, b) {
```

```
    return a + b;
}

console.log(add(2, 3)); // Outputs: 5
```

Arrow Function Example:

```
const add = (a, b) => a + b; // Implicit return

console.log(add(2, 3)); // Outputs: 5
```

Summary

- **Syntax**: Arrow functions ka syntax concise hota hai.
- **this Binding**: Normal functions mein this call ke context par depend karta hai, jabki arrow functions mein this parent scope se inherit hota hai.
- **arguments Object**: Normal functions mein arguments object hota hai, jabki arrow functions mein nahi.
- **Constructor Behavior**: Normal functions ko constructors banaya ja sakta hai, jabki arrow functions nahi.
- **Return Behavior**: Arrow functions mein single expression ka implicit return hota hai.

In differences ko samajhna aapko JavaScript functions ko effectively use karne mein madad karega.

With Example

1. Syntax

Normal Function:

```
function normalFunction(param1, param2) {
    return param1 + param2;
}

console.log(normalFunction(2, 3)); // Outputs: 5
```

Arrow Function:

```
const arrowFunction = (param1, param2) => {
    return param1 + param2;
};

console.log(arrowFunction(2, 3)); // Outputs: 5
```

2. this Binding

Normal Function:

```
const obj = {
    value: 10,
    normalFunction: function() {
        console.log(this.value); // `this` refers to `obj`
    }
};

obj.normalFunction(); // Outputs: 10
```

Arrow Function:

```
const obj = {
    value: 10,
    arrowFunction: () => {
        console.log(this.value); // `this` refers to the outer scope (global in this case)
    }
};
```

```
obj.arrowFunction(); // Outputs: undefined (in non-strict mode)
```

3. arguments Object

Normal Function:

```
function normalFunction() {
    console.log(arguments); // Outputs: [1, 2, 3]
}

normalFunction(1, 2, 3);
```

Arrow Function:

```
const arrowFunction = () => {
    console.log(arguments); // Throws an error: `arguments is not defined`
};

arrowFunction(1, 2, 3);
```

4. Constructor Behavior

Normal Function:

```
function Person(name) {
    this.name = name;
}
```

```
const john = new Person('John');

console.log(john.name); // Outputs: John
```

Arrow Function:

```
const Person = (name) => {

    this.name = name; // Throws an error: `this` is not a constructor

};


// const john = new Person('John'); // This will throw an error
```

5. Return Behavior

Normal Function:

```
function add(a, b) {

    return a + b;

}


console.log(add(2, 3)); // Outputs: 5
```

Arrow Function (Implicit Return):

```
const add = (a, b) => a + b; // Implicit return


console.log(add(2, 3)); // Outputs: 5
```

Summary of Differences

- **Syntax**: Arrow functions have a more concise syntax.

- **this Binding**: Normal functions bind this to the calling context, while arrow functions lexically bind this to the enclosing scope.
- **arguments Object**: Normal functions have access to the arguments object, while arrow functions do not.
- **Constructor Behavior**: Normal functions can be used as constructors, while arrow functions cannot.
- **Return Behavior**: Arrow functions allow for implicit returns for single expressions.

These examples should help clarify the differences between normal functions and arrow functions in .

Higher Order Function

What is a Higher Order Function?

A **Higher Order Function** in JavaScript is a function that can take one or more functions as arguments or return a function as its result. These functions are an important concept in functional programming and help you make your code modular and reusable.

Characteristics of Higher Order Functions

1. **Functions as Arguments**: Higher order functions can take other functions as arguments.
2. **Functions as Return Values**: These functions can also return other functions as their result.
3. **Encapsulation of Behavior**: They encapsulate behavior, allowing you to manage complex functionality through simpler functions.

Examples

1. Functions as Arguments

A function that takes another function as an argument:

javascript

Copy code

```
function greet(name) {
    return `Hello, ${name}!`;
}

function greetUser(greetingFunction, userName) {
    return greetingFunction(userName);
}

console.log(greetUser(greet, 'Alice')); // Outputs: Hello, Alice!
```

Explanation: Here, the greetUser function takes the greet function as an argument.

2. Functions as Return Values

A function that returns another function:

javascript

Copy code

```
function multiplier(factor) {
    return function(x) {
        return x * factor;
    };
}


const double = multiplier(2);
console.log(double(5)); // Outputs: 10
```

Explanation: Here, the multiplier function returns a new function that multiplies its input by factor.

Common Higher Order Functions

1. **map**: Applies a function to each element of an array and returns a new array.
2. **filter**: Filters elements of an array based on a condition and returns only those elements that meet the condition.
3. **reduce**: Aggregates the elements of an array into a single value.

Example with Array Methods

Using map

javascript

Copy code

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Outputs: [2, 4, 6, 8]
```

Using filter

javascript

Copy code

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

const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers); // Outputs: [2, 4]
```

Using reduce

javascript

Copy code

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

const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

console.log(sum); // Outputs: 10
```

Summary

- **Higher Order Functions** are functions that take one or more functions as arguments or return functions.
- They help make your code modular and reusable.
- Common examples include map, filter, and reduce, which are used for array manipulations.

Understanding this concept will help you use functions more effectively in JavaScript!

Hindi Explanation.

Higher Order Function JavaScript mein aise functions hote hain jo ek ya zyada functions ko argument ke roop mein le sakte hain ya phir ek function ko return karte hain. Yeh functions functional programming ka ek important concept hain aur isse aapko code ko modular aur reusable banane mein madad milti hai.

Characteristics of Higher Order Functions

1. **Functions as Arguments**: Higher order functions dusre functions ko arguments ke roop mein le sakte hain.
2. **Functions as Return Values**: Yeh functions apne return value ke roop mein bhi functions de sakte hain.
3. **Encapsulation of Behavior**: Yeh functions behavior ko encapsulate karte hain, jisse aap complex functionality ko simple functions ke through manage kar sakte hain.

Examples

1. Functions as Arguments

Ek function jo dusre function ko argument ke roop mein leta hai:

javascript

Copy code

```
function greet(name) {
    return `Hello, ${name}!`;
}

function greetUser(greetingFunction, userName) {
    return greetingFunction(userName);
}

console.log(greetUser(greet, 'Alice')); // Outputs: Hello, Alice!
```

Explanation: Yahaan greetUser function greet function ko argument ke roop mein le raha hai.

2. Functions as Return Values

Ek function jo dusre function ko return karta hai:

```
function multiplier(factor) {
    return function(x) {
        return x * factor;
    };
}

const double = multiplier(2);
console.log(double(5)); // Outputs: 10
```

Explanation: Yahaan multiplier function ek naya function return karta hai jo input value ko factor ke saath multiply karta hai.

Common Higher Order Functions

1. **map**: Ek array ke har element par function apply karta hai aur nayi array return karta hai.
2. **filter**: Ek array ke elements ko filter karne ke liye use hota hai, jisse sirf un elements ko return karta hai jo condition meet karte hain.
3. **reduce**: Ek array ke elements ko aggregate karne ke liye use hota hai, jisse ek single value return hoti hai.

Example with Array Methods

Using map

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

const doubled = numbers.map(num => num * 2);

console.log(doubled); // Outputs: [2, 4, 6, 8]
```

Using filter

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

const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers); // Outputs: [2, 4]
```

Using reduce

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

const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

console.log(sum); // Outputs: 10
```

Summary

- **Higher Order Functions** wo functions hote hain jo ek ya zyada functions ko arguments ke roop mein lete hain ya phir functions ko return karte hain.
- Yeh aapko code ko modular aur reusable banane mein madad karte hain.

- Common examples hain map, filter, aur reduce, jo array manipulations ke liye kaam aate hain.

Yeh concept aapko JavaScript mein functions ko behtar tarike se samajhne aur istemal karne mein madad karega!

What is a Callback Function?

A **callback function** in JavaScript is a function that is passed as an argument to another function. This second function executes the callback function at some point during its execution. Callback functions are particularly useful in asynchronous programming and event handling.

Key Features of Callback Functions

1. **Function as Argument**: Callback functions can be passed as arguments to other functions.
2. **Asynchronous Execution**: They are often used for asynchronous operations, where you don't need to wait for one operation to complete before moving on.
3. **Event Handling**: Callback functions are commonly used for handling events (like button clicks).

Example of a Callback Function

1. Simple Callback

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

function processUserInput(callback) {
    const name = 'Alice';
    callback(name);
}

// Calling processUserInput and passing greet as a callback
processUserInput(greet); // Outputs: Hello, Alice!
```

Explanation: Here, the greet function is passed as a callback to the processUserInput function. When processUserInput is called, it invokes the greet function.

2. Asynchronous Callback

```
function fetchData(callback) {
    setTimeout(() => {
        const data = 'Data loaded!';
        callback(data);
    }, 2000); // Simulating a delay of 2 seconds
}

fetchData((result) => {
    console.log(result); // Outputs: Data loaded! (after 2 seconds)
});
```

Explanation: Here, the fetchData function simulates an asynchronous operation (using setTimeout). Once the data is loaded, it calls the callback function.

Use Cases of Callback Functions

1. **Asynchronous Operations**: AJAX requests, file reading, timers, etc.
2. **Event Handling**: Button clicks, mouse movements, etc.
3. **Array Methods**: Functions like map, filter, and reduce take callback functions as arguments to operate on array elements.

Example with Array Method

Using forEach

```
const numbers = [1, 2, 3, 4];
numbers.forEach((num) => {
    console.log(num * 2); // Outputs: 2, 4, 6, 8
});
```

Summary

- A **callback function** is a function that is passed as an argument to another function, which then calls it during its execution.
- They are particularly useful in asynchronous programming and event handling.
- Common examples include handling events, asynchronous operations, and working with array methods.

Callback functions enhance flexibility and responsiveness in JavaScript, helping you create a better user experience!

Hindi Explanation

Callback Function JavaScript mein ek function hota hai jo kisi dusre function ko argument ke roop mein diya jata hai, aur yeh dusra function jab execute hota hai, tab callback function ko call kiya jata hai. Yeh asynchronous programming aur event handling mein kaafi useful hote hain.

Key Features of Callback Functions

1. **Function as Argument**: Callback functions ko aap dusre functions mein arguments ke roop mein pass kar sakte hain.
2. **Asynchronous Execution**: Callback functions ko asynchronous operations ke liye use kiya jata hai, jahan kisi operation ko complete hone ka intezaar nahi karna padta.
3. **Event Handling**: Callback functions ko events (jese button clicks) ke liye use kiya jata hai.

Example of a Callback Function

1. Simple Callback

javascript

Copy code

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

function processUserInput(callback) {
    const name = 'Alice';
    callback(name);
}

// Calling processUserInput and passing greet as a callback
```

```
processUserInput(greet); // Outputs: Hello, Alice!
```

Explanation: Yahaan greet function ko processUserInput function mein callback ke roop mein pass kiya gaya hai. Jab processUserInput call hota hai, tab yeh greet function ko call karta hai.

2. Asynchronous Callback

```
function fetchData(callback) {
    setTimeout(() => {
        const data = 'Data loaded!';
        callback(data);
    }, 2000); // Simulating a delay of 2 seconds
}

fetchData((result) => {
    console.log(result); // Outputs: Data loaded! (after 2 seconds)
});
```

Explanation: Yahaan fetchData function ek asynchronous operation (setTimeout) ko simulate kar raha hai. Jab data load hota hai, tab callback function ko call kiya jata hai.

Use Cases of Callback Functions

1. **Asynchronous Operations**: AJAX requests, file reading, timers, etc.
2. **Event Handling**: Button clicks, mouse movements, etc.
3. **Array Methods**: Functions like map, filter, and reduce take callback functions as arguments to operate on array elements.

Example with Array Method

Using forEach

```
const numbers = [1, 2, 3, 4];
numbers.forEach((num) => {
    console.log(num * 2); // Outputs: 2, 4, 6, 8
```

```
});
```

Summary

- **Callback Function** wo function hote hain jo kisi dusre function ko argument ke roop mein diya jata hai aur dusra function jab execute hota hai tab callback function ko call kiya jata hai.
- Yeh asynchronous programming aur event handling mein kaafi useful hote hain.
- Common examples include handling events, asynchronous operations, and working with array methods.

Callback functions JavaScript mein flexibility aur responsiveness ko badhate hain, jo aapko better user experience create karne mein madad karte hain!

Pure Function .

A **pure function** in programming, particularly in JavaScript, is a function that meets two main criteria:

1. **Deterministic**: Given the same input, a pure function always returns the same output. This means it does not rely on any external state or data that can change outside of the function.
2. **No Side Effects**: A pure function does not cause any side effects. This means it does not modify any external state, such as global variables, or perform any observable actions outside of returning a value (like logging to the console or modifying a DOM element).

Characteristics of Pure Functions

- **Predictable**: Since pure functions always produce the same output for the same input, they are predictable and easier to test.
- **Referential Transparency**: You can replace a call to a pure function with its result without changing the program's behavior.
- **Easier to Debug**: Because they do not depend on or modify external states, pure functions are easier to debug.

Example of a Pure Function

Here's an example of a pure function:

```
function add(a, b) {
    return a + b;
}

// Calling the pure function
console.log(add(2, 3)); // Outputs: 5
```

```
console.log(add(2, 3)); // Outputs: 5 (same input gives same output)
```

Example of an Impure Function

In contrast, here's an example of an impure function:

```
let counter = 0;

function incrementCounter() {
    counter += 1; // Modifies external state
    return counter;
}

// Calling the impure function
console.log(incrementCounter()); // Outputs: 1
console.log(incrementCounter()); // Outputs: 2 (different output with the same input)
```

Summary

- A **pure function** always returns the same output for the same input and has no side effects.
- They are predictable, easier to test, and help improve code maintainability.
- Understanding pure functions is essential in functional programming and for writing clean, reliable code in JavaScript!

Hindi Explanation.

Pure function programming mein, khaas taur par JavaScript mein, ek aisi function hoti hai jo do main criteria ko meet karti hai:

1. **Deterministic**: Same input dene par, pure function hamesha same output return karti hai. Iska matlab hai ki yeh kisi external state ya data par depend nahi karti jo function ke bahar change ho sakta hai.
2. **No Side Effects**: Pure function kisi external state ko modify nahi karti. Iska matlab hai ki yeh kisi global variable ko change nahi karti, ya kisi observable action ko nahi perform karti (jaise console mein log karna ya DOM element ko modify karna) jab tak sirf value return nahi hoti.

Characteristics of Pure Functions

- **Predictable**: Kyunki pure functions hamesha same output deti hain same input ke liye, yeh predictable hoti hain aur test karna aasan hota hai.
- **Referential Transparency**: Aap pure function ki call ko uski result se replace kar sakte hain bina program ki behavior ko badle.
- **Easier to Debug**: Yeh external states par depend nahi karti, isliye debugging karna aasan hota hai.

Pure Function Ka Example

Yahan ek pure function ka example diya gaya hai:

```
function add(a, b) {
    return a + b;
}

// Pure function ko call karna
console.log(add(2, 3)); // Outputs: 5
console.log(add(2, 3)); // Outputs: 5 (same input par same output)
```

Impure Function Ka Example

Iske contrast mein, yahan ek impure function ka example diya gaya hai:

```
let counter = 0;

function incrementCounter() {
    counter += 1; // External state ko modify karna
    return counter;
}

// Impure function ko call karna
console.log(incrementCounter()); // Outputs: 1
console.log(incrementCounter()); // Outputs: 2 (same input par different output)
```

Summary

- **Pure function** hamesha same output deti hai same input ke liye aur ismein koi side effects nahi hote.
- Yeh predictable hoti hain, test karne mein aasan hoti hain, aur code ki maintainability ko improve karti hain.
- Pure functions samajhna functional programming aur JavaScript mein clean, reliable code likhne ke liye zaroori hai!

Strict Mode

Strict Mode JavaScript ka ek feature hai jo code ko zyada secure aur efficient banane ke liye design kiya gaya hai. Jab aap strict mode ka istemal karte hain, to kuch aise behaviors ko disable kar diya jata hai jo code ko unexpected tarike se chalne ka mauka dete hain.

Strict Mode Ka Use Kaise Karte Hain

Strict mode ko activate karne ke liye, aapko apne JavaScript file ke shuruat mein ya function ke andar "use strict"; likhna hota hai:

```
"use strict";


function myFunction() {

    // Strict mode yahan apply hoga

}
```

Strict Mode Ke Benefits

1. **Error Detection**: Strict mode kuch errors ko detect karne mein madad karta hai jo normal mode mein chhupe reh sakte hain.
2. **Secure Code**: Isse aap apne code ko zyada secure bana sakte hain, kyunki kuch unsafe features disable ho jate hain.
3. **Eliminate Silent Errors**: Yeh kuch aise silent errors ko throw kar deta hai jo bina strict mode ke kuch unexpected behavior create kar sakte hain.

Strict Mode Ke Kuch Rules

Undeclared Variables: Aapko variables ko declare karna hoga. Undeclared variables use karne par error milega.

```
"use strict";

x = 10; // Error: x is not defined
```

1.

Read-Only Properties: Aap read-only properties ko modify nahi kar sakte.

```
"use strict";

Object.defineProperty(this, 'constant', {

    value: 42,

    writable: false

});

constant = 10; // Error: Cannot assign to read only property 'constant'
```

2.

Duplicate Parameter Names: Function mein duplicate parameter names nahi hone chahiye.
javascript
Copy code

```
"use strict";

function func(a, a, b) { // Error: Duplicate parameter name not allowed in this context

    return a + b;

}
```

3.

Using this in Global Context: Strict mode mein, agar aap function ko global context mein call karte hain, to this undefined hota hai, jabki normal mode mein this global object ko refer karta hai.

```
"use strict";

function myFunction() {

    console.log(this); // Outputs: undefined

}

myFunction();
```

4.

Summary

- **Strict Mode** JavaScript ka ek feature hai jo code ko zyada secure aur reliable banata hai.
- Isse errors ko detect karne mein madad milti hai aur unsafe features ko disable kiya jata hai.
- Strict mode ko activate karne ke liye, code ke shuruat mein "use strict"; likhna hota hai.

Strict mode ka istemal karna aapke code ki quality aur maintainability ko improve karne mein madad karta hai!

Event Loop Hindi Explanation.

Event Loop JavaScript ka ek core concept hai jo asynchronous programming ko manage karne ke liye use hota hai. Iska kaam hai JavaScript runtime environment mein tasks ko execute karna, jisme synchronous aur asynchronous operations dono shamil hote hain.

Event Loop Ka Kaam Kaise Karta Hai

JavaScript ek single-threaded language hai, iska matlab hai ki ek samay par sirf ek hi task execute hota hai. Lekin, JavaScript asynchronous operations jaise timers, AJAX requests, aur event listeners ko handle kar sakta hai. Yahan event loop ka role aata hai.

Components of the Event Loop

1. **Call Stack**: Yeh stack hai jahan function calls execute hote hain. Jab function call hota hai, to yeh stack mein push hota hai, aur jab function complete hota hai, to yeh stack se pop hota hai.
2. **Web APIs**: Yeh asynchronous tasks ko handle karte hain, jaise setTimeout, fetch, etc. Jab aap in functions ko call karte hain, to yeh tasks ko background mein handle karte hain.
3. **Callback Queue (Task Queue)**: Jab asynchronous tasks complete hote hain, to unke callbacks ko callback queue mein push kiya jata hai.
4. **Event Loop**: Yeh check karta hai ki kya call stack khali hai. Agar call stack khali hai, to yeh callback queue se next task ko call stack mein push karta hai.

Event Loop Ka Flow

1. JavaScript execution call stack se start hota hai.
2. Synchronous code execute hota hai.
3. Jab asynchronous function call hota hai, to yeh Web APIs ko call karta hai, aur control call stack se remove hota hai.
4. Jab asynchronous task complete hota hai, to callback ko callback queue mein add kiya jata hai.
5. Agar call stack khali hai, to event loop callback queue se next callback ko call stack mein push karta hai.

Example of Event Loop

```
console.log('Start');


setTimeout(() => {
    console.log('Timeout 1');
}, 0);
```

```
setTimeout(() => {
    console.log('Timeout 2');
}, 0);

console.log('End');
```

Output:

```
Start
End
Timeout 1
Timeout 2
```

Explanation:

- Pehle console.log('Start') aur console.log('End') synchronous tasks hain, jo turant execute hote hain.
- setTimeout calls asynchronous hain, jo Web APIs ko call karte hain. Unka callback queue mein add hone mein time lagta hai.
- Event loop tab in callbacks ko execute karta hai jab call stack khali ho jata hai.

Summary

- **Event Loop** JavaScript ka mechanism hai jo asynchronous tasks ko handle karta hai.
- Yeh call stack, Web APIs, aur callback queue ke beech coordination karta hai.
- Isse JavaScript ko single-threaded hone ke bawajood non-blocking behavior achieve karne ki capability milti hai.

Event loop ka samajhna JavaScript ki asynchronous programming ko samajhne ke liye zaroori hai!

Generator Function

Generator Function JavaScript mein ek special type ka function hota hai jo function* syntax se define kiya jata hai. Yeh function ek iterator return karta hai aur yeh control ko pause aur resume karne ki capability provide karta hai. Iska use aapko sequence of values ko generate karne ke liye karne mein madad karta hai, bina saare values ko ek sath compute kiye.

Generator Function Ki Features

1. **Yield Keyword**: Generator function ke andar yield keyword ka use hota hai, jo function ko pause kar deta hai aur ek value return karta hai. Jab aap function ko phir se call karte hain, to yeh wahi se continue hota hai.
2. **Iterator**: Generator function ek iterator object return karta hai, jise aap next() method ke through iterate kar sakte hain.
3. **Stateful**: Generator function apne internal state ko maintain karta hai. Jab aap yield statement par function ko pause karte hain, to state save ho jati hai.

Generator Function Ka Syntax

```
function* generatorFunction() {
    yield value1;
    yield value2;
    // ...
}
```

Example of a Generator Function

```
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

// Generator function ko call karna
const generator = numberGenerator();

console.log(generator.next()); // Outputs: { value: 1, done: false }
console.log(generator.next()); // Outputs: { value: 2, done: false }
console.log(generator.next()); // Outputs: { value: 3, done: false }
```

```
console.log(generator.next()); // Outputs: { value: undefined, done: true }
```

Explanation

- Jab aap numberGenerator function ko call karte hain, to yeh ek iterator object return karta hai.
- Aap next() method ka use karke generator function ko call karte hain. Har baar call karne par, function yield statement par pause hota hai aur current value return karta hai.
- Jab sab yield statements execute ho jate hain, to done property true ho jati hai.

Use Cases of Generator Functions

1. **Lazy Evaluation**: Generators values ko tab generate karte hain jab unki zarurat hoti hai, isliye yeh memory efficient hote hain.
2. **Infinite Sequences**: Generators ko infinite sequences generate karne ke liye use kiya ja sakta hai, jaise Fibonacci series.
3. **Asynchronous Programming**: Generators ka use asynchronous programming patterns ko simplify karne ke liye bhi hota hai, jahan aap control flow ko manage kar sakte hain.

Example of Infinite Generator

```
function* infiniteGenerator() {
    let i = 0;
    while (true) {
        yield i++;
    }
}

const gen = infiniteGenerator();

console.log(gen.next().value); // Outputs: 0
console.log(gen.next().value); // Outputs: 1
console.log(gen.next().value); // Outputs: 2
```

Summary

- **Generator Function** JavaScript mein ek powerful tool hai jo function* syntax aur yield keyword ke saath kaam karta hai.
- Yeh asynchronous programming, lazy evaluation, aur infinite sequences ke liye useful hai.

- Generators ko samajhna JavaScript ke advanced features aur patterns ko samajhne ke liye zaroori hai!

setTimeout setImmediate nextTick

In JavaScript, setTimeout, setImmediate, and nextTick are all methods used to schedule the execution of code. However, they have different use cases and timing characteristics. Let's explore each of them:

1. setTimeout

- **Definition**: setTimeout is used to schedule a function to be executed after a specified delay (in milliseconds).
- **Usage**: It can be used for delaying the execution of a function.

Example:

javascript

```
console.log('Start');


setTimeout(() => {
    console.log('Executed after 2 seconds');
}, 2000);


console.log('End');
```

Explanation: The setTimeout function delays the execution of its callback by 2 seconds. The other synchronous code runs immediately.

2. setImmediate

- **Definition**: setImmediate is used to execute a single callback after the current event loop cycle. It's specific to Node.js and is not available in browsers.
- **Usage**: It's often used when you want to execute a function after the current operation completes, but before any timers.

Example:

javascript

```
console.log('Start');
```

```
setImmediate(() => {

    console.log('Executed in the next iteration of the event loop');

});


console.log('End');
```

Explanation: The setImmediate callback runs after the current event loop cycle, which is after the synchronous code has finished executing.

3. process.nextTick

- **Definition**: process.nextTick is a Node.js-specific function that allows you to schedule a callback to be invoked in the same phase of the event loop, just after the currently executing operation.
- **Usage**: It's useful for ensuring that a function runs after the current operation but before any I/O tasks.

Example:

```
console.log('Start');


process.nextTick(() => {

    console.log('Executed in the next tick of the event loop');

});


console.log('End');
```

Explanation: The process.nextTick callback is executed immediately after the current operation completes, before any other I/O tasks or timers.

Summary of Differences

Method	Timing	Context
setTimeout	After a specified delay (ms)	Browsers and Node.js
setImmediate	After the current event loop cycle	Node.js only
process.nextTick	After the current operation	Node.js only

Conclusion

- **setTimeout** is for delayed execution.
- **setImmediate** is for executing code after the current event loop cycle.
- **process.nextTick** is for executing code right after the current operation, before I/O.

Understanding these timing functions is essential for managing asynchronous operations effectively in JavaScript!

Immediately Invoked Function Expression

Self-invoked function, yaani "Immediately Invoked Function Expression" (IIFE), ek aisa function hai jo apne aap chal jata hai jab wo define hota hai. Iska istemal aksar scope ko encapsulate karne ke liye kiya jata hai, taaki variables ko global scope mein expose na kiya jaye.

Syntax

IIFE ka syntax kuch is tarah hota hai:

```
(function() {
    // yahan aapka code
})();
```

Ya phir aap arrow function ka istemal bhi kar sakte hain:

```
(() => {
    // yahan aapka code
```

```
})();
```

Example

```
(function() {
    var message = "Hello, World!";
    console.log(message);
})();
```

Is example mein, message variable sirf IIFE ke andar hi accessible hai. Bahar is variable ko access nahi kiya ja sakta.

Benefits

1. **Encapsulation**: Ye global scope ko clean rakhta hai.
2. **Initialization**: Aap koi initialization code execute kar sakte hain bina kisi variable ko expose kiye.
3. **Private Variables**: Aap private variables bana sakte hain jo sirf IIFE ke andar hi accessible hote hain.

Is tarah se, IIFE ka istemal JavaScript mein code ko organize aur manage karne ke liye kiya jata hai!

Use Cases-

Self-invoked functions (IIFE) ko istemal karne ke kuch main reasons ye hain:

1. Scope Encapsulation

IIFE ke andar declare kiye gaye variables global scope mein nahi aate. Isse variable conflicts se bachne mein madad milti hai.

2. Private Variables

Aap IIFE ke andar private variables bana sakte hain jo sirf us function ke andar accessible hote hain. Ye approach data hiding ke liye useful hota hai.

3. Initialization Code

Kabhi-kabhi aapko koi initialization ka code likhna hota hai jo sirf ek baar execute ho. IIFE aise scenarios ke liye ideal hai.

4. Modularity

IIFE se aap apne code ko modules mein organize kar sakte hain, jisse code ka management aur maintainability asaan hota hai.

5. Avoiding Global Namespace Pollution

JavaScript mein global scope mein variables rakhna accha nahi hota, kyunki ye conflicts create kar sakte hain. IIFE aapke code ko self-contained rakhta hai.

Example

javascript

Copy code

```
var globalVar = "I am global";


(function() {
    var localVar = "I am local";
    console.log(globalVar); // Accessible
})();


console.log(localVar); // ReferenceError: localVar is not defined
```

Is example mein, localVar sirf IIFE ke andar accessible hai, jabki globalVar global scope mein accessible hai.

In sab wajahon se, IIFE ka istemal karke aap apne JavaScript code ko zyada organized aur safe bana sakte hain!

Var Let and Const Difference

var, let, aur const JavaScript mein variables declare karne ke liye istemal hote hain, lekin inke beech mein kuch key differences hain:

1. Scope

- **var**:
  - Function scope: Agar aap var ke saath variable declare karte hain, toh wo sirf us function ke andar accessible hota hai. Agar ye function ke bahar declare kiya gaya hai, toh ye global scope mein chala jata hai.
- **let aur const**:
  - Block scope: Ye variables sirf un blocks ke andar accessible hote hain jahan unhe declare kiya gaya hai (for example, loops ya if statements).

2. Hoisting

- **var**:
    - var ko hoisted kiya jata hai, matlab variable ko function ke top par le jaya jata hai. Iska matlab ye hai ki aap variable ko declare karne se pehle bhi use kar sakte hain, lekin uski value undefined hoti hai.

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

var a = 5;
```

-
- **let aur const**:
    - Ye hoisted hote hain, lekin unhe "temporal dead zone" (TDZ) ke andar rakhte hain. Iska matlab hai ki aap inhe declare karne se pehle access nahi kar sakte, warna ReferenceError milega.

```
console.log(b); // ReferenceError: Cannot access 'b' before initialization

let b = 10;
```

-

3. Mutability

- **var aur let**:
    - Ye dono mutable hain, matlab aap inhe baad mein nayi value assign kar sakte hain.

```
var x = 1;

x = 2; // valid



let y = 1;

y = 2; // valid
```

-
- **const**:
    - Ye immutable hain, matlab aap inhe nayi value assign nahi kar sakte. Lekin agar ye object ya array hain, toh aap us object ke properties ya array ke elements ko modify kar sakte hain.

javascript
Copy code

```
const z = 3;

z = 4; // TypeError: Assignment to constant variable



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

arr.push(4); // valid

-

Summary

- **var**: Function scope, hoisting ke sath undefined, mutable.
- **let**: Block scope, hoisting ke sath TDZ, mutable.
- **const**: Block scope, hoisting ke sath TDZ, immutable (lekin objects/arrays ki properties ko modify karna possible).

Is tarah se, aapko situation ke hisab se var, let, ya const ka istemal karna chahiye!

lekin objects/arrays ki properties ko modify karna possible. kaise possible h

const ke saath declare kiye gaye objects aur arrays ki properties ya elements ko modify karna isliye possible hai kyunki const sirf reference ko immutable banata hai, na ki us object ya array ke andar ki values ko.

Objects

Jab aap ek object ko const ke saath declare karte hain, toh aap us object ki properties ko modify kar sakte hain:

```
const person = {
    name: "Alice",
    age: 25
};

// Modifying a property
person.age = 26; // Valid
console.log(person); // { name: "Alice", age: 26 }

// Adding a new property
person.city = "New York"; // Valid
console.log(person); // { name: "Alice", age: 26, city: "New York" }
```

```
// Trying to reassign the whole object will result in an error

// person = { name: "Bob" }; // TypeError: Assignment to constant variable
```

Arrays

Agar aap ek array ko const ke saath declare karte hain, toh aap array ke elements ko modify ya add kar sakte hain:

javascript

Copy code

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


// Modifying an element

numbers[0] = 10; // Valid

console.log(numbers); // [10, 2, 3]


// Adding an element

numbers.push(4); // Valid

console.log(numbers); // [10, 2, 3, 4]


// Trying to reassign the whole array will result in an error

// numbers = [5, 6, 7]; // TypeError: Assignment to constant variable
```

Summary

- **const** ke saath declare kiye gaye objects aur arrays ki reference ko change nahi kiya ja sakta, lekin inki properties ya elements ko modify karna allowed hai.
- Is wajah se aap const ka istemal karte waqt data integrity ko maintain kar sakte hain, jabki values ko update karne ki flexibility bhi milti hai!

Event delegation

Event delegation ek technique hai jo JavaScript mein events ko handle karne ke liye istemal hoti hai. Is technique ke zariye aap parent elements par events ko attach karte hain, aur un events ko child elements par propagate karte hain. Is tarah, aapko har child element ke liye alag se event listener attach karne ki zarurat nahi hoti.

Kaise Kaam Karta Hai

1. **Event Bubbling**: Jab aap kisi child element par event trigger karte hain (jaise click), toh wo event parent elements ki taraf bubble hota hai. Event delegation is principle ka istemal karta hai.
2. **Parent Element ko Listener Attach Karna**: Aap parent element par ek event listener attach karte hain, aur jab event trigger hota hai, aap check karte hain ki kaunsa child element us event ko trigger kar raha hai.
3. **Efficiency**: Isse performance improve hoti hai, especially jab aapko dynamically elements add ya remove karne hote hain. Aapko sirf ek listener attach karna hota hai, chahe kitne bhi child elements hon.

Example

Maan lijiye, aapke paas ek list hai aur aap chahte hain ki jab kisi list item par click kiya jaye, toh kuch action ho:

```
<ul id="myList">
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>

<script>
    const list = document.getElementById('myList');

    // Event delegation
    list.addEventListener('click', function(event) {
        // Check if the clicked element is an li
        if (event.target.tagName === 'LI') {
            alert('You clicked: ' + event.target.textContent);
        }
    });
</script>
```

Advantages

1. **Performance**: Bahut saare elements ke liye alag-alag listeners attach karne ki zarurat nahi, sirf ek parent listener kaafi hai.
2. **Dynamic Elements**: Agar aap new child elements add karte hain, toh un par bhi event listener automatically work karega.
3. **Cleaner Code**: Code kaafi clean aur manageable hota hai, kyunki aapko multiple listeners manage nahi karne hote.

Summary

Event delegation ek powerful technique hai jo JavaScript mein events ko efficiently handle karne ke liye istemal hoti hai, aur ye aapko better performance aur cleaner code provide karta hai!

var, let, aur const JavaScript me variables declare karne ke liye use hone wale keywords hain, lekin inme kuch key differences hain. Yahan par inke beech ka comparison diya gaya hai:

1. Scope

- **var**:
  - Function scope ya global scope me declare hota hai. Agar var ko function ke andar declare kiya gaya hai, to wo sirf us function ke andar accessible hoga. Agar global scope me declare kiya gaya hai, to wo puri script me accessible hoga.
- **let**:
  - Block scope me declare hota hai. Yeh sirf us block (jaise loop ya if statement) ke andar accessible hoga jahan par yeh define kiya gaya hai.
- **const**:
  - let ki tarah, const bhi block scope me hota hai. Iska matlab hai ki yeh sirf us block ke andar accessible hota hai jahan par yeh define kiya gaya hai.

2. Reassignment

- **var**:
  - Is variable ko kabhi bhi reassign kiya ja sakta hai. Aap var se declared variable ko badal sakte hain.
- **let**:
  - let se declared variable ko bhi reassign kiya ja sakta hai.
- **const**:
  - const se declared variable ko reassign nahi kiya ja sakta. Yeh constant hota hai, lekin agar yeh ek object ya array hai, to uske properties ya elements ko modify kiya ja sakta hai.

3. Hoisting

- **var**:
  - var variables ko hoisted kiya jata hai, lekin unki value undefined hoti hai jab tak actual declaration nahi hota.
- **let**:
  - let variables ko bhi hoisted kiya jata hai, lekin unhe "temporal dead zone" (TDZ) ka samna karna padta hai. Matlab, aap unhe declare hone se pehle access nahi kar sakte.
- **const**:

- const variables ko bhi hoisted kiya jata hai aur unhe bhi TDZ ka samna karna padta hai.

Summary Table

Feature	var	let	const
Scope	Function/global scope	Block scope	Block scope
Reassignment	Yes	Yes	No
Hoisting	Yes (undefined)	Yes (TDZ)	Yes (TDZ)
Declaration	Can be redeclared	Cannot be redeclared	Cannot be redeclared

Conclusion

- **Use var**: Jab aapko function scope me variable declare karna ho aur aapko reassign karne ki flexibility chahiye.
- **Use let**: Jab aapko block scope me variable declare karna ho aur usse reassign karna ho.
- **Use const**: Jab aapko block scope me variable declare karna ho aur aap usse reassign nahi karna chahte.

Modern JavaScript development me, let aur const ka use var ke comparison me zyada recommended hai, kyunki yeh code ko zyada predictable aur maintainable banate hain.

In JavaScript, a **scope chain** refers to the hierarchical structure that determines how variables are accessed in nested functions or blocks. It consists of the current scope and the parent scopes, enabling variable resolution.

Key Points:

1. **Scopes**: There are two main types of scopes in JavaScript:
   - **Global Scope**: Variables declared outside of any function.
   - **Local Scope**: Variables declared within a function or block.
2. **Accessing Variables**: When a variable is accessed, JavaScript looks for it in the current scope. If it's not found, it moves up the scope chain to the parent scope, and continues until it either finds the variable or reaches the global scope.

Example:
javascript

Copy code

```
const globalVar = "I am global";


function outerFunction() {
    const outerVar = "I am outer";


    function innerFunction() {
        const innerVar = "I am inner";
        console.log(innerVar);  // Accesses innerVar
        console.log(outerVar);   // Accesses outerVar
        console.log(globalVar);  // Accesses globalVar
    }


    innerFunction();
}


outerFunction();
```

3.

In this example, innerFunction can access its own variable (innerVar), the variable from its parent function (outerVar), and the global variable (globalVar), showcasing the scope chain in action.

Difference between event delegation and propagation

Event Delegation and **Event Propagation** are both concepts related to JavaScript event handling, but they serve different purposes. Here's a breakdown of their differences:

---

1. Event Delegation

Definition:
Event delegation is a pattern in JavaScript where a single event listener is added to a parent element to manage events for its child elements, even those added dynamically after the listener is attached.

How It Works:
It leverages **event bubbling** (a phase of event propagation) to listen for events triggered on child elements by attaching a single event listener to their common ancestor.

Advantages:

- Reduces memory usage since fewer event listeners are created.
- Simplifies handling events for dynamically created elements.

Example:

javascript

Copy code

```
// Parent container

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



// Add an event listener to the parent

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

    if (event.target.matches('.child')) {

        console.log('Child element clicked:', event.target);

    }

});



// Dynamically adding a child element

const newChild = document.createElement('div');

newChild.className = 'child';

newChild.textContent = 'I am a new child!';

parent.appendChild(newChild);
```

In this example, the click event on dynamically added child elements is still handled by the parent event listener.

---

2. Event Propagation

Definition:
Event propagation refers to the flow of an event through the DOM after it has been triggered. It consists of three phases:

1. **Capture phase**: The event moves from the root element to the target element.
2. **Target phase**: The event reaches the target element.
3. **Bubble phase**: The event moves from the target element back up to the root.

Key Characteristics:

- **Capturing**: Events are intercepted on their way down to the target.
- **Bubbling**: Events are handled on their way back up the DOM tree.
- Most modern browsers support both phases, but bubbling is the default for most event handlers.

Example:

javascript

Copy code

```
// Capturing phase

document.addEventListener('click', () => {

    console.log('Document capturing');

}, true);



// Bubbling phase

document.addEventListener('click', () => {

    console.log('Document bubbling');

}, false);



// Event target

document.getElementById('child').addEventListener('click', () => {
```

```
    console.log('Child element clicked');
});
```

If the child is clicked:

1. "Document capturing" (capture phase)
2. "Child element clicked" (target phase)
3. "Document bubbling" (bubble phase)

Key Differences

Aspect	Event Delegation	Event Propagation
Purpose	Efficiently handle events for multiple child elements	Describes the flow of an event through the DOM
Concept	Uses bubbling to delegate events to a parent element	Includes capture, target, and bubble phases
Use Case	Managing events for dynamic elements	Understanding and controlling event flow in the DOM
Implementation	Add one listener to a parent, use `event.target`	Set listeners at different phases of propagation

By combining both concepts, you can efficiently and effectively manage complex event-driven behaviors in your JavaScript applications.

Explanation in hindi

1. Event Delegation

Event delegation ka matlab hota hai ek **parent element par event listener lagana** aur uske andar ke saare child elements ke events ko handle karna. Yaha pe hum **event bubbling** ka fayda uthate hain, jo allow karta hai ki ek event child se parent tak propagate ho.

Kyu Use Karte Hain?

- Ek-ek child element pe event listener lagane ki zarurat nahi hoti, toh code optimized ho jata hai.
- Agar dynamically koi new child element add hota hai, toh parent listener uska bhi event handle kar sakta hai.

Example:

javascript

Copy code

```
const parent = document.getElementById('parent');


// Parent pe ek hi event listener lagao
parent.addEventListener('click', function (event) {
    if (event.target.matches('.child')) {
        console.log('Child clicked:', event.target);
    }
});


// Naya child add karte hain
const newChild = document.createElement('div');
newChild.className = 'child';
newChild.textContent = 'Main naya child hoon!';
parent.appendChild(newChild);
```

Isme hota kya hai?
Agar koi `child` element click kare, toh event listener parent pe laga hone ke bawajood child ka event handle karega.

---

2. Event Propagation

Event propagation matlab jab DOM mein ek event trigger hota hai, toh wo kis tarah flow karega. Iska flow teen phases mein hota hai:

1. **Capture Phase**: Event root (sabse upar waala element) se target (jo element click hua) tak jata hai.
2. **Target Phase**: Event actual target element pe reach karta hai.
3. **Bubble Phase**: Event wapas target se root tak bubble karta hai.

Example:

javascript

Copy code

```
// Capture phase

document.addEventListener('click', () => {

    console.log('Document capture phase');

}, true);



// Bubble phase

document.addEventListener('click', () => {

    console.log('Document bubble phase');

}, false);



// Target element

document.getElementById('child').addEventListener('click', () => {

    console.log('Child clicked');

});
```

Agar child element click hota hai, toh flow kuch aisa hoga:

1. "Document capture phase" (upar se neeche jata hai).
2. "Child clicked" (target phase pe).
3. "Document bubble phase" (neeche se wapas upar jata hai).

Event Delegation vs Event Propagation

Feature	Event Delegation	Event Propagation
Purpose	Parent se child events ko handle karna.	Event ka flow samajhna aur control karna.
Use Case	Jab multiple/dynamic child elements ka event handle karna ho.	Jab event flow ko manipulate karna ho.
Phases	Sirf bubbling phase ka use karta hai.	Capture, target, aur bubble phase sab include hota hai.

Simple Language Mein:

- **Event Delegation**: "Parent bolega, mere sab bacchon ka event mujhe handle karne do."
- **Event Propagation**: "Event ek flow follow karega – upar se neeche aur fir neeche se upar."

Prototype Chaining in JavaScript

Prototype chaining is a feature in JavaScript that allows objects to inherit properties and methods from other objects through their prototypes. It's a mechanism for reusing code and enabling inheritance.

How It Works

1. Every JavaScript object has an internal link (denoted as `[[Prototype]]`) to another object called its **prototype**.
2. If you try to access a property or method on an object and it doesn't exist on that object, JavaScript looks up the chain (through its `[[Prototype]]`) until it finds the property or method or reaches `null`.

Example of Prototype Chaining

javascript

Copy code

```
// Define a constructor function
function Person(name) {
    this.name = name;
}

// Add a method to Person's prototype
Person.prototype.greet = function () {
    return `Hello, my name is ${this.name}`;
};

// Create an instance
const john = new Person('John');

// Accessing the method
console.log(john.greet()); // "Hello, my name is John"

// Prototype chain
console.log(john.__proto__ === Person.prototype); // true
console.log(Person.prototype.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null (end of the chain)
```

In this example:

1. `john` does not have a `greet` method directly.
2. JavaScript looks for `greet` in `john`'s prototype (which is `Person.prototype`).
3. If `Person.prototype` didn't have `greet`, JavaScript would check `Object.prototype`.
4. The chain stops when `null` is reached.

Diagram of Prototype Chain

plaintext

Copy code

```
john --> Person.prototype --> Object.prototype --> null
```

Key Concepts

1. **Prototype Property**:
    - Every function in JavaScript has a `prototype` property, which is an object used to define properties and methods to be inherited by instances created by the constructor.
2. **proto**:
    - An object's `__proto__` is a reference to its prototype (in modern JavaScript, this is referred to as `[[Prototype]]`).
    - You can traverse an object's prototype chain using `__proto__`.
3. **End of the Chain**:
    - The end of the chain is `Object.prototype`, whose prototype is `null`.

Real-World Example

javascript

Copy code

```
// Base object

const animal = {

    eats: true,

    walk() {

        console.log("Animal walks");

    }

};


// Derived object
```

```
const dog = {
    __proto__: animal, // Link dog to animal
    bark() {
        console.log("Dog barks");
    }
};

// Access properties and methods
console.log(dog.eats); // true (inherited from animal)
dog.walk(); // "Animal walks" (inherited from animal)
dog.bark(); // "Dog barks" (own method)

// Prototype chain
console.log(dog.__proto__ === animal); // true
console.log(animal.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null
```

Here:

- `dog` inherits from `animal`, and `animal` inherits from `Object.prototype`.

Important Points

1. **Avoid Overusing Prototype Chaining**:
    - Traversing long prototype chains can impact performance.
2. **Shadowing**:
    - If a property exists on an object itself, it will shadow (override) the prototype's property.

javascript
Copy code

```
dog.eats = false;
```

```
console.log(dog.eats); // false (own property, not inherited)
```

3.
4. **Functions and Prototypes**:
    - Functions themselves are objects and have their own prototype chain.

Benefits of Prototype Chaining

- Code reuse through inheritance.
- Dynamic property lookup (add or modify properties at runtime).

By understanding prototype chaining, you can write efficient, modular, and reusable code in JavaScript!

Explanation in hindi

Prototype chaining ek concept hai JavaScript mein jo allow karta hai ek object ko doosre object ke properties aur methods inherit karne ke liye. Ye inheritance ka ek tarika hai jisme objects ek chain ke through connected hote hain.

Kaise Kaam Karta Hai?

1. Har object ke paas ek hidden property hoti hai `[[Prototype]]` (modern JS mein `__proto__`).
2. Jab aap kisi object par koi property ya method access karte ho aur wo directly us object mein nahi hoti, tab JavaScript uske prototype chain mein search karta hai.
3. Ye chain tab tak continue hoti hai jab tak ya to property mil jaye ya chain `null` tak pahunch jaye.

Example: Prototype Chaining

javascript

Copy code

```
// Constructor function

function Person(name) {

    this.name = name;

}
```

```
// Prototype par method add karte hain
Person.prototype.greet = function () {
    return `Hello, my name is ${this.name}`;
};

// Ek instance banate hain
const john = new Person('John');

// Method call karte hain
console.log(john.greet()); // "Hello, my name is John"

// Prototype chain dekhte hain
console.log(john.__proto__ === Person.prototype); // true
console.log(Person.prototype.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null (chain ka end)
```

Explanation:

1. `john` ke paas directly `greet` method nahi hai.
2. JavaScript `john` ke prototype chain mein `greet` method dhoondta hai (`Person.prototype` mein milta hai).
3. Agar `Person.prototype` mein nahi milta, toh next `Object.prototype` mein check karta.
4. Chain `null` par khatam hoti hai.

Prototype Chain Diagram

plaintext

Copy code

```
john --> Person.prototype --> Object.prototype --> null
```

Real-World Example

javascript

Copy code

```
// Base object
const animal = {
    eats: true,
    walk() {
        console.log("Animal walks");
    }
};

// Derived object
const dog = {
    __proto__: animal, // Link dog to animal
    bark() {
        console.log("Dog barks");
    }
};

// Properties aur methods access karte hain
console.log(dog.eats); // true (inherited from animal)
dog.walk(); // "Animal walks" (inherited from animal)
dog.bark(); // "Dog barks" (own method)

// Prototype chain check
```

```
console.log(dog.__proto__ === animal); // true

console.log(animal.__proto__ === Object.prototype); // true

console.log(Object.prototype.__proto__); // null
```

Explanation:

- `dog` ka prototype `animal` hai, aur `animal` ka prototype `Object.prototype` hai.
- Agar `dog` ke andar `eats` ya `walk` property nahi hoti, toh wo `animal` ke andar search karega.

Key Points

1. **Inheritance Using Prototype**:
    - Ek object doosre ke properties aur methods use kar sakta hai prototype chaining ki wajah se.
2. **Shadowing**:
    - Agar child object ke paas apni property ho, toh wo parent ki property ko shadow kar deta hai.

javascript
Copy code

```
dog.eats = false;

console.log(dog.eats); // false (own property, not inherited)
```

3.
4. **Performance Impact**:
    - Agar chain zyada lambi ho, toh lookup slow ho sakta hai.

Prototype Chaining ka Fayda

- **Code Reuse**: Ek baar method ya property define kar do aur multiple objects use kar sakte hain.
- **Dynamic Property Lookup**: Runtime par properties ya methods add/modify kar sakte hain.

Simple Language Mein:

- **Prototype Chain** ek aisi chain hai jisme ek object doosre object ki properties aur methods inherit karta hai.
- Agar ek object ke andar property/method nahi milti, toh wo uske prototype mein search karta hai.
- Chain tab tak chalegi jab tak ya to property mil jaye ya `null` aa jaye.

Aise hi **prototype chaining** ki wajah se JavaScript inheritance ko simple aur powerful banata hai!