In React, controlled and uncontrolled elements refer to how form data is managed.

✅ **Controlled Components**

React controls the value of the input through state.

You use useState (or this.state in class components).

The input’s value is always determined by React.

Example:

import { useState } from 'react';

function ControlledForm() {

const [name, setName] = useState('');

return (

<input type="text"value={name} onChange={(e) => setName(e.target.value)}

/>

);

}

🔍 Characteristics:

One-way data binding: React → Input

Easier to validate or manipulate input values

Better suited for form submissions or complex forms

🚫 **Uncontrolled Components**

The DOM controls the input value.

You access the value using a ref (reference to the DOM element).

React does not track changes to the value.

Example:

import { useRef } from 'react';

function UncontrolledForm() {

const inputRef = useRef();

const handleSubmit = () => {

alert(inputRef.current.value);

};

return (

<><input type="text" ref={inputRef} />

<button onClick={handleSubmit}>Submit</button>

</> );

}

🔍 Characteristics:

Useful for simple forms or when integrating with non-React code

Harder to validate or keep in sync with app state

🆚 Summary Table:

Feature Controlled Uncontrolled

State management React (useState) DOM (useRef)

Value source React state DOM input value

Validation Easier and inline with React Manual via ref

Use case Complex forms, validation Simple inputs, quick scripts

Do you want to know when to use one over the other in a real-world scenario?

**Derived State in React**

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

Derived state refers to state that is computed from other state or props, rather than being directly stored and updated.

📌 Why Avoid Storing Derived State?

Storing derived values in state can lead to:

Inconsistency if not properly synced

Redundant updates and complexity

Derived state is not stored directly in React — instead, it is calculated from other data like props or state.

example- ageInMonths = age \* 12

ex2-

function Welcome({ firstName, lastName }) {

const fullName = firstName + ' ' + lastName; // this is derived state

return <h1>Hello, {fullName}</h1>;

}

**Spread Operator (...)**

💬 Purpose:

To spread or unpack values from an array or object — into a new array, object, or function arguments.

🔹 1. Spread in Arrays

📌 Example 1: Copying an array

const original = [1, 2, 3];

const copy = [...original];

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

➡ This creates a new array. It’s like cloning — changing copy won’t affect original.

📌 Example 2: Combining arrays

const a = [1, 2];

const b = [3, 4];

const combined = [...a, ...b];

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

🔹 2. Spread in Objects

📌 Example: Copying and adding properties

const obj1 = { x: 10, y: 20 };

const obj2 = { ...obj1, z: 30 };

console.log(obj2); // { x: 10, y: 20, z: 30 }

✅ Use this to clone objects or merge multiple objects.

🔹 3. Spread in Function Calls

📌 Example:

const numbers = [5, 10, 15];

console.log(Math.max(...numbers)); // 15

✅ ...numbers breaks the array into individual arguments like Math.max(5, 10, 15)

⚠️ Caution: Spread creates a shallow copy, not deep

const obj = { a: 1, nested: { b: 2 } };

const copy = { ...obj };

copy.nested.b = 5;

console.log(obj.nested.b); // 5 (it changed!)

Use libraries like lodash or custom deep clone functions for nested structures.

🌀 **Rest Operator (...)**

💬 Purpose:

To gather multiple elements into an array or object — usually in function parameters or destructuring.

🔹 1. Rest in Functions (Variable arguments)

📌 Example:

function add(...nums) {

return nums.reduce((sum, val) => sum + val, 0);

}

console.log(add(1, 2, 3, 4)); // 10

✅ Here, ...nums collects all arguments into an array: [1, 2, 3, 4].

🔹 2. Rest in Array Destructuring

📌 Example:

const [first, ...others] = [10, 20, 30, 40];

console.log(first); // 10

console.log(others); // [20, 30, 40]

🔹 3. Rest in Object Destructuring

📌 Example:

const { name, ...rest } = { name: 'John', age: 25, city: 'Pune' };

console.log(name); // John

console.log(rest); // { age: 25, city: 'Pune' }

🧠 How to Tell the Difference?

Context ... acts as Meaning

Right side (e.g. [...arr], {...obj}) Spread Expands values

Function parameter (e.g. function(...args)) Rest Collects values into array

Destructuring (e.g. const [a, ...b] = arr) Rest Captures remaining items

Operator Position Valid? Example

Spread Start ✅ [...arr, 4]

Middle ✅ [1, ...arr, 4]

End ✅ [1, 2, ...arr]

Rest Start ❌ function(...args, last) ❌

Middle ❌ function(a, ...rest, b) ❌

End ✅ function(a, b, ...rest) ✅

🔹 **var, let, and const: Overview**

**Feature var let const**

Scope Function-scoped Block-scoped ({}) Block-scoped ({})

Reassignable ✅ Yes ✅ Yes ❌ No (constant reference)

Redeclarable ✅ Yes ❌ No ❌ No

Hoisted ✅ Yes (initialized as undefined) ✅ Yes (but not initialized) ✅ Yes (but not initialized)

Temporal Dead Zone ❌ No ✅ Yes ✅ Yes

**1. var – Function Scoped**

function testVar() {

if (true) {

var a = 10;

}

console.log(a); // 10 (accessible outside the block)

}

testVar();

var is accessible outside the block it's defined in (inside the same function).

Redeclaring a var variable does not throw an error.

var x = 1;

var x = 2; // ✅ No error

**✅ 2. let – Block Scoped**

function testLet() {

if (true) {

let b = 20;

console.log(b); // 20

}

// console.log(b); // ❌ ReferenceError: b is not defined

}

testLet();

let is not accessible outside the block {} it's declared in.

Redeclaring let in the same scope throws an error.

let y = 5;

// let y = 10; // ❌ SyntaxError

**✅ 3. const – Block Scoped + Constant**

const z = 30;

z = 40; // ❌ TypeError: Assignment to constant variable

const cannot be reassigned.

But: for objects and arrays, their contents can still change.

const person = { name: "John" };

person.name = "Jane"; // ✅ Allowed

**Hoisting-**

Hoisting is JavaScript’s behavior of moving variable and function declarations to the top of their scope before code execution.

var is hoisted (but value is undefined)

ex-

console.log(a); // undefined

var a = 10;

🔸 This behaves like:

var a; // declaration hoisted

console.log(a); // undefined

a = 10; // assignment happens here

let and const are hoisted too — BUT in Temporal Dead Zone (TDZ)

console.log(b); // ❌ ReferenceError

let b = 10;

Even though b is hoisted, it cannot be accessed before declaration — that's the TDZ.

Same with const:

console.log(c); // ❌ ReferenceError

const c = 20;

What is TDZ (Temporal Dead Zone) in JavaScript?

Temporal Dead Zone (TDZ) is the time between when a variable is hoisted and when it is actually declared (initialized) — during this time, accessing the variable results in a ReferenceError.

🔑 In Simple Words:

TDZ is the phase after a block starts but before the let or const variable is declared — where the variable exists, but cannot be accessed.

🔹 Example of TDZ:

{

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

let a = 5;

}

The variable a is hoisted but not initialized.

So during the TDZ (from block start { to let a = 5), you can't access a.

🔹 Example with var (No TDZ):

{

console.log(b); // ✅ undefined (no error)

var b = 10;

}

var is hoisted and initialized to undefined, so there is no TDZ.

🔹 Real-world use case:

function greet() {

if (true) {

// TDZ starts here for `name`

// console.log(name); // ❌ ReferenceError

let name = "Alice"; // TDZ ends here

console.log(name); // ✅ Alice

}

}

greet();

❗ Why does TDZ exist?

TDZ helps catch errors and enforce good coding practices. Without it:

You might read a variable before it's properly initialized.

It helps avoid unexpected undefined values like with var.

✅ Rule to Avoid TDZ:

Always declare variables (let / const) at the top of their block, before you use them.

strict mode-

Strict mode is a special mode in JavaScript that makes your code more secure, clean, and less error-prone by enforcing stricter parsing and error handling.

Why use "use strict"?

Strict mode:

Prevents accidental bugs

Disallows unsafe actions

Makes debugging easier

❌ Using undeclared variables

"use strict";

x = 10; // ❌ ReferenceError: x is not defined

Behavior Without "use strict" With "use strict"

🛑 Undeclared Variables Allowed (automatic global) ❌ Throws an error

🛑 Duplicate Function Parameters Allowed ❌ Throws an error

🛑 Assignment to Read-only Properties Allowed ❌ Throws an error

🛑 this in Global Functions Refers to global object (window) this is undefined

🛑 Use of with Statement Allowed ❌ Throws an error

🛑 Global eval Variables Allowed Creates new variables only inside eval()

🛑 Delete Variables Allowed ❌ Throws an error

note-

It doesn’t apply automatically — you must write "use strict" explicitly.

Many modern tools like ES6 modules or TypeScript use strict mode by default.

What Are Queues in JavaScript?

JavaScript runs in a single-threaded environment — one thing happens at a time. So, to manage async operations (like setTimeout, promises, etc.), JavaScript uses queues and the event loop.

There are mainly two types of queues:

Macrotask Queue (a.k.a. Task Queue)

Microtask Queue (a.k.a. Job Queue)

The Event Loop

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

The event loop is the mechanism that:

Picks one task from the Call Stack (main execution).

Executes all microtasks after that.

Then moves on to the next macrotask.

Repeats forever.

Macrotask → All Microtasks → Next Macrotask → All Microtasks → ...

1. Macrotask Queue / callback queue

Macrotasks include:

setTimeout()

setInterval()

setImmediate() (Node.js)

DOM Events

Ajax callbacks

UI rendering

🧪 Example:

console.log("Start");

setTimeout(() => {

console.log("Macrotask - setTimeout");

}, 0);

console.log("End");

🧾 Output:

Start

End

Macrotask - setTimeout

The setTimeout goes into the macrotask queue and runs after the current execution and all microtasks.

🔷 2. Microtask Queue

Microtasks include:

Promise.then(), catch(), finally()

queueMicrotask() (manual microtask)

MutationObserver (DOM change tracking)

🧪 Example:

console.log("Start");

Promise.resolve().then(() => {

console.log("Microtask - Promise");

});

setTimeout(() => {

console.log("Macrotask - setTimeout");

}, 0);

console.log("End");

🧾 Output:

Start

End

Microtask - Promise

Macrotask - setTimeout

Even though setTimeout is set for 0ms, promises run first because microtasks run after every synchronous task before any macrotask.

🔄 Full Flow Example:

console.log("script start");

setTimeout(() => {

console.log("setTimeout");

}, 0);

Promise.resolve().then(() => {

console.log("promise1");

}).then(() => {

console.log("promise2");

});

console.log("script end");

🧾 Output:

script start

script end

promise1

promise2

setTimeout

🧠 Why?

script start & script end: synchronous

promise1 & promise2: microtasks

setTimeout: microtask

Summary

JavaScript uses task queues to handle async operations.

Microtasks have higher priority than macrotasks.

After every macrotask, all microtasks are processed before the next macrotask.

Understanding this helps avoid unexpected behavior in asynchronous code.

what is event loop

ChatGPT said:

The event loop is the heart of JavaScript’s concurrency model. It allows JavaScript (which is single-threaded) to handle asynchronous operations like timers, user input, API calls, and promises without blocking the main thread.

🧠 In Simple Words:

The event loop continuously checks:

Is the call stack empty?

Are there tasks (macrotasks or microtasks) waiting to run?

If yes, it pushes the next task into the call stack for execution.

🔁 How the Event Loop Works (Step-by-Step):

🟢 Runs synchronous code (line-by-line).

🔵 Handles microtasks (e.g., .then(), queueMicrotask()).

🔴 Runs one macrotask (e.g., setTimeout, events).

🔁 Repeats steps 2–3 forever.

🔄 Diagram Overview

Call Stack ─────────► Executes one thing at a time

▲

│

Event Loop ─► Checks queues continuously

│

┌──────────┴──────────┐

▼ ▼

Microtask Queue Macrotask Queue

(Promise, etc.) (setTimeout, etc.)

🧪 Example:

console.log('Start');

setTimeout(() => {

console.log('Macrotask');

}, 0);

Promise.resolve().then(() => {

console.log('Microtask');

});

console.log('End');

🧾 Output:

Start

End

Microtask

Macrotask

✅ Why?

Start and End are synchronous.

Promise.then is a microtask → runs before setTimeout.

setTimeout is a macrotask → runs last.

🧠 Why Is the Event Loop Important?

Makes non-blocking I/O possible.

Helps UI stay responsive in the browser.

Enables features like async/await, Promises, and setTimeout.

Curring-

---------

Currying is a technique in functional programming where a function with multiple arguments is transformed into a sequence of functions, each taking a single argument.

ex-

Instead of:

function add(a, b) {

return a + b;

}

You curry it like:

function add(a) {

return function(b) {

return a + b;

};

}

Now you can use it like:

const add5 = add(5); // returns a function

console.log(add5(3)); // 8

✅ Why Use Currying?

Reusability: Create specialized functions easily.

Readability: Express logic more clearly.

Function Composition: Works well with other functional techniques.

ex-

Original Function

function multiply(a, b, c) {

return a \* b \* c;

}

Curried Version

function curriedMultiply(a) {

return function(b) {

return function(c) {

return a \* b \* c;

};

};

}

console.log(curriedMultiply(2)(3)(4)); // 24

ex-

Currying with ES6 Arrow Functions

const curriedMultiply = a => b => c => a \* b \* c;

console.log(curriedMultiply(2)(3)(4)); // 24

Concept Meaning

Currying Transform a function of n arguments → n functions of 1 arg each

Syntax f(a)(b)(c) instead of f(a, b, c)

Benefits Reusability, clarity, functional chaining

ex-

Currying vs Partial Application

Currying: Breaks function into a chain of 1-arg functions.

Partial Application: Fixes some arguments of a function and returns a new function.

// Partial Application

function multiply(a, b) {

return a \* b;

}

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

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

ex-

function sum(a, b, c) {

return a + b + c;

}

function curry(fn) {

return function curried(...args) {

if (args.length >= fn.length) {

return fn.apply(this, args);

} else {

return function(...nextArgs) {

return curried.apply(this, args.concat(nextArgs));

};

}

};

}

const curriedSum = curry(sum);

You pass sum into curry.

curry returns a new function (curriedSum) that allows you to pass arguments one-by-one (or in groups).

console.log(curriedSum(1)(2)(3)); // 6

console.log(curriedSum(1, 2)(3)); // 6

console.log(curriedSum(1)(2, 3)); // 6

High order function-

A Higher-Order Function (HOF) is a function that either:

Takes one or more functions as arguments, or

Returns a function as its result

👉 In short, functions that work with other functions.

Why are they useful?

JavaScript treats functions as first-class citizens — you can pass them around like values (variables, parameters, return values).

This makes HOFs possible and very powerful for abstraction, composition, and reusability.

Examples of Higher-Order Functions

1. setTimeout (takes a function as an argument)

js

Copy

Edit

setTimeout(() => {

console.log('This is delayed');

}, 1000);

Here, setTimeout is a higher-order function — it takes a function as an argument.

2. Array.map() (takes a function)

js

Copy

Edit

const nums = [1, 2, 3];

const doubled = nums.map(n => n \* 2);

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

map() is a higher-order function.

It takes a callback function to apply to each element.

3. Returning a function

js

Copy

Edit

function greet(name) {

return function(message) {

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

};

}

const greetJohn = greet("John");

greetJohn("Welcome!"); // Hello, John. Welcome!

greet() returns a function → so it’s a higher-order function.

4. Creating custom higher-order functions

js

Copy

Edit

function repeat(operation, times) {

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

operation();

}

}

repeat(() => console.log("Doing task..."), 3);

repeat takes a function as input and runs it times times.

Feature Description

Takes function as argument map, filter, reduce, setTimeout, etc.

Returns a function Currying, partial application, factory funcs

Combines or transforms behavior Great for abstraction and reusability

what is factory functions?

A Factory Function is a function that returns a new object (or function) each time it’s called — like a factory producing a new product.

It’s a design pattern used to create multiple similar objects without using classes or constructors.

ex-

function createUser(name, age) {

return {

name,

age,

greet() {

console.log(`Hi, I'm ${name} and I'm ${age} years old.`);

}

};

}

const user1 = createUser('Alice', 25);

const user2 = createUser('Bob', 30);

user1.greet(); // Hi, I'm Alice and I'm 25 years old.

user2.greet(); // Hi, I'm Bob and I'm 30 years old.

Here, createUser is a factory function because it returns a new user object each time it's called.

Why Use Factory Functions?

Simpler than using class or new

Easy to customize objects

Great for code reusability and organization

Can create private variables using closures

Example 2: Factory Function with Closure (private data)

function secretHolder(secret) {

return {

getSecret() {

return secret;

}

};

}

const obj = secretHolder('myPassword123');

console.log(obj.getSecret()); // 'myPassword123'

// `secret` is private — cannot access it directly

The secret variable is private — it’s only accessible via the returned function (getSecret), thanks to closures.

Summary

Factory Function: A function that returns a new object or function.

Used to create multiple similar objects without classes or constructors.

Works well with closures, functional patterns, and clean code practices.

Factory Function vs Constructor Function vs Class

Feature Factory Function Constructor (new) Class

Returns Any object this by default this

Uses new keyword ❌ No ✅ Yes ✅ Yes

Can return custom types ✅ Yes ❌ Limited ❌ Limited

Uses closure for private data ✅ Yes ❌ No ❌ (without tricks)

closures-

When a function is defined inside another function, and the inner function uses variables from the outer function — that's a closure.

Even after the outer function is done running, the inner function still has access to the outer function’s variables.

A closure is a function that remembers variables from its outer (lexical) scope, even after that scope has finished executing.

ex-

function outer() {

const outerVar = "I am from outer!";

function inner() {

console.log(outerVar); // inner uses variable from outer

}

return inner;

}

const fn = outer(); // outer() runs and returns inner()

fn(); // logs: "I am from outer!"

inner() closes over the variable outerVar.

Even though outer() has finished executing, inner() still remembers outerVar — that’s a closure.

Real-World Analogy

Think of a closure like a backpack 🎒.

When a function is created, it carries a backpack that holds all the variables from where it was created.

Even if it's used somewhere else later, it still has access to those variables in its backpack.

Use Cases of Closures

1. Data Privacy / Encapsulation

function createCounter() {

let count = 0;

return function() {

count++;

return count;

};

}

const counter1 = createCounter();

console.log(counter1()); // 1

console.log(counter1()); // 2

const counter2 = createCounter();

console.log(counter2()); // 1

count is private — it’s accessible only to the returned function.

2. Partial Application

function multiply(x) {

return function(y) {

return x \* y;

};

}

const double = multiply(2); // x = 2

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

The inner function remembers x = 2 — that's closure in action.

3. Callbacks and Asynchronous Code

function delayedGreeting(name) {

setTimeout(() => {

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

}, 1000);

}

delayedGreeting("Alice");

Even though setTimeout runs after 1 second, the inner arrow function still has access to name due to closure.

Concept Explanation

Closure Inner function remembers variables from outer scope

Scope Created at the time function is defined

Use Cases Data hiding, factory functions, async logic

**Lexical scope/enivronment-**

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

lexical scope (also known as static scope) refers to how variable scope is determined by the physical location of the code in the source file — specifically, where variables and blocks of code are written.

**Key Concepts of Lexical Scope**

Scope is determined at compile time, not runtime.

Nested functions have access to variables declared in their outer (parent) scopes.

Scope does not change based on where a function is called, only where it is defined.

function outer() {

let a = 10;

function inner() {

console.log(a); // inner can access 'a' because it's defined within 'outer'

}

inner();

}

outer();

Here:

a is in the lexical environment of outer.

inner() is defined inside outer(), so it has access to a.

**Lexical Environment**

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

A lexical environment is a structure that holds:

Variable identifiers as keys

Variable values

A reference to its outer lexical environment (like a parent scope)

So when JavaScript executes code, it looks up variables in the current lexical environment and, if not found, goes up the scope chain.

**Summary:**

Term Meaning

Lexical Scope Scope defined by the placement of functions and blocks in source code

Lexical Environment Internal structure that tracks identifiers and their values

Scope Chain Chain of lexical environments used to resolve variables

**call() ,apply() and bind() method-**

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

All three methods are used to manipulate the value of this inside a function.

call()

Calls a function immediately, with a given this and arguments passed individually.

function greet(greeting, punctuation) {

console.log(`${greeting}, ${this.name}${punctuation}`);

}

const person = { name: "Alice" };

greet.call(person, "Hello", "!"); // Hello, Alice!

this → refers to person

"Hello" and "!" are passed as individual arguments

🔹 apply()

Same as call(), but arguments are passed as an array.

greet.apply(person, ["Hi", "."]); // Hi, Alice.

Useful when you already have arguments in an array

🔹 bind()

Does NOT call the function immediately.

Instead, it returns a new function with this permanently bound.

const greetAlice = greet.bind(person, "Hey");

greetAlice("?"); // Hey, Alice?

bind() is great for later use or callbacks.

ex-

const user = { name: "Bob" };

function say(message) {

console.log(`${message}, ${this.name}`);

}

// call

say.call(user, "Hello"); // Hello, Bob

// apply

say.apply(user, ["Hi"]); // Hi, Bob

// bind

const sayHiToBob = say.bind(user, "Hey");

sayHiToBob(); // Hey, Bob

Summary Table

Method Calls the Function? this Set To Arguments Passed As

call() ✅ Immediately First argument Individually: arg1, arg2

apply() ✅ Immediately First argument Array: [arg1, arg2]

bind() ❌ Returns a new function First argument Can be pre-set or passed later

When to Use?

Use Case Method

Immediate function call call() / apply()

Pre-binding a function for later use bind()

Arguments already in an array apply()

first class function-

In JavaScript, functions are first-class citizens.

This means functions can be:

Stored in variables

Passed as arguments to other functions

Returned from functions

Assigned to object properties (Store them in data structures (like arrays, objects))

ex-

// Assigned to a variable

const greet = function() {

console.log("Hello");

};

// Passed as an argument

function callFn(fn) {

fn(); // "Hello"

}

callFn(greet);

// Returned from a function

function outer() {

return function() {

console.log("I'm inner");

};

}

const inner = outer();

inner(); // "I'm inner"

\*Storing functions in an object or array---

const operations = {

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

subtract: function(a, b) { return a - b; }

};

console.log(operations.add(10, 5)); // Output: 15

const funcs = [

() => console.log("First"),

() => console.log("Second")

];

funcs[1](); // Output: Second

✅ Functions can be stored inside arrays and objects just like values.

Higher-Order Functions (HOF)-

A Higher-Order Function is a function that either:

Takes one or more functions as arguments, OR

Returns a function.

// 1. Takes function as argument

function repeat(n, action) {

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

action(i);

}

}

repeat(3, console.log); // logs 0 1 2

// 2. Returns a function

function multiplier(x) {

return function(y) {

return x \* y;

};

}

const double = multiplier(2);

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

So functions like map, filter, reduce are HOFs.

Why are map, filter, and reduce considered Higher-Order Functions (HOFs)?

Because:

❗️They take other functions as arguments — specifically, callback functions.

🔍 Let's break it down:

🔹 map()

const numbers = [1, 2, 3];

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

// `map` takes a function `(num => num \* 2)` as input

🔹 filter()

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

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

// `filter` takes a function that returns true or false

🔹 reduce()

const numbers = [1, 2, 3];

const sum = numbers.reduce((acc, curr) => acc + curr, 0);

// `reduce` takes a reducer function as its first argument

✅ Summary Reason:

map, filter, and reduce are higher-order functions because they operate on other functions by accepting them as arguments, which is exactly what a higher-order function is defined to do.

What is a Higher-Order Component (HOC)?

A Higher-Order Component is a function that takes a component and returns a new component.

It’s like a wrapper that adds extra functionality or behavior to an existing component without modifying the original one.

📘 Syntax:

const EnhancedComponent = higherOrderFunction(WrappedComponent);

✅ Simple Example:

🎯 Goal: Add logging when a component mounts.

function withLogger(WrappedComponent) {

return function EnhancedComponent(props) {

console.log(`Component ${WrappedComponent.name} is rendering`);

return <WrappedComponent {...props} />;

};

}

🧩 Usage:

function Hello(props) {

return <h1>Hello, {props.name}</h1>;

}

const HelloWithLogger = withLogger(Hello);

// In App.js

<HelloWithLogger name="Alice" />

🖨️ Console Output:

Component Hello is rendering

🧠 Key Points:

Concept Description

Takes a component withLogger(Hello)

Returns a new component EnhancedComponent with extra logic

Does not mutate original It wraps it and adds behavior, doesn’t change Hello itself

Passes props through Always use {...props} to maintain original functionality

🛠 Real-World Use Cases for HOCs

Use Case Example Function

Add logging withLogger(Component)

Inject authentication withAuth(Component)

Handle loading/spinner withLoader(Component)

Reuse form logic withFormValidation(Component)

Add theming or styling withTheme(Component)

⚠️ Caveats of HOCs

Can cause wrapper hell (deep nesting)

Harder to trace in DevTools

Be careful with ref forwarding

Better suited for logic reuse—now mostly replaced by custom hooks in modern React ---

🚀 HOC vs Hook

Feature HOC Hook

Based on Component wrappers Function logic

Reuse logic in Class or function components Function components only

Preferred in modern React ❌ Less preferred ✅ Preferred way now

ex2-

function withUppercase(WrappedComponent) {

return function EnhancedComponent(props) {

const uppercasedName = props.name.toUpperCase();

return <WrappedComponent {...props} name={uppercasedName} />;

};

}

function Greeting({ name }) {

return <h2>Hello, {name}</h2>;

}

const LoudGreeting = withUppercase(Greeting);

// <LoudGreeting name="bob" /> => "Hello, BOB"

What is a Callback Function?

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

A callback function is a function passed as an argument to another function, which is then executed later, usually after some operation or event.

It allows asynchronous or deferred execution of code.

🧠 Think of it like this:

You ask a friend (a function) to call you (your callback function) once they’re done with a task. That’s a callback.

🧪 Basic Example

function greetUser(name, callback) {

console.log("Hello " + name);

callback(); // <-- this is the callback function

}

function sayBye() {

console.log("Goodbye!");

}

greetUser("Alice", sayBye);

🖨️ Output:

Hello Alice

Goodbye!

🔁 Real-Life Analogy

Imagine you're baking a cake:

function bakeCake(callback) {

console.log("Baking the cake...");

setTimeout(() => {

console.log("Cake is ready!");

callback(); // Notify when cake is ready

}, 2000);

}

function serveCake() {

console.log("Serving the cake.");

}

bakeCake(serveCake);

🖨️ Output after 2 seconds:

Baking the cake...

Cake is ready!

Serving the cake.

📚 Where are Callbacks used?

1. Array methods like map, filter, forEach use callbacks:

[1, 2, 3].forEach(function (num) {

console.log(num \* 2);

});

2. Event Handling:

button.addEventListener("click", function () {

alert("Button clicked!");

});

3. Asynchronous APIs:

setTimeout(function () {

console.log("Executed after 1 second");

}, 1000);

🚫 Callback Hell

When callbacks are nested deeply, it becomes messy and hard to maintain: called as pyramid of doom

doTask1(function () {

doTask2(function () {

doTask3(function () {

console.log("All tasks done");

});

});

});

✅ Solution: Use Promises or async/await.

✅ Summary

Feature Description

What is it? A function passed into another function as an argument

When used? After some task/event completes

Used in setTimeout, event listeners, APIs, map, filter, etc.

Problems Callback hell in deeply nested logic

Solution Promises and async/await

What is a Promise?

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Syntax:

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

// async task

});

✅ States of a Promise:

State Meaning

pending Initial state, not fulfilled/rejected

fulfilled Operation completed successfully

rejected Operation failed

🧪 Example:

const fetchData = () => {

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

setTimeout(() => {

resolve("Data received!");

// Or reject("Failed to fetch data");

}, 2000);

});

};

fetchData()

.then((data) => console.log(data)) // ✅ success

.catch((err) => console.error(err)); // ❌ failure

Output after 2 seconds:

Data received!

🔹 2. Why use Promises?

They solve the problem of callback hell, by making async code cleaner and more readable.

🔹 3. What is async/await?

async/await is syntactic sugar over Promises that lets you write asynchronous code like it's synchronous.

🧪 Example with async/await:

const fetchData = () => {

return new Promise((resolve) => {

setTimeout(() => {

resolve("Fetched with async/await!");

}, 2000);

});

};

async function getData() {

try {

const result = await fetchData();

console.log(result);

} catch (err) {

console.error(err);

}

}

getData();

Output after 2 seconds:

Fetched with async/await!

✅ Comparison: .then() vs async/await

With .then():

fetchData().then((data) => {

console.log(data);

});

With async/await:

const data = await fetchData();

console.log(data);

✅ Summary Table

Feature Promise async/await

Syntax style Chained with .then() Looks like synchronous code

Error handling .catch() try...catch block

Cleaner code ❌ Can be messy with nesting ✅ Very clean and readable

Return value Promise object Actual resolved value

**promise chaining-**

Promise chaining in JavaScript is a way to execute multiple asynchronous operations in sequence, where each operation starts when the previous one succeeds. You use .then() to chain promises.

ex-

fetch('https://api.example.com/user')

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

.then(user => {

console.log('User:', user);

return fetch(`https://api.example.com/orders?userId=${user.id}`);

})

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

.then(orders => {

console.log('Orders:', orders);

})

.catch(error => {

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

});

***another example-***

createOrder(cart)

.then((orderId)=>proceedToPayment(orderId))

.then((paymentInfo)=>showOrderSummary(paymentInfo))

.then((paymentInfo)=>updateWalletBalance(paymentInfo));

**Explanation**

fetch() returns a Promise.

The first .then() gets the response and parses it as JSON.

The second .then() uses the parsed data to make another API call.

Another .then() parses the second response.

A .catch() at the end handles any error that occurs in the chain.

**Good Practices**

Always return a value or a promise inside .then() if you want to continue chaining.

Use .catch() once at the end to handle errors for the entire chain.

Prefer async/await for better readability in complex cases (modern alternative to chaining).

**example using async/await-**

async function getUserAndOrders() {

try {

const userRes = await fetch('https://api.example.com/user');

const user = await userRes.json();

const orderRes = await fetch(`https://api.example.com/orders?userId=${user.id}`);

const orders = await orderRes.json();

console.log('User:', user);

console.log('Orders:', orders);

} catch (error) {

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

}

}

getUserAndOrders();

**What is fetch()?**

fetch() is a built-in JavaScript function used to make HTTP requests (like GET, POST, PUT, DELETE) and handle responses — typically used to fetch data from APIs.

It returns a Promise.

fetch(url, options)

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

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

.catch(error => console.error('Error:', error));

fetch("https://jsonplaceholder.typicode.com/posts", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({

title: "My Post",

body: "Post content",

userId: 1,

}),

})

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

.then(data => console.log("Created:", data))

.catch(error => console.error("Error:", error));

Using fetch() with async/await (cleaner)

async function getPost() {

try {

const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");

const data = await response.json();

console.log(data);

} catch (error) {

console.error("Fetch error:", error);

}

}

getPost();

Common Options in fetch()

Option Description

method HTTP method: GET, POST, PUT, DELETE

headers Headers to send (e.g., Content-Type)

body Data to send (usually JSON for POST/PUT)

mode CORS, no-cors, same-origin

fetch() does not reject on HTTP errors (like 404). You must check response.ok.

if (!response.ok) {

throw new Error("Network response was not OK");

}

**Prototype in javascript-**

In JavaScript, everything is an object (or can behave like one), and each object has a hidden property called [[Prototype]], which points to another object. When you use a function as a constructor (using new), JavaScript sets the newly created object’s [[Prototype]] to the function’s prototype object.

The prototype property is used mainly to:

Share methods and properties across all instances of an object.

Enable inheritance between constructor functions.

**The Role of prototype**

**1. Creating a Constructor Function**

function Person(name, age) {

this.name = name;

this.age = age;

}

This defines a function called Person. If you use new Person("Alice", 25), it creates a new object and sets its internal [[Prototype]] (accessible via \_\_proto\_\_) to Person.prototype.

**2. Adding Methods to the Prototype**

Person.prototype.sayHello = function() {

console.log(`Hi, I'm ${this.name}`);

};

Here, instead of adding the method inside the constructor (which would create a new copy for each instance), it’s added to the Person.prototype, so all instances share the same function.

const alice = new Person("Alice", 25);

alice.sayHello(); // Output: Hi, I'm Alice

const bob = new Person("Bob", 30);

bob.sayHello(); // Output: Hi, I'm Bob

Both alice and bob have access to sayHello() through the prototype chain.

**The Prototype Chain**

When JavaScript looks for a property or method:

It checks the object itself.

If not found, it checks the object's \_\_proto\_\_ (which is Person.prototype in this case).

This continues up the chain (e.g., Object.prototype, then null).

console.log(alice.\_\_proto\_\_ === Person.prototype); // true

console.log(Person.prototype.\_\_proto\_\_ === Object.prototype); // true

This chain is called the prototype chain.

**Inheritance with Prototypes**

**Suppose we have a base class:**

function Animal(name) {

this.name = name;

}

Animal.prototype.speak = function() {

console.log(`${this.name} makes a sound.`);

};

**And we want to create a subclass Dog:**

function Dog(name, breed) {

Animal.call(this, name); // Call the parent constructor

this.breed = breed;

}

// Inherit the prototype

Dog.prototype = Object.create(Animal.prototype);

// Fix the constructor reference

Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {

console.log(`${this.name} barks!`);

};

const dog = new Dog("Buddy", "Labrador");

dog.speak(); // Buddy makes a sound.

dog.bark(); // Buddy barks!

**Key Points:**

Object.create(Animal.prototype) sets up inheritance.

Dog.prototype.constructor = Dog resets the constructor reference.

dog has access to both speak and bark through the prototype chain.

**Prototype vs \_\_proto\_\_**

prototype is a property on constructor functions.

\_\_proto\_\_ is an internal reference on all objects, pointing to the object’s prototype.

function Car() {}

const car1 = new Car();

console.log(car1.\_\_proto\_\_ === Car.prototype); // true

console.log(Car.prototype.constructor === Car); // true

**Why Use prototype?**

Memory-efficient: Methods are shared, not duplicated for each object.

Enables inheritance.

Core to how JavaScript OOP works behind the scenes.

**Behind the Scenes of class (ES6+)**

When you write:

class User {

constructor(name) {

this.name = name;

}

greet() {

console.log(`Hi, I'm ${this.name}`);

}

}

It is syntactic sugar for:

function User(name) {

this.name = name;

}

User.prototype.greet = function() {

console.log(`Hi, I'm ${this.name}`);

};

🧪 Summary

Concept Description

prototype Property of a constructor function used to define shared methods

\_\_proto\_\_ Internal property of all objects pointing to their constructor’s prototype

Prototype Chain A lookup mechanism that links objects and their prototypes

Object.create() Creates a new object with a specified prototype

**What is \_\_proto\_\_?**

\_\_proto\_\_ is a special internal property of every JavaScript object that points to the prototype of the constructor function that created it.

In other words:

✅ \_\_proto\_\_ is the link to the object's prototype chain, allowing inheritance of properties and methods.

**Why \_\_proto\_\_ Matters**

JavaScript uses \_\_proto\_\_ when looking up properties:

First, it looks in the object itself.

If not found, it looks at \_\_proto\_\_ (i.e., the prototype).

This continues up the prototype chain until the property is found or it reaches null.

function Animal(name) {

this.name = name;

}

Animal.prototype.speak = function() {

console.log(`${this.name} makes a sound.`);

};

const dog = new Animal("Rex");

// Where is 'speak' found?

dog.speak(); // Looks in dog → not found → checks dog.\_\_proto\_\_ (which is Animal.prototype) → found!

console.log(dog.\_\_proto\_\_ === Animal.prototype); // true

dog --> \_\_proto\_\_ --> Animal.prototype --> \_\_proto\_\_ --> Object.prototype --> \_\_proto\_\_ --> null

**Difference Between prototype and \_\_proto\_\_**

Feature prototype \_\_proto\_\_

Belongs to Constructor functions All objects

Type Property Internal reference

Purpose Defines shared methods Links to parent prototype

Used for Inheritance setup Property lookup

**Note on Modern Usage**

\_\_proto\_\_ is widely supported but is not recommended for use in production code.

Use Object.getPrototypeOf(obj) or Object.setPrototypeOf(obj, proto) instead.

const proto = Object.getPrototypeOf(dog);

console.log(proto === Animal.prototype); // true

**another example for prototype chain-**

The prototype chain is a series of links between JavaScript objects, which allows property and method inheritance.

When you try to access a property or method on an object, JavaScript:

Looks for the property on the object itself.

If not found, it looks at the object's [[Prototype]] (aka \_\_proto\_\_).

It continues walking up the prototype chain until:

It finds the property/method.

Or it reaches null (the end of the chain).

function Animal(name) {

this.name = name;

}

Animal.prototype.sayHello = function () {

console.log(`Hello, I'm ${this.name}`);

};

const dog = new Animal("Rex");

dog

│

└──> dog.\_\_proto\_\_ (== Animal.prototype)

│

└──> Animal.prototype.\_\_proto\_\_ (== Object.prototype)

│

└──> Object.prototype.\_\_proto\_\_ == null (end of chain)

So, if you call dog.toString(), here's what happens:

dog has no toString method → check Animal.prototype

Animal.prototype has no toString → check Object.prototype

Object.prototype has toString → it gets called.

console.log(dog.\_\_proto\_\_ === Animal.prototype); // true

console.log(Animal.prototype.\_\_proto\_\_ === Object.prototype); // true

console.log(Object.prototype.\_\_proto\_\_ === null); // true

**Inheritance via Prototype Chain**

**Example with Two Constructors**

function Animal() {}

Animal.prototype.eat = function () {

console.log("eating...");

};

function Dog() {}

Dog.prototype = Object.create(Animal.prototype); // Inherit

Dog.prototype.bark = function () {

console.log("barking...");

};

const d = new Dog();

**Chain:**

d → Dog.prototype → Animal.prototype → Object.prototype → null

So:

d.bark() comes from Dog.prototype

d.eat() comes from Animal.prototype

d.toString() comes from Object.prototype

⚙️ How JavaScript Uses the Prototype Chain

When you do:

d.eat();

JavaScript performs:

d.hasOwnProperty('eat') → false

→ d.\_\_proto\_\_.hasOwnProperty('eat') → false

→ d.\_\_proto\_\_.\_\_proto\_\_.hasOwnProperty('eat') → true ✅

→ Use it!

✅ Summary Table

Term Meaning

\_\_proto\_\_ The internal prototype link of any object

.prototype The property of a constructor function (used to define inherited members)

Object.prototype Top of the prototype chain

null End of prototype chain

**1. Array Destructuring**

✅ Basic Syntax

const numbers = [1, 2, 3];

const [a, b, c] = numbers;

console.log(a); // 1

console.log(b); // 2

console.log(c); // 3

🔄 Skip Elements

const [first, , third] = [10, 20, 30];

console.log(first); // 10

console.log(third); // 30

🧰 **Default Values**

const [x = 5, y = 7] = [1];

console.log(x); // 1

console.log(y); // 7

📚 Swap Variables

let p = 1, q = 2;

[p, q] = [q, p];

console.log(p, q); // 2 1

💡 Rest Operator

const [head, ...rest] = [1, 2, 3, 4];polyfill

console.log(head); // 1

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

**🧱 2. Object Destructuring**

✅ Basic Syntax

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

const { name, age } = person;

console.log(name); // Alice

console.log(age); // 25

✏️ Rename Variables

const user = { id: 101, isAdmin: true };

const { id: userId, isAdmin: isUserAdmin } = user;

console.log(userId); // 101

console.log(isUserAdmin); // true

🧰 Default Values

const settings = { theme: "dark" };

const { theme, fontSize = 16 } = settings;

console.log(theme); // dark

console.log(fontSize); // 16

💡 Rest Properties

const { a, b, ...others } = { a: 1, b: 2, c: 3, d: 4 };

console.log(a); // 1

console.log(others); // { c: 3, d: 4 }

🎯 Destructuring in Function Parameters

✅ Array Example

function sum([a, b]) {

return a + b;

}

console.log(sum([5, 10])); // 15

✅ Object Example

function greet({ name, age }) {

console.log(`Hello ${name}, you are ${age} years old.`);

}

greet({ name: "Bob", age: 30 });

🔄 Nested Destructuring

🧱 Objects

const user = {

name: "John",

address: {

city: "Mumbai",

zip: 400001

}

};

const { address: { city } } = user;

console.log(city); // Mumbai

📦 Arrays

const colors = ["red", ["green", "blue"]];

const [primary, [secondary, tertiary]] = colors;

console.log(secondary); // green

✅ Summary Table

Feature Array Object

Basic syntax [a, b] = array {a, b} = object

Skip values [a, , c] N/A

Rename N/A {a: alias}

Default values [a = 1] {a = 1}

Rest pattern [a, ...rest] {a, ...rest}

Nested destructuring [[a], b] {a: {b}}

**Debouncing -**

Debouncing is a technique to limit how often a function is executed.

It ensures that a function runs only once after a specified delay — after the last time it was called.

🧠 **Why Use Debouncing?**

It’s useful when:

You want to prevent excessive calls to a function

Especially for events that fire rapidly like:

scroll

resize

keyup

search input

📦 Real-World Example

Suppose you're typing in a search box, and you want to fetch results only after the user stops typing for 500ms.

Without debouncing, it might call the API on every keystroke.

📌 Debounce Function Implementation

function debounce(func, delay) {

let timer;

return function (...args) {

clearTimeout(timer); // reset timer every time the function is called

timer = setTimeout(() => {

func.apply(this, args); // call the original function after delay

}, delay);

};

}

🧪 How to Use

function handleSearchInput(e) {

console.log("Searching for:", e.target.value);

}

const debouncedSearch = debounce(handleSearchInput, 500);

document.getElementById("searchBox").addEventListener("input", debouncedSearch);

So now:

If the user types “hello”

The function won’t run on each letter

It runs only after the user stops typing for 500ms

⏱ Timeline Example

User types: h e l l o

Time: ↑ ↑ ↑ ↑ ↑

Each key resets the timer.

Only when user pauses for 500ms, `handleSearchInput` runs.

🆚 **Debounce vs Throttle**

**Feature Debounce Throttle**

Executes after user stops triggering at regular intervals

Use for search input, auto-save scroll, resize

Limits frequency after burst frequency continuously

✅ **Summary**

Debounce: Waits until no more calls come for a specified time.

Reduces function call rate and improves performance.

Ideal for text inputs, autocomplete, or resizing events.

**Throttling-**

Throttling ensures that a function is called at most once every X milliseconds, no matter how many times it's triggered.

📌 It limits the execution rate of a function — useful when an event happens frequently and you don’t want to handle it every time.

**Why Use Throttling?**

It's especially helpful for:

scroll events

resize events

mousemove events

API polling

Button spamming prevention

🔁 **Real-Life Analogy**

**Imagine a gate that only lets one person through every 5 seconds, even if there's a crowd outside.**

**That’s throttling — controlling the rate of execution.**

📦 **Throttle Function Implementation**

Here’s a simple version of a throttle function:

function throttle(func, delay) {

let lastCall = 0;

return function (...args) {

const now = Date.now();

if (now - lastCall >= delay) {

lastCall = now;

func.apply(this, args);

}

};

}

🧪 Example: Throttle Scroll Event

function onScroll() {

console.log("Scroll event:", new Date().toLocaleTimeString());

}

const throttledScroll = throttle(onScroll, 1000);

window.addEventListener("scroll", throttledScroll);

Even if the user scrolls rapidly, the function runs only once per second.

📊 Timeline Illustration

Let's say throttle(func, 1000) is used:

User action: ░░░░░░░░░░░░░░░░░░░░ (scrolling rapidly)

Function call: ● ● ● ● (every 1000ms)

Only one function call per second, no matter how many scroll events happen.

**🧠 Debounce vs Throttle**

**Feature Debounce Throttle**

Executes After user stops action At intervals during continuous action

Use Cases Search box, auto-save, input validation Scroll, resize, button click limiter

Behavior Delays execution Limits frequency

Goal Wait for calm Control rate

**Function Statement vs Function Expression**

🔹 Function Statement (aka Function Declaration)

This is when you define a function using the function keyword with a name.

function greet() {

console.log("Hello!");

}

greet(); // ✅ Works fine

✅ Characteristics:

Hoisted (you can call it before it's defined in code).

Named function.

greet(); // ✅ Works

function greet() {

console.log("Hello!");

}

🔹 **Function Expression**

This is when a function is assigned to a variable.

const greet = function () {

console.log("Hi there!");

};

greet(); // ✅ Works only after definition

⚠️ Characteristics:

Not hoisted like function statements.

Can be anonymous (without a name).

Useful for passing functions as arguments or defining conditionally.

greet(); // ❌ Error: greet is not a function

const greet = function () {

console.log("Hello!");

};

✅ **Summary: Function Statement vs Expression**

Feature Function Statement Function Expression

Syntax function name() {} const name = function() {}

Name required? Yes Optional (can be anonymous)

Hoisted? Yes No

Use case Reusable global functions Assigning to variables, passing as arguments

✅ 2. **Function Parameters vs Arguments**

🔹 **Parameters**

Placeholders used in the function definition.

Think of them like variables waiting for values.

function add(a, b) {

return a + b;

}

Here, a and b are parameters.

**🔹 Arguments**

Actual values you pass when calling the function.

add(3, 5); // 3 and 5 are arguments

✅ Summary: Parameters vs Arguments

Term Where Used Example

Parameters In function definition function sum(x, y)

Arguments In function call sum(10, 20)

**🧪 Bonus Tip: Extra or Missing Arguments**

JavaScript is flexible:

function test(a, b) {

console.log(a, b);

}

test(1); // 1 undefined

test(1, 2, 3); // 1 2 (extra is ignored)

You can access all passed arguments using the arguments object (or rest parameters ...args).

**children props in react-**

const Card = (props) => {

return (

<div className="card">

{props.children}

</div>

);

};

const App = () => {

return (

<Card>

<h2>Title</h2>

<p>This is some content inside the card!</p>

</Card>

);

};

In the Card component, props.children represents whatever is placed inside <Card>...</Card>.

children prop A special prop that holds nested JSX

How children works:

React automatically assigns any nested JSX to the children prop.

It can be:

A single element

Multiple elements

Text

A function (in some advanced use cases)

Why use children?

It helps in:

Creating layout or wrapper components.

Reusing component structures.

Composing UIs dynamically.

**What is Prop Drilling in React?**

📌 Definition:

Prop Drilling is the process of passing data (props) from a parent component down to deeply nested child components through intermediate components — even if those intermediate components don't use the data.

🔧 Example of Prop Drilling

// App.js

import React from 'react';

import Parent from './Parent';

function App() {

return <Parent username="John" />;

}

export default App;

// Parent.js

import React from 'react';

import Child from './Child';

const Parent = ({ username }) => {

return <Child username={username} />;

};

export default Parent;

// Child.js

import React from 'react';

const Child = ({ username }) => {

return <p>Hello, {username}!</p>;

};

export default Child;

❌ Problem:

Here, the username prop is passed through Parent even though Parent doesn’t use it. This becomes hard to manage as the component tree grows.

✅ Solution:

Use Context API or state management tools (like Redux) to avoid prop drilling.

**Why does useState not reflect the updated value immediately?**

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

When you update a state using useState, the updated value is not available immediately in the next line.

This is because:

State updates in React are asynchronous and batched for performance.

🔍 Example

const [count, setCount] = useState(0);

function handleClick() {

setCount(count + 1);

console.log(count); // ❌ Still prints the old value

}

Here, setCount(count + 1) schedules a state update, but it doesn’t update count right away. So console.log(count) still shows the old value.

🔄 What's really happening?

setCount(count + 1) tells React: “Hey, I want to update count.”

React schedules the update to run after the current render finishes.

On the next render, count will have the new value.

✅ How to get the updated value correctly?

1. Use useEffect to react to state changes:

useEffect(() => {

console.log("Updated count:", count);

}, [count]);

This runs after the count state is updated and the component is re-rendered.

2. Or use a function to update state:

setCount(prevCount => {

console.log("Previous count:", prevCount); // ✅ Correct value

return prevCount + 1;

});

Using the function form of setCount() gives you the latest state.

🧠 Summary

❓ Question ✅ Answer

Why doesn't updated state reflect immediately? Because useState updates are asynchronous and batched

How to see updated state? Use useEffect() or setState(prev => ...)

Does console.log() immediately after setCount() work? ❌ No, it logs the old value

**Scope Chain**

**📘 Definition:**

The **scope chain** is the mechanism that JavaScript uses to **look up variables**. When a variable is used, JavaScript searches for it in the current scope. If it’s not found, it moves **up to the outer scopes** one by one, until it either finds the variable or reaches the **global scope**.

**🔍 Example:**

javascript

CopyEdit

function outer() {

let a = 10;

function inner() {

let b = 20;

console.log(a + b); // ✅ a is found in the outer scope

}

inner();

}

outer();

* inner() has access to its own variable b, and also a from outer().
* If a wasn't found in inner, JavaScript goes to outer() to find it.
* That’s the **scope chain** in action.

**🧠 Lexical Environment**

**📘 Definition:**

A **lexical environment** is a structure that holds variable definitions **at the time and place a function is defined**, **not when it is executed**. It contains:

1. **Local memory** (environment record): where variables, functions, and parameters are stored.
2. A reference to the **outer lexical environment**.

**🔄 Relation with Scope:**

* The **lexical environment** forms the basis of **scope**.
* It is **created when a function is declared**, not when it's called.
* JavaScript uses this environment chain to create the **scope chain** during execution.

**🔍 Example:**

javascript

CopyEdit

function outer() {

let a = 100;

function inner() {

console.log(a); // 🔁 inner uses the lexical environment of outer

}

return inner;

}

const result = outer(); // outer() is called, 'a' is 100

result(); // inner() is executed — still remembers 'a' due to lexical environment

Even though outer() is already finished executing, inner() **remembers** the value of a — this is due to the lexical environment (and it's how **closures** work!).

**🧩 Summary Table**

| **Concept** | **Description** |
| --- | --- |
| **Scope Chain** | The path JavaScript follows to look for variables from inner to outer scopes. |
| **Lexical Environment** | The structure that holds variable/function references where the function was defined. |

**✅ Key Differences**

| **Feature** | **Scope Chain** | **Lexical Environment** |
| --- | --- | --- |
| Created when | Function is executed | Function is defined |
| Purpose | Variable lookup | Maintains variable bindings |
| Chain direction | From inner to outer (execution-time) | From definition context (compile-time) |

**1. Type Conversion vs Type Coercion**

**✅ Type Conversion (Explicit)**

You **manually convert** one type to another.

**Example:**

let num = "123";

let converted = Number(num); // Converts string to number

console.log(typeof converted); // number

**Common conversions:**

* Number("123") → 123
* String(123) → "123"
* Boolean(0) → false

**✅ Type Coercion (Implicit)**

JavaScript **automatically converts types** behind the scenes when performing operations.

**Example:**

let result = "5" + 3;

console.log(result); // "53" → string (number coerced to string)

let result2 = "5" - 3;

console.log(result2); // 2 → number (string coerced to number)

**Why?**

JavaScript is **dynamically typed**, so it tries to help you by converting types during operations like +, ==, or comparisons.

**🔍 Coercion in Comparisons**

console.log(1 == "1"); // true (number coerced to string or vice versa)

console.log(1 === "1"); // false (strict equality, no coercion)

**📌 Summary: Conversion vs Coercion**

| **Feature** | **Type Conversion** | **Type Coercion** |
| --- | --- | --- |
| Who does it? | You (manually) | JavaScript (automatically) |
| Example | Number("5") | "5" + 1 → "51" |
| Control | High | Low |

**🔷 2. Truthy vs Falsy Values**

In JavaScript, every value is **either truthy or falsy** when used in a **boolean context** (like an if statement).

**❌ Falsy Values**

These are values that become false when converted to a boolean:

Boolean(false) // false

Boolean(0) // false

Boolean(-0) // false

Boolean("") // false

Boolean(null) // false

Boolean(undefined) // false

Boolean(NaN) // false

**✅ Truthy Values**

**Everything else** is truthy, including:

Boolean(1) // true

Boolean("hello") // true

Boolean([]) // true

Boolean({}) // true

Boolean(function(){}) // true

**🔍 Example Use:**

let name = "";

if (name) {

console.log("Name is:", name);

} else {

console.log("Name is not provided"); // This runs because "" is falsy

}

**🧠 Real-life Analogy**

* **Truthy**: Something exists or has value.
* **Falsy**: Empty, nothing, or not useful.

**✅ Handy Trick: Boolean()**

You can quickly check if a value is truthy/falsy using:

console.log(Boolean([])); // true

console.log(Boolean(0)); // false

**💡 BONUS TIP: Double NOT !!**

let isValid = !!"hello"; // true

let isEmpty = !!""; // false