Explore More

Subcription : Premium CDAC NOTES & MATERIAL @99

Contact to Join
Premium Group

Click to Join
Telegram Group

For More E-Notes

Join Our Community to stay Updated

TAP ON THE ICONS TO JOIN!

	codewitharrays.in freelance project available to buy contact on 8007592194	
SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

	freelance_Project available to buy contact on 8007592194	
21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+NextJs+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql

41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48	Train Ticket Booking Project	React+Springboot+MySql
49	Quizz Application Project	JSP+Springboot+MySql
50	Hotel Room Booking Project	React+Springboot+MySql
51	Online Crime Reporting Portal Project	React+Springboot+MySql
52	Online Child Adoption Portal Project	React+Springboot+MySql
53	online Pizza Delivery System Project	React+Springboot+MySql
54	Online Social Complaint Portal Project	React+Springboot+MySql
55	Electric Vehical management system Project	React+Springboot+MySql
56	Online mess / Tiffin management System Project	React+Springboot+MySql
57		React+Springboot+MySql
58		React+Springboot+MySql
59		React+Springboot+MySql
60		React+Springboot+MySql

Spring Boot + React JS + MySQL Project List

Sr.No	Project Name	YouTube Link
1	Online E-Learning Hub Platform Project	https://youtu.be/KMjyBaWmgzg?si=YckHuNzs7eC84-IW
2	PG Mate / Room sharing/Flat sharing	https://youtu.be/4P9clHg3wvk?si=4uEsi0962CG6Xodp
3	Tour and Travel System Project Version 1.0	https://youtu.be/-UHOBywHaP8?si=KHHfE_A0uv725f12
4	Marriage Hall Booking	https://youtu.be/VXz0kZQi5to?si=llOS-QG3TpAFP5k7
5	Ecommerce Shopping project	https://youtu.be/vJ_C6LkhrZ0?si=YhcBylSErvdn7paq
6	Bike Rental System Project	https://youtu.be/FIzsAmIBCbk?si=7ujQTJqEgkQ8ju2H
7	Multi-Restaurant management system	https://youtu.be/pvV-pM2Jf3s?si=PgvnT-yFc8ktrDxB
8	Hospital management system Project	https://youtu.be/IynIouBZvY4?si=CXzQs3BsRkjKhZCw
9	Municipal Corporation system Project	https://youtu.be/cVMx9NVyI4I?si=qX0oQt-GT-LR_5jF
10	Tour and Travel System Project version 2.0	https://youtu.be/_4u0mB9mHXE?si=gDiAhKBowi2gNUKZ

Sr.No	Project Name	YouTube Link
11	Tour and Travel System Project version 3.0	https://youtu.be/Dm7nOdpasWg?si=P_Lh2gcOFhlyudug
12	Gym Management system Project	https://youtu.be/J8_7Zrkg7ag?si=LcxV51ynfUB7OptX
13	Online Driving License system Project	https://youtu.be/3yRzsMs8TLE?si=JRI_z4FDx4Gmt7fn
14	Online Flight Booking system Project	https://youtu.be/m755rOwdk8U?si=HURvAY2VnizIyJlh
15	Employee management system project	https://youtu.be/lD1iE3W_GRw?si=Y_jv1xV_BljhrD0H
16	Online student school or college portal	https://youtu.be/4A25aEKfei0?si=RoVgZtxMk9TPdQvD
17	Online movie booking system project	https://youtu.be/Lfjv_U74SC4?si=fiDvrhhrjb4KSlSm
18	Online Pizza Delivery system project	https://youtu.be/Tp3izreZ458?si=8eWAOzA8SVdNwlyM
19	Online Crime Reporting system Project	https://youtu.be/0UlzReSk9tQ?si=6vN0e70TVY1GOwPO
20	Online Children Adoption Project	https://youtu.be/3T5HC2HKyT4?si=bntP78niYH802I7N

Q - 1) What is JavaScript?

JavaScript is a lightweight, interpreted programming language with first-class functions most commonly used as a part of web pages for client-side scripting.

Q - 2) Explain the difference between var, let, and const.

var is function-scoped and can be re-declared/updated. let is block-scoped and can be updated but not re-declared in the same scope. const is block-scoped and cannot be updated or re-declared.

Q - 3) What is Hoisting in JavaScript?

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope.

Q - 4) What are Closures in JavaScript?

A closure is a function that has access to its own scope, the outer function's scope, and the global scope.

Q - 5) Explain event delegation.

Event delegation is a technique where a single event listener is attached to a parent element that manages events triggered by its children, leveraging event bubbling.

Q - 6) What is the event loop in JavaScript?

The event loop continuously checks the call stack and the callback queue, executing queued callbacks when the call stack is empty, ensuring non-blocking asynchronous behavior.

Q - 7) What are JavaScript Promises?

Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value.

Q - 8) What is a callback function?

A callback function is a function passed into another function as an argument, to be executed after the first function completes.

Q - 9) Explain async/await.

async marks a function as asynchronous, and await pauses the execution of an async function until the Promise is resolved.

Q - 10) What is the use of bind, call, and apply?

These methods change the context (this) of a function. call and apply execute the function immediately, while bind returns a new function with the bound context.

Q - 11) Explain the logic for palindrome check.

function isPalindrome(str) { str = str.replace(/[^a-zA-Z0-9]/g, ‘‘).toLowerCase(); return str === str.split(‘‘).reverse().join(‘‘); }

Q - 12) Write a javascript code for finding the missing number in the given array.

function findMissingNumber(arr) { const n = arr.length + 1; const sum = (n * (n + 1)) / 2; return sum - arr.reduce((a, b) => a + b, 0); }

Q - 13) Explain the logic for factorial of the number using recursion.

function fibonacci(n) { if (n <= 1) return n; return fibonacci(n - 1) + fibonacci(n - 2); }

Q - 14) Explain sum of Array Elements using Reduce in JavaScript?

function sumArray(arr) { return arr.reduce((a, b) => a + b, 0); }

Q - 15) Explain about flatten a Nested Array in JavaScript.

function flattenArray(arr) { return arr.flat(Infinity); }

Q - 16) Explain Debounce Function in JS?

function debounce(func, delay) { let timeout; return function(...args)
{ clearTimeout(timeout); timeout = setTimeout(() => func.apply(this, args), delay); }; }

Q - 17) Explain the Throttle function?

function throttle(func, limit) { let inThrottle; return function(...args) { if (!inThrottle)
{ func.apply(this, args); inThrottle = true; setTimeout(() => inThrottle = false, limit); } }; }

Q - 18) How to perform deep clone of an object in JS?

function deepClone(obj) { return JSON.parse(JSON.stringify(obj)); }

Q - 19) How to perform merge of two sorted arrays.

function mergeArrays(arr1, arr2) { return [...arr1, ...arr2].sort((a, b) => a - b); }

Q - 20) WAP to remove Duplicates from Array

function removeDuplicates(arr) { return [...new Set(arr)]; }

Q - 21) WAP to check if Two Strings are Anagrams

function areAnagrams(str1, str2) { const sortedStr1 = str1.split('').sort().join(''); const sortedStr2 = str2.split('').sort().join(''); return sortedStr1 === sortedStr2; }

Q - 22) WAP to check if a number is Prime

function isPrime(num) { if (num <= 1) return false; for (let i = 2; i <= Math.sqrt(num); i++) { if (num % i === 0) return false; } return true; }

Q - 23) WAP to find the sum of digits of a number.

function sumOfDigits(n) { return n.toString().split('').reduce((acc, num) => acc + parseInt(num), 0); }

Q - 24) How to generate random hex color

function randomHexColor() { return '#' + Math.floor(Math.random() * 16777215).toString(16); }

Q - 25) How to reverse a linked list?

function reverseLinkedList(head) { let prev = null; let curr = head; while (curr) { let next = curr.next; curr.next = prev; prev = curr; curr = next; } return prev; }

Q - 26) WAP to find the sum of all elements in a binary tree.

function sumTree(root) { if (!root) return 0; return root.value + sumTree(root.left) + sumTree(root.right); }

Q - 27) Explain the binary search on sorted array.

function binarySearch(arr, target) { let left = 0, right = arr.length - 1; while (left <= right) { const mid = Math.floor((left + right) / 2); if (arr[mid] === target) return mid; else if (arr[mid] < target) left = mid + 1; else right = mid - 1; } return -1; }

Q - 28) WAP to find the intersection of two arrays.

function arrayIntersection(arr1, arr2) { return arr1.filter(value => arr2.includes(value)); }

Q - 29) Explain about event emitter pattern with example.

class EventEmitter { constructor() { this.events = {}; } on(event, listener) { if (! this.events[event]) this.events[event] = []; this.events[event].push(listener); } emit(event, ...args) { if (this.events[event]) this.events[event].forEach(listener => listener(...args)); } }

Q - 30) Find the Longest Substring without Repeating Characters

function longestUniqueSubstring(s) { let maxLength = 0; let start = 0; const seen = new Map();

for (let i = 0; i < s.length; i++) { if (seen.has(s[i])) start = Math.max(start, seen.get(s[i]) + 1); seen.set(s[i], i); maxLength = Math.max(maxLength, i - start + 1); } return maxLength; }

Q - 31) Explain the logic to perform array rotation?

function rotateArray(arr, k) { k %= arr.length; return [...arr.slice(-k), ...arr.slice(0, -k)]; }

Q - 32) WAP to get Nth Fibonacci Number using Dynamic Programming

function fibonacciDP(n, memo = {}) { if (n <= 1) return n; if (n in memo) return memo[n]; memo[n] = fibonacciDP(n - 1, memo) + fibonacciDP(n - 2, memo); return memo[n]; }

Q - 33) WAP to check if a Linked List is Circular

function isCircular(head) { let slow = head, fast = head; while (fast && fast.next) { slow = slow.next; fast = fast.next.next; if (slow === fast) return true; } return false; }

Q - 34) WAP to find the Maximum Product Subarray

function maxProductSubarray(arr) { let maxProduct = arr[0], minProduct = arr[0], result = arr[0]; for (let i = 1; i < arr.length; i++) { const tempMax = maxProduct; maxProduct = Math.max(arr[i], maxProduct * arr[i], minProduct * arr[i]); minProduct = Math.min(arr[i], tempMax * arr[i], minProduct * arr[i]); result = Math.max(result, maxProduct); } return result; }

Q - 35) What is a pure function?

A pure function returns the same output for the same inputs and does not produce side effects.

Q - 36) What are arrow functions, and how are they different from regular functions?

Arrow functions provide a shorter syntax and do not have their own this, arguments, or prototype.

Q - 37) Explain debouncing in JavaScript.

Debouncing ensures that a function is only invoked once after a certain period of inactivity, preventing repeated function calls in quick succession.

Q - 38) What is currying in JavaScript?

Currying is a technique of evaluating a function with multiple arguments as a sequence of functions, each with a single argument.

Q - 39) What is memoization?

Memoization is an optimization technique where expensive function calls are cached, storing the result for future calls with the same inputs.

Q - 40) Explain this in JavaScript.

this refers to the object that is executing the current function. Its value depends on the context of the function call.

Q - 41) What is the difference between call(), apply(), and bind()?

call(): Definition: Invokes a function and allows you to pass arguments individually. How it works: You specify the this value (the context for the function) and pass in the arguments one by one. function greet(greeting, name) { console.log(greeting + ',' + name); }

greet.call(null, 'Hello', 'Alice'); // Output: Hello, Alice Use case: When you want to call a function and pass arguments separately.

apply(): Definition: Similar to call(), but you pass the arguments as an array. How it works: You specify the this value, and the second argument is an array (or array-like object) containing the arguments. function greet(greeting, name) { console.log(greeting + ',' + name); }

greet.apply(null, ['Hello', 'Alice']); // Output: Hello, Alice

Use case: When you have an array of arguments and want to call a function. This is particularly useful when dealing with built-in methods like Math.max() which expect a list of individual arguments.

bind(): Definition: Returns a new function where the this value is bound to a specified object or value. How it works: It doesn't call the function immediately but instead returns a new function with a fixed this value and (optionally) some preset arguments. This new function can be called later. function greet(greeting, name) { console.log(greeting + ',' + name); }

const greetAlice = greet.bind(null, 'Hello', 'Alice'); greetAlice(); // Output: Hello, Alice

Use case: When you want to create a copy of a function with a specific this context or preset arguments that can be reused or called later.

Q - 42) What are prototypes in JavaScript? How does inheritance work with prototypes?

Prototypes in JavaScript In JavaScript, every object has a prototype, which is essentially another object from which it inherits properties and methods. When you try to access a property or method on an object, JavaScript first looks for it on the object itself. If it doesn't find it, it looks at the object's prototype, and so on, up the prototype chain until it either finds the property or reaches the end of the chain (null).

Prototype Object: Every JavaScript function (which includes constructors) has a prototype property that is used to build new objects when the function is used as a constructor. **proto** vs prototype: **proto** is a reference to the prototype object from which the current object inherits. prototype is a property of constructor functions that points to an object that will be used as the prototype for objects created by that constructor. How Inheritance Works with Prototypes JavaScript implements inheritance through the prototype chain. This form of inheritance is called prototypal inheritance.

1. Prototype Chain If an object doesn't have a property or method, JavaScript will look for it in the object's prototype. This is known as the prototype chain. function Animal() { this.isAlive = true; }

Animal.prototype.walk = function() { console.log('Walking...'); };

let dog = new Animal();

console.log(dog.isAlive); // true dog.walk(); // Walking...

2. Constructor Functions and Prototypes When you create an object using a constructor function, the object's **proto** property is set to the constructor function's prototype object.

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

Person.prototype.greet = function() { console.log(`Hello, my name is ${this.name}`); };

let john = new Person('John'); john.greet(); // Hello, my name is John

3. Prototypal Inheritance You can set the prototype of an object to inherit properties and methods from another object. This allows objects to share properties and methods.

const animal = { breathe: function() { console.log("Breathing..."); } };

const mammal = Object.create(animal); mammal.feedMilk = function()
{ console.log("Feeding milk..."); };

const human = Object.create(mammal); human.speak = function()
{ console.log("Speaking..."); };

human.breathe(); // Breathing... human.feedMilk(); // Feeding milk... human.speak(); // Speaking...

4. Inheritance with class Syntax (ES6) The class syntax introduced in ES6 provides a clearer way to implement prototypal inheritance:

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

walk() { console.log(`${this.name} is walking`); } }

class Dog extends Animal { bark() { console.log(`${this.name} is barking`); } }

const rover = new Dog('Rover'); rover.walk(); // Rover is walking rover.bark(); // Rover is barking

Q - 43) What are generator functions? How do they differ from regular functions?

Generator functions are a special type of function in Python that allow you to iterate over data lazily (i.e., one item at a time, as needed) rather than computing everything at once. They are defined using the def keyword like regular functions, but instead of returning values using the return statement, they use the yield keyword.

Key Characteristics of Generator Functions: yield Keyword: Instead of returning a value and exiting, a generator uses yield to produce a value and pause the function's execution. Each time yield is encountered, the function's state (variables, program counter, etc.) is saved. When the generator is called again, it resumes from where it left off, continuing until it encounters another yield or the function terminates. Lazy Evaluation: Generators don't compute values until requested, making them memory efficient. They generate values one at a time and are useful for handling large datasets or infinite sequences. Iterator Protocol: Generator functions automatically implement the iterator protocol. This means they return an iterator object when called, which can be iterated over using a loop or manually with the next() function. State Preservation: Unlike regular functions, which forget everything after returning, a generator function preserves its state between successive calls, allowing it to "remember" where it left off. def count_up_to(n): count = 1 while count <= n: yield count count += 1

Use Cases for Generators: Efficient handling of large data: Process large files or datasets without loading everything into memory. Infinite sequences: For example, generating Fibonacci numbers or prime numbers without explicitly storing the entire sequence.

Q - 44) How do promises work in JavaScript? What is the difference between Promise.all, Promise.race, and Promise.allSettled?

romises in JavaScript are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a cleaner way to handle asynchronous code compared to traditional callbacks.

States of a Promise A Promise can be in one of three states:

Pending: The initial state, neither fulfilled nor rejected. Fulfilled: The operation completed successfully, resulting in a value. Rejected: The operation failed, resulting in a reason (error). Creating a Promise You can create a Promise using the Promise constructor:

const myPromise = new Promise((resolve, reject) => { // Asynchronous operation if (/* operation successful */) { resolve(value); // fulfilled } else { reject(error); // rejected } });

Handling Promises You can handle promises using .then(), .catch(), and .finally() methods:

myPromise .then(value => { // handle fulfilled state }) .catch(error => { // handle rejected state }) .finally(() => { // cleanup actions });

Difference Between Promise Methods Promise.all() Takes an iterable of Promises and returns a single Promise that resolves when all of the Promises have resolved, or rejects with the reason of the first Promise that rejects. Returns an array of results in the order of the Promises. Promise.all([promise1, promise2]) .then(results => { // handle results }) .catch(error => { // handle the first error });

Promise.race()

Takes an iterable of Promises and returns a single Promise that resolves or rejects as soon as one of the Promises in the iterable resolves or rejects, with its result or error. Promise.race([promise1, promise2]) .then(result => { // handle the first resolved Promise }) .catch(error => { // handle the first rejected Promise });

Promise.allSettled()

Takes an iterable of Promises and returns a single Promise that resolves after all of the Promises have settled (each may resolve or reject). Returns an array of objects that each describe the outcome of each Promise. Promise.allSettled([promise1, promise2]) .then(results => { // handle each result results.forEach(result => { if (result.status === 'fulfilled') { // handle success } else { // handle failure } }); });

Q - 45) Explain async/await in JavaScript. What are its benefits over promises?

async/await in JavaScript is a syntax that makes working with asynchronous code easier and more readable compared to using promises directly. Here's how it works and its benefits:

Basics of async/await Async Function: To use await, you need to define an async function. This function always returns a promise. async function myFunction() { // function body }

Await Expression: Inside an async function, you can use await before a promise. The code will pause execution until the promise is resolved or rejected.

async function fetchData() { const data = await fetch('https://api.example.com/data'); const json = await data.json(); return json; }

Benefits of async/await over promises Synchronous-like Code: async/await allows you to write asynchronous code that looks and behaves more like synchronous code, which can be easier to read and understand. // Using promises fetchData() .then(data => console.log(data)) .catch(error => console.error(error));

// Using async/await async function main() { try { const data = await fetchData(); console.log(data); } catch (error) { console.error(error); } }

Error Handling: With async/await, you can use standard try/catch blocks for error handling, making it more intuitive than chaining .catch() with promises.

Avoiding Callback Hell: async/await helps avoid deeply nested callbacks, which can occur with promises when you have multiple asynchronous operations that depend on each other.

Control Flow: It provides better control flow, especially when dealing with multiple asynchronous calls that need to be executed in sequence or when waiting for multiple promises.

Q - 46) What is the difference between synchronous and asynchronous programming in JavaScript?

In JavaScript, the difference between synchronous and asynchronous programming primarily revolves around how tasks are executed and how they handle operations that take time, like network requests or file reading.

Synchronous Programming Execution Order: In synchronous programming, tasks are executed one after another. Each task must complete before the next one starts. Blocking: If a task takes a long time (like waiting for a network response), it blocks the execution of subsequent tasks, leading to potential performance issues and unresponsiveness. console.log(“Start”); console.log(“Doing something”); console.log(“End”);

Asynchronous Programming Execution Order: In asynchronous programming, tasks can start and be executed in the background while other tasks continue to run. This allows for more efficient handling of long-running operations. Non-blocking: Asynchronous operations do not block the execution of other code. Instead, you can use callbacks, promises, or async/await to handle the completion of tasks. console.log(“Start”);

setTimeout(() => { console.log(“Doing something after 2 seconds”); }, 2000);

console.log(“End”);

Q - 47) What are higher-order functions in JavaScript?

Higher-order functions in JavaScript are functions that either take one or more functions as arguments or return a function as their result. They are a key feature of functional programming and enable more abstract and flexible code. Here are a few key concepts:

Functions as First-Class Citizens: In JavaScript, functions can be treated like any other value. This means you can pass them around, store them in variables, and return them from other functions. Examples of Higher-Order Functions: Map: Takes a function and applies it to each element of an array, returning a new array const numbers = [1, 2, 3]; const doubled = numbers.map(num => num * 2); // [2, 4, 6]

Filter: Takes a function and returns a new array containing only elements that satisfy the condition defined by the function.

const evens = numbers.filter(num => num % 2 === 0); // [2]

Reduce: Takes a function and reduces an array to a single value by applying the function cumulatively to its elements.

const sum = numbers.reduce((acc, num) => acc + num, 0); // 6

Returning Functions: You can create functions that return other functions.

function multiplier(factor) { return function(x) { return x * factor; }; } const double = multiplier(2); console.log(double(5)); // 10

Q - 48) What is the module pattern in JavaScript, and how does it help in organizing code?

The module pattern in JavaScript is a design pattern that helps in organizing and structuring code by encapsulating functionality within a single object or function. This pattern promotes the creation of modules that can expose specific methods and properties while keeping other parts private, which enhances maintainability and prevents global namespace pollution.

Key Features of the Module Pattern: Encapsulation: You can create private variables and methods that are not accessible from outside the module, protecting the internal state and functionality. Public API: The module can expose specific methods and properties to the outside world, providing a controlled interface for interacting with the module. Reusability: Modules can be reused across different parts of an application or even in different projects, promoting DRY (Don't Repeat Yourself) principles. const CounterModule = (function () { // Private variable let count = 0;

// Private method const increment = function () { count++; };

// Public API return { increment: function () { increment(); console.log(count); }, reset: function () { count = 0; console.log(count); } }; })();

// Usage CounterModule.increment(); // 1 CounterModule.increment(); // 2 CounterModule.reset(); // 0

Benefits of the Module Pattern: Separation of Concerns: Code is organized into distinct modules, making it easier to manage and understand. Reduced Global Scope: By using closures, the module pattern minimizes the risk of naming collisions in the global scope. Testability: Modules can be tested independently, allowing for more straightforward unit testing. Q - 49) What are WeakMap and WeakSet in JavaScript? How do they differ from Map and Set? WeakMap and WeakSet are specialized collections in JavaScript that allow you to store key-value pairs or unique values, respectively, with certain memory management benefits. Here's a breakdown of each and how they differ from Map and Set:

WeakMap Structure: A WeakMap holds key-value pairs where the keys are objects and the values can be any value. Weak References: The keys are held weakly, meaning that if there are no other references to a key object, it can be garbage collected. This prevents memory leaks in cases where the keys are no longer needed. No Iteration: You cannot iterate over a WeakMap (no methods like forEach or iterators). This is due to the weak reference nature, as keys may be garbage collected. WeakSet Structure: A WeakSet is a collection of objects, similar to Set, but only allows objects as values. Weak References: Like WeakMap, the values in a WeakSet are held weakly, allowing for garbage collection if no other references exist. No Iteration: You also cannot iterate over a WeakSet. Differences from Map and Set Memory Management: Map and Set hold strong references to their keys/values, which can lead to memory leaks if objects are no longer in use but still referenced by the collection. In

contrast, WeakMap and WeakSet allow for garbage collection of unreferenced objects. Iteration: Both Map and Set support iteration and can be traversed, while WeakMap and WeakSet do not provide this capability. Use Cases WeakMap: Useful for caching and private data structures where you want to ensure that the keys can be garbage collected when no longer needed. WeakSet: Useful for tracking the existence of objects without preventing their garbage collection.

Q - 50) What are decorators in JavaScript? How would you implement a basic decorator pattern?

Decorators in JavaScript are a design pattern used to add new functionality to an existing object or class without modifying its structure. They are often used for things like logging, access control, and validation.

The basic idea is to wrap an object or a class with another function (the decorator) that adds additional behavior. In JavaScript, decorators are primarily used with classes and functions, especially in frameworks like Angular.

Implementing a Basic Decorator Pattern Here's a simple example to illustrate the decorator pattern:

function logExecutionTime(fn) { return function(...args) { const start = performance.now(); const result = fn(...args); const end = performance.now(); console.log(`Execution time: ${end - start} milliseconds`); return result; }; }

// Usage function computeSquare(num) { return num * num; }

const decoratedComputeSquare = logExecutionTime(computeSquare); decoratedComputeSquare(5); // Logs execution time

Class Decorator:

function addTimestamp(target) { return class extends target { constructor(...args) { super(...args); this.timestamp = new Date(); } }; }

@addTimestamp class MyClass { constructor(name) { this.name = name; } }

const instance = new MyClass('Example'); console.log(instance.timestamp); // Logs the timestamp

Function Decorator: logExecutionTime is a higher-order function that takes a function fn, wraps it, and logs the execution time whenever the function is called. Class Decorator: addTimestamp is a class decorator that extends the original class to add a timestamp property when an instance is created.

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/ccee2025notes

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project