**JAVASCRIPT CONCEPTS**

**Group 1: JavaScript Basics (High Priority)**

1. **What is JavaScript and what is ES6?**

**JavaScript** is a high level, versatile programming language used to make web pages interactive.

**ES6** (ECMAScript 6), released in 2015, is a major update to JavaScript, adding features like let and const for variables, arrow functions, classes, template literals, destructuring, promises for async tasks, and native modules. It’s often called "modern JavaScript" and is widely used today.

1. **What are the data types in JavaScript?**

**Primitive Data Types**

1. **Number**: Represents numeric values, both integers and floating-point (e.g., 42, 3.14).
2. **String**: Represents text or sequences of characters (e.g., "hello").
3. **Boolean**: Represents true or false values (true, false).
4. **Undefined**: A variable that has been declared but not assigned a value (undefined).
5. **Null**: Represents the intentional absence of any value (null).
6. **Symbol**: A unique, immutable value used as object property keys (introduced in ES6, e.g., Symbol("id")).
7. **BigInt**: Represents large integers with arbitrary precision (e.g., 123n, introduced in ES2020).

**Non-Primitive Data Type**

1. **Object**: A collection of key-value pairs, used to store complex data. Includes:
2. **Arrays**: Ordered lists (e.g., [1, 2, 3]).
3. **Functions**: Reusable blocks of code (e.g., function () {}).
4. **Dates**, **RegExp**, and other built-in object types.
5. **What is the difference b/w var, let, and const?**

**var**: Function-scoped, hoisted with undefined, can be redeclared and reassigned.

var x = 1;

var x = 2; *// No error*

console.log(x); *// 2*

**let**: Block-scoped, hoisted but not initialized, reassignable, no redeclaration. (TDZ)

let y = 1;

y = 2; *// Works*

*// let y = 3; // Error: redeclaration*

**const**: Block-scoped, hoisted but not initialized, cannot be reassigned (object contents can still change).

const z = 1;

*// z = 2; // Error: reassignment*

const obj = { a: 1 };

obj.a = 2; *// Works: object mutation*

1. **What happens if you declare a variable without var, let, or const in strict mode?**

In JavaScript, when you’re in **strict mode** and declare a variable without var, let, or const, it throws a **ReferenceError** because implicit globals are disallowed. Strict mode, enabled by adding "use strict"; at the top of a script or function, enforces stricter rules to catch common mistakes and make code safer.

1. **What is the difference between undefined and null?**
2. **Undefined** Indicates a variable has been declared but not yet assigned a value, or a function returns nothing explicitly.event

let x;

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

function test() {}

console.log(test()); *// undefined*

let obj = {};

console.log(obj.key); *// undefined*

1. **Null** Represents the intentional absence of any object value—an explicit "no value" or "empty" state.

let y = null;

console.log(y); *// null*

let obj = { data: "value" };

obj.data = null; *// Explicitly reset*

console.log(obj.data); *// null*

1. **How does JavaScript handle type coercion? Provide an example.**

JavaScript handles **type coercion** by automatically converting one data type to another during operations when the types don’t match. This happens implicitly, often in comparisons, arithmetic, or string operations, based on built-in rules.

let num = 5;

let str = "10";

*// Addition: String concatenation (number coerced to string)*

console.log(num + str); *// "510"*

*// Subtraction: Number coercion (string coerced to number)*

console.log(num - str); *// -5*

*// Loose comparison: Both coerced to numbers*

console.log(num == str); *// false (5 ≠ 10)*

*// Boolean coercion in condition*

let empty = "";

if (empty) {

console.log("Truthy");

} else {

console.log("Falsy"); *// Prints "Falsy" (empty string is falsy)*

}

1. **What are the truthy and falsy values in JavaScript?**

**Falsy**: Exactly 6 values (false, 0, "", null, undefined, NaN).

**Truthy**: Everything else—most values in JavaScript are truthy by default.

**Gotchas**:

"0" is truthy (it’s a non-empty string), unlike 0.

[] and {} are truthy despite being empty, because they’re objects.

1. **How does the typeof operator work with arrays and functions?**
2. **typeof** Arrays is **“object”**

let arr = [1, 2, 3];

console.log(typeof arr); *// "object"*

1. **typeof** Functions is **“function”**

const funcExpr = function() {};

const arrowFunc = () => {};

console.log(typeof funcExpr); *// Output: "function"*

console.log(typeof arrowFunc); *// Output: "function"*

1. **What do you understand by hoisting in JavaScript?**

**Hoisting** in JavaScript is a behavior where variable and function declarations are moved—or "hoisted"—to the top of their containing scope during the compilation phase, before the code is executed. Means you can use variables and functions before they are explicitly declared in the code.

1. **Function Hoisting**

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

function sayHello() {

console.log("Hello!");

}

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

var sayHi = function() {

console.log("Hi!");

};

1. **Variable Hoisting**

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

var x = 5;

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

var x; // Hoisted, initialized as undefined

console.log(x);

x = 5;

console.log(x);

**let** and **const**

console.log(y); // Error: Cannot access 'y' before initialization // TDZ

let y = 10;

console.log(z); // Error: Cannot access 'z' before initialization // TDZ

const z = 20;

1. **What is this keyword in JavaScript?**

In JavaScript, the **this** keyword refers to the context in which a function is executed. Its value is determined dynamically at runtime, depending on how a function is called, rather than where it is defined. This makes this a powerful but sometimes confusing feature.

**1. Global Context**

When this is used outside of any function or in the global scope, it refers to the global object:

In browsers, this is the window object.

In Node.js, this is the global object.

**2. Function Context**

a) **Regular Function Call**

In a standalone function call (not attached to an object), this defaults to the global object (window in browsers) in non-strict mode. In strict mode ("use strict"), this is undefined.

*// Non Strict Mode*

function showThis() {

console.log(this);

}

showThis(); *// Output: Window (global object)*

*// Strict Mode*

"use strict";

function showThis() {

console.log(this);

}

showThis(); *// Output: undefined*

b) **Method Call**

When a function is called as a method of an object, this refers to the object that owns the method.

const obj = {

name: "Alice",

greet: function() {

console.log(this.name);

}

};

obj.greet(); *// Output: "Alice"*

**c) Explicit Binding**

You can explicitly set this using methods like call(), apply(), or bind():

call(thisArg, arg1, arg2, ...) and apply(thisArg, [args]) invoke the function immediately with a specified this.

bind(thisArg) returns a new function with this permanently set.

**3. Constructor Context**

When a function is used as a constructor with the new keyword, this refers to the newly created object. The new keyword creates a new object, sets this to that object, and returns it implicitly.

function Person(name) {

this.name = name;

}

const alice = new Person("Alice");

console.log(alice.name); *// Output: "Alice"*

**4. Arrow Functions**

Arrow functions (=>) behave differently: they do not have their own this. Instead, they inherit this from the surrounding (lexical) scope at the time they are defined.

const obj = {

name: "Charlie",

greet: function() {

const arrowFunc = () => console.log(this.name);

arrowFunc();

}

};

obj.greet(); *// Output: "Charlie"*

**5. Event Handlers**

In event listeners, this typically refers to the DOM element that triggered the event.

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

console.log(this); *// The button element*

});

**Key Rules to Determine this**

**Default**: Global object (or undefined in strict mode) for regular function calls.

**Object Method**: The object calling the method.

**Explicit**: Set by call, apply, or bind.

**New**: The newly created object in a constructor.

**Arrow Function**: Lexical this from the enclosing scope.

1. **What is temporal dead zone?**

The **Temporal Dead Zone (TDZ)** in JavaScript is a concept related to the behavior of variables declared with let and const. It refers to the time period between the start of a block scope and the point where the variable is declared, during which the variable cannot be accessed. Attempting to access it in this "dead zone" results in a ReferenceError.

**1. Basic TDZ Example**

console.log(x); *// ReferenceError: Cannot access 'x' before initialization*

let x = 5;

**2. Comparison with var**

console.log(y); *// Output: undefined*

var y = 10;

**3. TDZ with const**

console.log(z); *// ReferenceError: Cannot access 'z' before initialization*

const z = 20;

**4. TDZ in a Block Scope**

function example() {

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

let a = 100;

}

example();

**5. TDZ and Assignment**

let x = x + 1; *// ReferenceError: Cannot access 'x' before initialization*

1. **What is scope chaining?**

**Scope chaining** in JavaScript is the mechanism by which the JavaScript engine resolves variable lookups by searching through a chain of nested scopes, starting from the current scope and moving outward to outer (enclosing) scopes until it finds the variable or reaches the global scope.

When a variable is referenced, the engine checks the current scope, then the parent scope, and so on, until it finds the variable or hits the global scope (causing a ReferenceError if not found).

let a = "global";

function outer() {

let b = "outer";

function inner() {

console.log(a + b); *// Accesses "global" and "outer"*

}

inner();

}

outer(); *// Output: "globalouter"*

**Chain**: inner → outer → global.

1. **What is == and ===?**

In JavaScript, == and === are comparison operators used to check equality between two values.

== (Loose Equality) **-** Compares two values for equality after performing **type coercion** (converting one or both values to a common type).

=== (Strict Equality) **-** Compares two values for equality **without type coercion**. Both the value and the type must match.

1. **What is string interpolation?**

**String interpolation** in JavaScript is a way to embed expressions (like variables or calculations) directly into a string, making it easier to create dynamic strings without manual concatenation. It’s most commonly done using **template literals**, introduced in ES6 (ECMAScript 2015), which use backticks (`) and the ${expression} syntax.

let name = "Alice";

let greeting = `Hello, ${name}!`;

console.log(greeting); *// Output: "Hello, Alice!"*

**Group 2: Functions and Functional Programming (High Priority)**

1. **What are the different ways to create a function in JavaScript?**

**1. Function Declaration**

function greet(name) {

return `Hello, ${name}!`;

}

console.log(greet("Alice")); *// Output: "Hello, Alice!"*

**2.** **Function Expression**

const greet = function(name) {

return `Hello, ${name}!`;

};

console.log(greet("Bob")); *// Output: "Hello, Bob!"*

Named version (useful for recursion or debugging):

const greet = function sayHello(name) {

return `Hello, ${name}!`;

};

3. **Arrow Function**

const greet = (name) => `Hello, ${name}!`;

console.log(greet("Charlie")); *// Output: "Hello, Charlie!"*

Single parameter can skip parentheses:

const sayHi = name => `Hi, ${name}!`;

Multi-line requires curly braces and return:

const add = (a, b) => {

let sum = a + b;

return sum;

};

4. **Immediately Invoked Function Expression (IIFE)**

(function() {

console.log("I run once!");

})(); *// Output: "I run once!"*

5. **Constructor Function**

function Person(name) {

this.name = name;

}

const alice = new Person("Alice");

console.log(alice.name); *// Output: "Alice"*

6. **Method Definition (in Objects)**

const obj = {

greet(name) { *// Shorthand method syntax (ES6)*

return `Hello, ${name}!`;

}

};

console.log(obj.greet("Eve")); *// Output: "Hello, Eve!"*

7. **Generator Function**

function\* generator() {

yield 1;

yield 2;

yield 3;

}

const gen = generator();

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

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

1. **What is an arrow function?**

An **arrow function (=>)** is a concise way to write functions in JavaScript. It was introduced in **ES6 (ECMAScript 2015)** and has a **lexical this binding**, meaning it inherits this from the surrounding scope.

1. **No Own this Binding:**

Unlike traditional functions, arrow functions don’t have their own this. Instead, they inherit this from the surrounding (lexical) scope at the time they’re defined.

1. **No arguments Object:**

Arrow functions **do not have an arguments object**. Use **rest parameters (...args)** instead:

const sum = (...args) => args.reduce((a, b) => a + b);

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

const sum = () => {

console.log(arguments);

};

sum(1, 2, 3); // ❌ Uncaught ReferenceError: arguments is not defined

1. **Cannot Be Used as Constructors:**

You can’t use new with arrow functions because they lack a prototype property and their own this:

const MyFunc = () => {};

new MyFunc(); *// TypeError: MyFunc is not a constructor*

1. **Implicit Return:**

For one-liners, the result is returned automatically, parentheses () are **optional**:

const square = x => x \* x;

console.log(square(4)); *// 16*

1. **What is the difference b/w the named function and arrow function?**

**1️. Syntax:**

Named Function: Uses function keyword.

Arrow Function: Uses =>, shorter syntax.

**2️. this Binding:**

Named Function: this depends on how the function is called.

Arrow Function: this is lexically inherited from the surrounding scope.

**3️. arguments Object:**

Named Function: Has arguments object.

Arrow Function: ❌ No arguments, use ...args instead.

**4️. Usage in Objects & Classes:**

Named Function: Works well as object methods.

Arrow Function: ❌ Avoid using in object methods (inherits incorrect this).

**5️. Usage as Constructor:**

Named Function: Can be used with new.

Arrow Function: ❌ Cannot be used as a constructor.

**6. Implicit Return:**

Named Function: Requires return statement.

Arrow Function: Single expression auto-returns.

**7️. Best Used For:**

Named Function: Object methods, constructors, when this is needed.

Arrow Function: Callbacks (setTimeout, .map(), .filter()), event handlers in React.

1. **What is anonymous function?**

An **anonymous function** is a function **without a name**. It is typically used when a function is not required to be reused multiple times.

* **Assigned to Variables:**

const result = function(a, b) {

return a + b;

};

result(2, 3); *// 5*

* **Used as Arguments in Higher-Order Functions:**

setTimeout(function() {

console.log("Delayed!");

}, 1000);

* **Inside Array Methods (map, filter, reduce)**

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

const squared = numbers.map(function(num) {

return num \* num;

});

console.log(squared); // [1, 4, 9, 16]

* **IIFE (Immediately Invoked Function Expression): Runs once, right after definition.**

(function() {

console.log("Runs immediately!");

})();

1. **What are high order functions in JavaScript? Explain with example?**

A **higher-order function (HOF)** is a function that **takes another function as an argument** OR **returns a function**.

HOFs are powerful for abstraction, code reuse, and functional programming.

1. **Function as an Argument:**

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

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

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

1. **Function Returning a Function**

function multiplyBy (factor) {

return (number) => number \* factor; *// Returns a new function*

}

const double = multiplyBy (2); *// double is now a function*

const triple = multiplyBy (3); *// triple is another function*

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

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

multiplyBy is the HOF.

**Built-in Higher-Order Functions**

* **map()** Transforms an array by applying a function to each element.
* **filter()** Returns a new array with elements that pass a condition.
* **reduce()** Reduces an array to a single value.
* **forEach()** Executes a function for each element (does not return a new array).
* **sort()** Sorts elements based on a function.

1. **What is a pure function?**

**Always produces the same output** for the same input (deterministic).

**Has no side effects**, meaning it doesn’t modify external state or depend on anything outside its parameters.

function add (a, b) {

return a + b;

}

console.log (add (2, 3)); *// 5*

console.log (add (2, 3)); *// 5 (always 5)*

1. **What is recursion function?**

A **recursive function** is a function that calls itself to solve a problem by breaking it into smaller, similar subproblems. It continues calling itself with modified inputs until a **base case** is reached, which stops the recursion.

function factorial(n) {

*// Base case: if n is 0 or 1, return 1*

if (n <= 1) return 1;

*// Recursive case: n \* factorial of (n-1)*

return n \* factorial (n - 1);

}

console.log (factorial (5)); *// 5 \* 4 \* 3 \* 2 \* 1 = 120*

console.log (factorial (3)); *// 3 \* 2 \* 1 = 6*

function fibonacci(n) {

*// Base cases: n = 0 or 1*

if (n <= 0) return 0;

if (n === 1) return 1;

*// Recursive case: sum of previous two numbers*

return fibonacci (n - 1) + fibonacci (n - 2);

}

console.log (fibonacci (6)); *// 8 (sequence: 0, 1, 1, 2, 3, 5, 8)*

1. **What is function binding in JavaScript? What is bind()? What is the behaviour of the bind?**

* Function binding refers to the process of **fixing the this keyword** inside a function, ensuring that it refers to a specific object **regardless of how the function is called**.
* The **bind() method** creates a **new function** with a permanently bound this value.
* **Fixes this**: The new function always uses the this value provided to bind (), ignoring how it’s invoked.
* **Immutable Binding**: Once bound, the this value can’t be changed, even with call () or apply().
* **Partial Application**: Pre-set arguments are locked in, and additional arguments can be passed when calling the bound function.
* **Returns a New Function**: The original function is unaffected.

const obj = {

name: "Alice",

greet: function() {

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

}

};

const unboundGreet = obj.greet;

unboundGreet(); *// Hello, undefined (this is global/window)*

const boundGreet = obj.greet.bind(obj);

boundGreet(); *// Hello, Alice (this is locked to obj)*

1. **Difference b/w call, bind, apply?**

In JavaScript, call, bind, and apply are methods used to manipulate the this context and pass arguments to functions.

* **call () -** Invokes a function immediately with a specified this context and individual arguments.

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

Arguments are passed one by one (comma-separated).

const obj = { name: "Alice" };

function greet(greeting) {

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

}

greet.call(obj, "Hello"); *// Output: "Hello, Alice"*

* **apply () -** Invokes a function immediately with a specified this context and an array of arguments.

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

Arguments are passed as an array

const obj = { name: "Bob" };

function greet(greeting, punctuation) {

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

}

greet.apply(obj, ["Hi", "!"]); *// Output: "Hi, Bob!"*

* **bind () -** Creates a new function with a specified this context and optional preset arguments, but does **not** invoke it immediately.

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

Returns a new function that can be called later; useful for event handlers or delayed execution

const obj = { name: "Charlie" };

function greet(greeting) {

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

}

const boundGreet = greet.bind(obj, "Hey");

boundGreet(); *// Output: "Hey, Charlie"*

1. **What is constructor?**

In JavaScript, a **constructor** is a function or method used to create and initialize objects when called with the new keyword.

**Constructor Function:**

function Person(name) {

this.name = name;

}

const p = new Person("Alice"); *// Creates object with name "Alice"*

**Class Constructor (ES6):**

class Person {

constructor(name) {

this.name = name;

}

}

const p = new Person("Bob"); *// Same result*

1. **Difference b/w methods and functions?**
2. **Functions:** A standalone block of code designed to perform a task, defined independently of any object.

function greet(name) {

return `Hello, ${name}`;

}

console.log(greet("Alice")); *// Output: "Hello, Alice"*

1. **Methods:** A function that belongs to an object or class, acting as a property of that object.

const person = {

name: "Bob",

greet: function() {

return `Hello, ${this.name}`;

}

};

console.log(person.greet()); *// Output: "Hello, Bob"*

class Person {

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

greet() { return `Hello, ${this.name}`; }

}

const p = new Person("Charlie");

console.log(p.greet()); *// Output: "Hello, Charlie"*

1. **What is the difference between function declaration and function expression?**

* A **function declaration** defines a function with a **name** and can be called **before its definition** due to **hoisting**.

sayHi(); // Works fine

function sayHi() {

console.log("Hi");

}

* A **function expression** defines a function and assigns it to a variable. It **isn't hoisted**, meaning it cannot be called before its definition. Before defined gives ReferenceError.

const greet = function(name) {

return `Hello, ${name}`;

};

console.log(greet("Bob")); *// Output: "Hello, Bob"*

1. **How does the arguments object work in a function?**

The arguments object is an **array-like** object available inside **regular functions** (not arrow functions) that contains all the arguments passed to the function.

Has a length property and indexed elements (e.g., arguments[0]), but lacks array methods like map or forEach.

function example(a, b) {

console.log(arguments); *// Array-like object with all arguments*

console.log(arguments[0]); *// First argument*

console.log(arguments.length); *// Number of arguments passed*

}

example(1, 2, 3);

*// Output:*

*// [1, 2, 3] (array-like)*

*// 1*

*// 3*

1. **What happens if you use return inside an arrow function with curly braces vs. without?**

**Arrow Function with Curly Braces (Block Body)**

**Syntax**: (params) => { statements; return value; }

**Behavior**: You must explicitly use the return keyword to return a value. Without it, the function returns undefined by default (like regular functions).

const add = (a, b) => {

let sum = a + b;

return sum; // Explicit return required

};

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

**Arrow Function without Curly Braces (Concise Body)**

**Syntax**: (params) => expression

**Behavior**: The expression is implicitly returned—no return keyword needed. You can’t include multiple statements or a block; it’s limited to a single expression.

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

console.log(add(2, 3)); *// Output: 5*

1. **Write a higher-order function that takes a callback to double numbers in an array.**

function doubleArray(arr, callback) {

return arr.map(callback);

}

*// Callback to double a number*

const double = (num) => num \* 2;

*// Usage*

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

const doubled = doubleArray(numbers, double);

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

1. How does function hoisting differ from variable hoisting?

* **Function Hoisting**: Lifts the whole function, ready to use anywhere in its scope.
* **Variable Hoisting**: Lifts just the name; value availability depends on var vs. let/const.

**Group 3: Objects and Prototypes (Medium-High Priority)**

1. How to create an Object using a function?
2. Can const change in object?
3. How to add one more property to Object?
4. How to add new key to object?
5. How to get the length/depth of the Object?
6. What are closures in JavaScript?
7. What are Prototypes in JavaScript?
8. Explain JavaScript in OOPS with an example.
9. What is destructuring?
10. What is the difference between Object.create() and the new keyword?
11. How does prototypal inheritance work in JavaScript? Give an example.
12. What happens if you try to access a property that doesn’t exist on an object?
13. How can you prevent an object’s properties from being modified?
14. Write a closure that maintains a counter and increments it on each call.

Group 4: Asynchronous JavaScript (Medium-High Priority)

What is Asynchronous code?

What are callbacks in JavaScript?

What are the disadvantages of callback?

What are Promises in JavaScript and how does it work and its usage?

What is Pyramid of Doom in JavaScript? What is use of Promise in that?

What are the methods in Promise?

What is the difference between promise and callback?

What is async await?

What is the difference b/w Promise and Async await?

When do you use async/await or promise?

Why is the promise introduced as the same thing can be done with a callback?

Write a function for the promise method. async await and try catch promise completely?

What is event loop?

console b/w setTimeout and promise timing?

What is the microtask queue, and how does it differ from the macrotask queue?

How would you handle multiple promises running in parallel?

What happens if a promise rejects but there’s no .catch() handler?

Convert a callback-based function into a promise-based one.

How does async/await handle errors compared to promises?

Group 5: Arrays and Array Methods (Medium Priority)

What are array functions? Explain them?

What are filters and maps in JS?

What is basic difference is map and reduce?

What is reduce function in Array?

Filter the object with given city/name Pune.

Filter the array having the name with having skill will be JavaScript...

Make a copy of an array.

What is spread operators?

By cloning a spread array A to array B, by changing the value of A array the value B array is also changes, how to overcome this problem and why is behave like that.

Combine both array in a single array, after that sort the array in ascending order of ID.

What are inbuilt methods in Array?

How to flatten nested array output?

How would you remove duplicates from an array without using Set?

What’s the difference between forEach and map in terms of return value?

Write a custom reduce function to sum an array of numbers.

How can you sort an array of objects by a specific property?

What happens when you use slice vs. splice on an array?

Group 6: Advanced Topics and Optimization (Medium-Low Priority)

What is setTimeout and setInterval?

What is Map (allocate a memory) and WeakMap in JavaScript?

What is the use of mutation observer?

What is generator function?

What is the use of yield?

How yield works, ++ operator?

What is IIFI?

What are different optimization use in JavaScript?

How lazy loading works?

What is the best way to load the JavaScript files when there are multiple files and some import in the exist?

Best way to do import?

What is regular expression in JavaScript?

What is the difference between Map and Object in JavaScript?

How does garbage collection work with WeakMap?

What are the performance implications of using setInterval vs. recursive setTimeout?

How would you implement a generator to iterate over a range of numbers?

What are the benefits of lazy loading in a large-scale application?

Group 7: Practical Coding and Examples (Low-Medium Priority)

Write a program of reversing the string.

Prepare a snippet that count each character of string how many times it is present in it.

Find the index of object within the Array where the name of the book is four.

Filter the object which have student access is TRUE? Output will be array having student access is TRUE. without pre defined functions?

How to flatten nested object output?

Write a function to check if a string is a palindrome.

Create a function that debounces another function call.

How would you deep clone an object with nested properties?

Write a recursive function to calculate the factorial of a number.

Implement a simple throttle function for event handling.

Group 8: DOM and Events (Low-Medium Priority)

How to create an event in JavaScript? addEventListner use?

2 div with the same id, document.getElementById targets which div?

Add active class & style in JavaScript in react and JavaScript by select an list of element?

What’s the difference between event.preventDefault() and event.stopPropagation()?

How can you delegate an event to multiple child elements efficiently?

What happens if you call getElementById on a non-existent ID?

How would you dynamically add 10 list items to a <ul> using JavaScript?

What is event bubbling, and how can you control it?

Group 9: React and UI (Low Priority)

Create a one to do list from the text with having button to add and button to remove individually in React.

Initially in each data need to add one more key, once click this array item.

How would you handle state updates in React for a list of items?

What’s the difference between controlled and uncontrolled components in React?

How can you optimize re-rendering of a React component with a large list?

Write a React hook to fetch data from an API.

How do you conditionally apply a CSS class in React?

Group 10: Miscellaneous (Low Priority)

What is TypeScript? Any advantage?

Mention git command use in git commit/fetch.

What is the purpose of the inbuilt function?

0.1 + 0.2 == 0.3 console = Because of the limited precision of floating point numbers round-off errors can occur during calculations.

What is data type of null. => Object

Function name person and variable name with value abc and I have a function get name and typing this.name and out of this person function we are creating an object of person name and then calling the method what is the output?

Call by Property Tom and jerry JavaScript

What are the key differences between JavaScript and TypeScript in terms of error handling?

How does the call stack work in JavaScript execution?

What are the pros and cons of using regular expressions for input validation?

Why does NaN === NaN return false in JavaScript?

How would you handle module dependencies in a large JavaScript project?