

JavaScript Topics

1. Variables & Data Types : var, let, const.

2. Primitive types: string, number, boolean, null, undefined, symbol.

3. Null vs Undefined (imp):

a. null: is explicitly assigned to a variable to indicate that it is intentionally empty or has no value.

Type: It is of type object. (This is a historical quirk in JavaScript.)

Usage: It's typically used when you want to intentionally assign an empty value to a variable or object, indicating that something is intentionally missing.

b. undefined: means that a variable has been declared but has not been assigned a value, or a function doesn't return anything.

Type: It is its own type, undefined.

Usage: It's the default value for uninitialized variables or function arguments that are not passed. It also indicates that a function didn't return anything.

6. Reference types: object, array, function.

7. Control Structures: if, else, switch

8. Loops: for, while, do-while, for...of, for...in

9. Functions:

- a. Function declarations vs expressions
- b. Arrow functions
- c. Immediately Invoked Function Expression (IIFE)
- d. Higher-order functions (functions that take other functions as arguments or return them)
- e. Callback functions

10. Promises : It is a way to handle asynchronous operations.

- a. Promise.all()
- b. Promise.resolve()
- c. Promise.then()
- d. Promise.any()
- e. Promise.race()
- f. Promise.reject()

1. Async/await : Allows you to write asynchronous code in a more synchronous-looking manner.

2. Callback Function : A callback is a function that is passed as an argument to another function and is executed after the completion of that main function.

3. Closures : A closure in JavaScript is a function that has access to variables in its parent scope, even after the parent function has returned.

4. Scope :

a. Global vs local scope

b. Function scope, block scope (with let and const)

5. Hoisting :

a. Variable hoisting

b. Function hoisting

6. Event loop and task queue (microtasks and macrotasks)

7. Execution Context : An execution context is the environment in which the code is executed.

a. Global Execution Context (GEC)

b. Function Execution Context

8. Scope Chain & Execution Contexts :

The Scope chain is a crucial concept that determines how variables are looked up in different contexts when a function or block of code is executed

An Execution context is an abstract concept that represents the environment in which the JavaScript code is evaluated and executed. Every time a function is invoked or a block of code is run, a new execution context is created.

16. Memoisation: It is a technique used to optimize functions by caching the results of expensive function calls and reusing those results when the same inputs occur again. This helps avoid redundant computations, improving performance in scenarios where a function is called repeatedly with the same arguments.

17. Debouncing : Limits the rate at which a function gets invoked. Helps avoid multiple function calls for events that trigger frequently, such as keystrokes or resize events.

18. Throttling : Ensures that a function is called at most once in a given period of time, no matter how often the event is triggered.

19. Currying : Why: Currying transforms a function that takes multiple arguments into a series of functions that each take one argument. This is useful for partially applying arguments.

Where to use: Functional programming, reusing functions with fixed arguments.

20. `setTimeout()`, `setInterval()`, and `clearTimeout()` :

a. `setTimeout()` : Executes a function after a specified delay (in milliseconds).

b. `setInterval()` : Executes a function repeatedly at a specified interval (in milliseconds).

c. `clearTimeout()`: Cancels a previously scheduled `setTimeout()` operation.

21. Template Literals: Template literals, also known as template strings, are a feature in JavaScript that allow for easier string interpolation and multi-line strings. They are denoted by backticks (```) instead of single or double quotes.

22. LocalStorage & SessionStorage:

localStorage: Known for storing data persistently across browser sessions, remaining available even after the browser is closed.

sessionStorage: Known for storing data only for the duration of a single browser session, clearing when the tab or browser is closed.

23. Regular Expressions (RegExp): A Regular Expression (RegExp or RegEx) is a sequence of characters that defines a search pattern. RegEx is primarily used for string searching and manipulation, allowing you to search, match, and replace patterns in text.

24. this Keyword : this keyword refers to the context in which a function is executed. It is a special keyword that behaves differently depending on how a function is called.

25. In the global execution context (outside any function), this refers to the global object (window in browsers, global in Node.js).

26. In a regular function (not in strict mode), this refers to the global object (window in the browser).

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

28. OOPs in JavaScript:

Classes In JavaScript

Classes and Objects in JavaScript

How to create a JavaScript class in ES6

this Keyword JavaScript

New Keyword in JavaScript

Object Constructor in JavaScript

Inheritance in JavaScript

Encapsulation in JavaScript

Static Methods In JavaScript

OOP in JavaScript

Getter and Setter in JavaScript

26. Operators :

- a. Arithmetic Operators: +, -, _, /, %
- b. Comparison Operators: ==, ===, !=, !==, >, <, >=, <=
- c. Logical Operators: &&, ||, !
- d. Assignment Operators: =, +=, -=, _=, /=
- e. Unary Operators: ++, --, typeof, delete
- f. Ternary Operator (imp) : condition ? expr1 : expr2

27. Break and Continue

break (exit the loop)

continue (skip to the next iteration) 28. Parameters and Arguments :

Parameters are the variables defined in the function declaration (or function signature) that specify what kind of values the function expects to receive when it is called.

Arguments are the actual values passed to the function when it is called.

29. Destructuring : Destructuring is a syntax in JavaScript that allows you to unpack values from arrays or properties from objects into distinct variables. It simplifies the process of extracting multiple properties or elements from an object or array, making your code cleaner and more readable.

a. Array destructuring

b. Object Destructuring

30. Rest and Spread Operator : The Rest and Spread operators are both represented by ... in JavaScript, but they serve different purposes depending on how they are used.

The Rest operator is used to collect multiple elements and bundle them into a single array or object. It's mainly used in function parameters to collect arguments, or in destructuring to collect remaining properties.

The Spread operator is used to unpack elements of an array or object into individual elements or properties. It allows you to expand or “spread” an iterable (array or object) into individual elements or properties.

31.Event Delegation : Using event listeners on parent elements to handle child element events

32. Higher-Order Functions : A Higher-Order Function is a function that either takes one or more functions as arguments or returns a function as its result.

33. Anonymous functions : An anonymous function is a function that does not have a name. These functions are typically defined inline and can be assigned to variables, passed as arguments, or used in other places where a function is required.

Key Characteristics:

No Name: The function is defined without a name.

Often Used Inline: Commonly used as callback functions or passed as arguments to higher-order functions.

Can be Assigned to Variables: Can be assigned to variables or properties just like any other value. 35. Lexical scoping: The process of determining the scopes of the variables or functions during runtime is called lexical scoping.

How Lexical Scoping Works:

When you define a function, it has access to variables that are within its scope (i.e., variables declared inside the function and variables from outer functions, including the global scope).

If a function is nested inside another, the inner function can access variables from the outer function (this is called closures).

36. Array Methods

push(), pop(), shift(), unshift()

concat(), slice(), splice()

map(), filter(), reduce(), forEach()

find(), findIndex()

sort(), reverse()

join(), split()

indexOf(), includes(), lastIndexOf()

37. Object Methods:

Object.assign(), Object.create(), Object.keys(), Object.values(),
Object.entries(), Object.hasOwnProperty(), Object.freeze(), Object.seal()

38. Prototypes :

Prototype chain

Inheritance using prototypes

39. Classes

Class syntax, constructors, methods

Inheritance using extends

super() and super() constructor

40. call(), apply(), and bind() : for controlling the context of this

41. Event bubbling and capturing:

Event Bubbling occurs when an event is triggered on an element, and the event then “bubbles up” from the target element to its ancestor elements in the DOM tree. In most cases, events bubble up by default unless you specifically prevent it

Event Capturing is the opposite of event bubbling. The event is first captured by the root element and then trickles down to the target element.

42. Generators & Iterators:

Why: Generators allow for lazy evaluation, meaning they yield values on demand rather than all at once. Useful for large data sets or infinite sequences.

Where to use: Implementing custom iterators, lazy evaluation of sequences.

43. WeakMap and WeakSet:

Why: Helps with memory management in JavaScript. WeakMap and WeakSet allow garbage collection of keys or values when there are no more references to them.

Where to use: Managing references to objects without preventing garbage collection. For example, caching DOM nodes where you don't want to create memory leaks.

44. Polyfill :

Why: Adds support for features that are not natively available in older browsers by providing code that mimics modern functionality.

Where to use: Ensuring compatibility with older browsers (e.g., older versions of Internet Explorer) for new JavaScript features like Promise, fetch, etc.

45. Prototypal Inheritance:

Why: JavaScript uses prototypes for inheritance, rather than the classical object-oriented inheritance. Understanding how the prototype chain works is key to understanding JavaScript's inheritance model.

Where to use: Building object hierarchies, adding methods to constructors.

46. Cookies: Storing and retrieving cookies in JavaScript

47. Advanced Array Methods

`Array.prototype.find()` : Finding the first element in an array that matches a condition

`Array.prototype.filter()`: Filtering elements based on a condition

`Array.prototype.reduce()`: Reducing an array into a single value

`Array.prototype.map()`: Creating a new array by applying a function to each element

`Array.prototype.sort()`: Sorting arrays with custom sorting functions 48. Design Patterns :

Module Pattern: Encapsulating code into modules

Singleton Pattern: Ensuring a class has only one instance

Observer Pattern: Notifying multiple objects when one object's state changes.

Factory Pattern: Provides a way to instantiate objects while keeping the creation logic separate from the rest of the application.

Strategy Pattern: Allows you to define a strategy (algorithm) for a particular operation and change it at runtime.

Decorator Pattern: Dynamically adding behavior to an object without affecting its structure. 49. Lazy Loading: Delaying loading of content until it's needed.

50. Working with JSON

JSON Basics

JSON syntax, parsing with `JSON.parse()`, stringifying with `JSON.stringify()`

Working with APIs

Fetching data from an API using `fetch()`

Handling API responses with Promises or `Async/Await` 51. DOM Manipulation

DOM Selection

`document.getElementById()`, `document.querySelector()`,
`document.querySelectorAll()`

Event Handling

Event listeners: `addEventListener()`, `removeEventListener()`

`event.target`, `event.preventDefault()`, `event.stopPropagation()`

Modifying DOM Elements

Changing text, HTML, attributes, styles

Adding/removing elements dynamically (createElement(), appendChild(), removeChild())

DOM Traversal

parentNode, childNodes, nextSibling, previousSibling 52. Error Handling :

try...catch...finally: Handling errors in synchronous code

Custom Errors: Creating custom error classes

Throwing Errors: throw keyword for throwing errors manually 53. String Methods :

charAt(), charCodeAt(), concat(), includes(), indexOf(), lastIndexOf(), slice(), split(), toLowerCase(), toUpperCase(), trim(), replace(), search(), match(), repeat(), startsWith(), endsWith(), padStart(), padEnd(), localeCompare(), fromCharCode().

54. Date methods:

Date.now(), Date.parse(), Date.UTC(), getDate(), getDay(), getFullYear(), getHours(), getMilliseconds(), getMinutes(), getMonth(), getSeconds(), getTime(), getTimezoneOffset(), setDate(), setFullYear(), setHours(), setMilliseconds(), setMinutes(), setMonth(), setSeconds(), setTime(), toDateString(), toISOString(), toLocaleDateString(), toLocaleTimeString(), toString().

55. Generator: A generator in JavaScript is a special type of function that allows you to pause and resume its execution.

function*, yield, next(), return(), throw().

56. JavaScript Proxy : A Proxy in JavaScript is a special object that allows you to intercept and customize operations on objects, such as property access, assignment, function calls, and more. It acts as a wrapper for another object and can redefine fundamental operations (like get, set, deleteProperty, etc.) on that object.

Commonly Used Traps (Methods):

get(target, prop, receiver): Intercepts property access.

set(target, prop, value, receiver): Intercepts property assignment.

has(target, prop): Intercepts the in operator.

deleteProperty(target, prop): Intercepts property deletion.

apply(target, thisArg, argumentsList): Intercepts function calls.

construct(target, args): Intercepts the new operator.

defineProperty(target, prop, descriptor): Intercepts property definition. 57.

Javascript Array and Object Cloning : Shallow or Deep?

A shallow clone of an object or array creates a new instance, but it only copies the top-level properties or elements. If the original object or array contains references to other objects (nested objects or arrays), these inner objects are not copied. Instead, the shallow clone will reference the same objects.

A deep clone creates a completely independent copy of the original object or array. It recursively copies all the properties or elements, including nested objects or arrays. This means that deep cloning ensures that no references to nested objects are shared between the original and the clone.

58. loose equality (==) and strict equality (===) :

Loose equality compares two values for equality after performing type coercion. This means that the values are converted to a common type (if they are of different types) before making the comparison.

When using ==, JavaScript attempts to convert the operands to the same type before comparing them.

Strict equality compares two values without performing any type conversion. It checks both the value and the type of the operands.

For ===, the operands must be of the same type and value to be considered equal.

59. Call by Value Vs Call by Reference :

Call by Value: When an argument is passed to a function by value, a copy of the actual value is passed. Any changes made to the argument inside the function do not affect the original variable outside the function.

When it happens: This occurs when primitive types (like numbers, strings, booleans, null, undefined, and symbols) are passed to a function.

Call by Reference: When an argument is passed by reference, the reference (or memory address) of the actual object is passed to the function. This means any changes made to the argument inside the function will directly modify the original object outside the function.

When it happens: This occurs when non-primitive types (like objects, arrays, and functions) are passed to a function.

60. JavaScript Set: A Set in JavaScript is a built-in collection object that allows you to store unique values of any type, whether primitive or object references.

Key Characteristics of a Set:

Unique Elements: A Set automatically ensures that each value it contains is unique. If you try to add a duplicate value, it will be ignored.

Ordered: The elements in a Set are ordered, meaning the values are stored in the order they were added. However, Sets do not allow duplicate entries.

Iterable: Sets are iterable, so you can loop over the elements in a Set using `for...of`, or methods like `.forEach()`.

No Indexes: Unlike Arrays, Set elements are not accessed by an index. They are stored by insertion order, but you can't reference them by a number.

Basic Methods of a Set :

`add(value)`: Adds a value to the Set. If the value already exists, it does nothing (no duplicates).

`has(value)`: Checks if the Set contains the specified value. Returns `true` or `false`.

`delete(value)`: Removes the specified value from the Set.

`clear()`: Removes all elements from the Set.

`size`: Returns the number of elements in the Set.

`forEach(callback)`: Executes a provided function once for each value in the Set.

61. JavaScript Map : A Map in JavaScript is a built-in object that stores key-value pairs. Unlike regular JavaScript objects, Maps allow keys of any data type (including objects, functions, and primitive types like strings and numbers) to be used.

Maps also maintain the insertion order of their keys, making them useful in scenarios where order matters.

Basic Methods of a Map:

`set(key, value)`: Adds or updates an element with the specified key and value in the Map.

`get(key)`: Retrieves the value associated with the specified key.

`has(key)`: Checks if a Map contains a key.

`delete(key)`: Removes the element associated with the specified key.

`clear()`: Removes all elements from the Map.

`size`: Returns the number of key-value pairs in the Map.

`forEach(callback)`: Executes a provided function once for each key-value pair in the Map.

`keys()`: Returns an iterator object containing all the keys in the Map.

`values()`: Returns an iterator object containing all the values in the Map.

`entries()`: Returns an iterator object containing an array of [key, value] pairs.

62. The Fetch API: The Fetch API allows us to make async requests to web servers from the browser. It returns a promise every time a request is made which is then further used to retrieve the response of the request.

63. Import/Export:

Modules: In JavaScript, a module is a file that contains code you want to reuse. Instead of having everything in one file, you can split your code into separate files and then import what you need. This keeps the code clean, organized, and maintainable.

Imports: This is how you bring in functionality from other modules into your current file.

Exports: This is how you make variables, functions, classes, or objects from one module available for use in other modules.

64. Pure Functions, Side Effects, State Mutation and Event Propagation:

65. Recursion:

Recursion is a fundamental programming concept where a function calls itself in order to solve a problem. Recursion is often used when a problem can be broken down into smaller, similar sub-problems. In JavaScript, recursion is useful for tasks like traversing trees, solving puzzles, and more.

Key Concepts:

Base Case: The condition that stops the recursion. Without a base case, recursion can lead to infinite function calls, causing a stack overflow error.

Recursive Case: The part of the recursion where the function calls itself with a smaller or simpler version of the problem.

66. The apply, call, and bind methods:

67. window methods:

alert(), confirm(), prompt(), setTimeout(), setInterval(), clearTimeout(), clearInterval(), open(), close(), requestAnimationFrame().

68. Mouse events:

click, dblclick, mousedown, mouseup, mousemove, mouseover, mouseout, mouseenter, mouseleave, contextmenu.

69. Keyboard events:

keydown, keypress, keyup.

70. Form events:

submit, change, focus, blur, input, reset, select, keypress, keydown, keyup.

71. Debugging:

72. Cross-Origin Resource Sharing (CORS):

73. Web Workers: A mechanism for running scripts in background threads, allowing JavaScript to perform computationally expensive tasks without blocking the main thread.

74. Service Workers: A script that runs in the background of your browser, enabling features like push notifications, background sync, and caching for offline functionality.

75. Lazy loading or Infinite scrolling):

Lazy Loading and Infinite Scrolling are two techniques commonly used to enhance performance and user experience in web applications, especially when dealing with large amounts of data or media (like images, lists, or articles).

Lazy Loading is a design pattern in web development where resources (such as images, scripts, videos, or even content) are loaded only when they are required.

The main goal of lazy loading is to improve the initial loading time of a webpage by reducing the number of resources loaded initially.

Infinite Scrolling is a technique that automatically loads more content as the user scrolls down the page, typically without the need for pagination. This is widely used in social media platforms, news sites, and any web application that needs to display large datasets (e.g., Instagram, Twitter, Facebook).

76: Progressive Web Apps (PWAs): Building web applications that work offline, provide push notifications, and have native-like performance (through service workers and other browser APIs).

77. Server-sent events: Server-sent events (SSE) are a simple and efficient technology for enabling real-time updates from the server to the client over a single HTTP connection.

78. Strict Mode: Strict Mode is a feature in JavaScript that ensures that you avoid errors and problematic features.

79. Security: (Not a JavaScript concept, but important to know)

Cross-Site Scripting (XSS)

Cross-Site Request Forgery (CSRF)

Content Security Policy (CSP)

CORS (Cross-Origin Resource Sharing)

JWT (JSON Web Tokens) 80. Temporal Dead Zone (TDZ) : It is a term that refers to the period of time between the creation of a variable and its initialization in the execution context. During this time, the variable exists but cannot be accessed — attempting to do so will result in a `ReferenceError`.

The TDZ occurs for variables declared using `let` and `const` but not for `var` because `var` declarations are hoisted and initialized with `undefined`.

Thats it guys.

Thank you for taking the time to read my blog! I hope you found it helpful and informative. If I missed any important topics or concepts, I sincerely apologize. Feel free to leave a comment or reach out if you have any questions or suggestions. Your feedback is always appreciated!