1. ⭐ **<u>Variables & Data Types :</u>** var, let, const.
2. ⭐ **Primitive types:** string, number, boolean, null, undefined, symbol.
3. ⭐ **<u>Null vs Undefined (imp):</u>**
4. ⭐. `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.
5. ⭐. `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. ⭐ **Object Property Shorthand :** Property shorthand in JavaScript allows you to create object properties without explicitly specifying the property name if the variable name and the property name are the same.

10. ⭐ **Nullish Coalescing Operator :** The **nullish coalescing operator** (`??`) in JavaScript is used to return the right-hand side operand when the left-hand operand is either `null` or `undefined` . This is useful when you want to provide a default value for `null` or `undefined` without overriding other falsy values like `0` , `false` , or `""` (empty string).

11. ⭐ **"use strict" :** "Use strict" is used to indicate that the code should be executed in strict mode, applying a stricter set of rules for JavaScript coding.

12. ⭐ **Optional Chaining :** The Optional Chaining Operator allows developers to access properties of an object without worrying about TypeError, making the code more concise and less prone to bugs.

13. ⭐ **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

14. ⭐ **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()
    g.Promise.allSettled()

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

16. ⭐ **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.

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

18. ⭐ **Scope :**
    a. Global vs local scope
    b. Function scope, block scope (with let and const)

19. ⭐ **Hoisting :**
    a. Variable hoisting
    b. Function hoisting

20. ⭐ **Event loop and task queue : (microtasks and macrotasks)**

21. ⭐ **Execution Context** : An **execution context** is the environment in which the code is executed.
    a. Global Execution Context (GEC)
    b. Function Execution Context

22. ⭐ **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.

23. ⭐ **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.

24. ⭐ **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.

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

26. ⭐ **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.

27. ⭐ **setTimeout(), setInterval(), clearTimeout() and clearInterval()** :
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.

28. ⭐ **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.

29. ⭐ **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.

30. ⭐ **Regular Expressions (RegExp):** A **Regular Expression** (RegEx or RegExp) 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.

31. ⭐ **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.
1. In the global execution context (outside any function), `this` refers to the **global object** (`window` in browsers, `global` in Node.js).
2. In a regular function (not in strict mode), `this` refers to the **global object** (`window` in the browser).
3. When a function is called as a method of an object, `this` refers to the **object** the method is called on.

32. ⭐ **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

33. ⭐ **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`

34. ⭐ **Break and Continue**
`break` (exit the loop)
`continue` (skip to the next iteration)

35. ⭐ **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.

36. ⭐ **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**

37. ⭐ **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.

38. ⭐ **Event Delegation :** Using event listeners on parent elements to handle child element events

39. ⭐ **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.

40. ⭐ **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.

41. ⭐ **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**).

42. ⭐ **Array Methods :**

- `push()` , `pop()` , `shift()` , `unshift()`
- `concat()` , `slice()` , `splice()`
- `map()` , `filter()` , `reduce()` , `forEach()`
- `find()` , `findIndex()`
- `sort()` , `reverse()`
- `join()` , `split()`
- `indexOf()` , `includes()` , `lastIndexOf()`

43. ⭐ **Object Methods :**

- `Object.assign()` , `Object.create()` , `Object.keys()` , `Object.values()` , `Object.entries()` , `Object.hasOwn()` , `Object.freeze()` , `Object.seal()`

44. ⭐ **Prototypes :**

- Prototype chain :
- Inheritance using prototypes

45. ⭐ **Classes :** A JavaScript class is a blueprint used to create objects in Javascript.

- **Constructor() :** The constructor() is a method used to initialize the object properties. It is executed automatically when a new object of the given class is created.
- **Class Methods :** Instance methods, Static methods, Getter and Setter methods, Private Methods
- **Class Fields :** Instance Fields (Public Fields) , Static Fields (Static Properties), Private Fields
- **Inheritance** : Inheritance using **`extends`** , Method Overriding
- `super()` **and** `super()` **constructor**

46. ⭐ **call(), apply(), and bind() :** for controlling the context of this

47. ⭐ 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.

48. ⭐ 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.

49. ⭐ 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.

50. ⭐ 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.

51. ⭐ 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.

52. ⭐ Cookies : Storing and retrieving cookies in JavaScript

53. ⭐ **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

54. ⭐ **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.

55. ⭐ **Lazy Loading :** Delaying loading of content until it's needed.

56. ⭐ **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

57. ⭐ **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`

58. ⭐ **Error Handling** :

- **`try...catch...finally`:** Handling errors in synchronous code
- **Custom Errors:** Creating custom error classes
- **Throwing Errors:** `throw` keyword for throwing errors manually

59. ⭐ **String Methods** :

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

60. ⭐ **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()`.

61. ⭐ **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()` .

62. ⭐ 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.

How JavaScript Proxy Works Under The Hood ?

Commonly Used Traps (Methods):

1. **`get(target, prop, receiver)`** : Intercepts property access.
2. **`set(target, prop, value, receiver)`** : Intercepts property assignment.
3. **`has(target, prop)`** : Intercepts the `in` operator.
4. **`deleteProperty(target, prop)`** : Intercepts property deletion.
5. **`apply(target, thisArg, argumentsList)`** : Intercepts function calls.
6. **`construct(target, args)`** : Intercepts the `new` operator.
7. **`defineProperty(target, prop, descriptor)`** : Intercepts property definition.

63. ⭐ 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.

64. ⭐ 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.

65. ⭐ **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.

66. ⭐ **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:

1. **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.

2. **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.

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

4. **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 :

1. **`add(value)`** : Adds a value to the Set. If the value already exists, it does nothing (no duplicates).
2. **`has(value)`** : Checks if the Set contains the specified value. Returns `true` or `false` .
3. **`delete(value)`** : Removes the specified value from the Set.
4. **`clear()`** : Removes all elements from the Set.
5. **`size`** : Returns the number of elements in the Set.
6. **`forEach(callback)`** : Executes a provided function once for each value in the Set.

67. ⭐ **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 :

1. **`set(key, value)`** : Adds or updates an element with the specified key and value in the Map.
2. **`get(key)`** : Retrieves the value associated with the specified key.
3. **`has(key)`** : Checks if a Map contains a key.
4. **`delete(key)`** : Removes the element associated with the specified key.
5. **`clear()`** : Removes all elements from the Map.
6. **`size`** : Returns the number of key-value pairs in the Map.
7. **`forEach(callback)`** : Executes a provided function once for each key-value pair in the Map.
8. **`keys()`** : Returns an iterator object containing all the keys in the Map.
9. **`values()`** : Returns an iterator object containing all the values in the Map.

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

68. ⭐ **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.

⭐ **Axios vs. Fetch API (IMP Interview POV):** Article 1, Article 2, Article 3

69. ⭐ **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.

70. ⭐ **Pure Functions, Side Effects , State Mutation and Event Propagation:**

71. ⭐ **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:

1. **Base Case:** The condition that stops the recursion. Without a base case, recursion can lead to infinite function calls, causing a stack overflow error.
2. **Recursive Case:** The part of the recursion where the function calls itself with a smaller or simpler version of the problem.

72. ⭐ **The apply, call, and bind methods :**

73. ⭐ **Window methods:**
Window methods are part of the **Window interface** in JavaScript, which represents

the browser window or frame containing a web page.

These methods allow developers to interact with and manipulate the browser environment, including displaying messages, handling timers, opening or closing browser windows, and optimizing animations.

`alert()`: Displays a message in a modal dialog box with an **OK** button.Blocks further code execution until the user closes it.

`confirm()`: Displays a modal dialog box with a message and OK/Cancel buttons.Returns true if the user clicks OK, and false if Cancel.

`prompt()`: Displays a dialog box that prompts the user to enter input.Returns the input as a string or null if canceled.

`setTimeout()`: Executes a function after a specified delay (in milliseconds).
`setInterval()`: Repeatedly executes a function at a specified time interval (in milliseconds).

`clearTimeout()`: Stops the execution of a function scheduled with setTimeout().

`clearInterval()`: Stops a function scheduled with `setInterval()`.
`open()`: Opens a new browser window or tab.
`close()`: Closes a window opened using `window.open()`.

74. ⭐ **Mouse events:**
Mouse events are part of the DOM (Document Object Model) in JavaScript and are triggered when a user interacts with a web page using a mouse or similar pointing device.

These events allow developers to detect and respond to mouse actions, such as clicks, movement, and hover effects, to create dynamic and interactive user interfaces.

`click`:
Triggered when the user presses and releases the left mouse button on an element.
Example: Clicking a button to submit a form.

`dblclick`:
Triggered when the user double-clicks on an element.

Example: Opening a file or folder on double-click.

`mousedown` :
Triggered when the user presses any mouse button over an element.
Example: Detecting when a user starts dragging an item.

`mouseup` :
Triggered when the user releases a mouse button that was pressed on an element.
Example: Detecting when a drag-and-drop action ends.

`mousemove` :
Triggered repeatedly when the user moves the mouse over an element.
Example: Updating the position of a custom cursor or tooltip.

`mouseover` :
Triggered when the user moves the mouse pointer over an element or its child elements.
Example: Highlighting a menu item when hovered.

`mouseout` :
Triggered when the user moves the mouse pointer out of an element or its child elements.
Example: Removing a hover effect when the pointer leaves an element.

`mouseenter` :
Triggered when the mouse pointer enters an element (does not bubble).
Example: Displaying a tooltip when the pointer enters a specific area.

`mouseleave` :
Triggered when the mouse pointer leaves an element (does not bubble).
Example: Hiding a tooltip when the pointer exits a specific area.

`contextmenu` :
Triggered when the user right-clicks on an element, opening the context menu.
Example: Showing a custom context menu for an image or text.

75. ⭐ **Keyboard events:**
Keyboard events are part of the DOM in JavaScript and are triggered when a user interacts with a keyboard.

These events allow developers to detect and respond to key presses, enabling features such as form validation, hotkeys, and interactive user experiences.

`keydown` , `keypress` , `keyup` .

76. ⭐ **Form events:**
`submit` , `change` , `focus` , `blur` , `input` , `reset` , `select` , `keypress` , `keydown` , `keyup` .

77. ⭐ **Debugging :**

78. ⭐ **Cross-Origin Resource Sharing (CORS):**

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

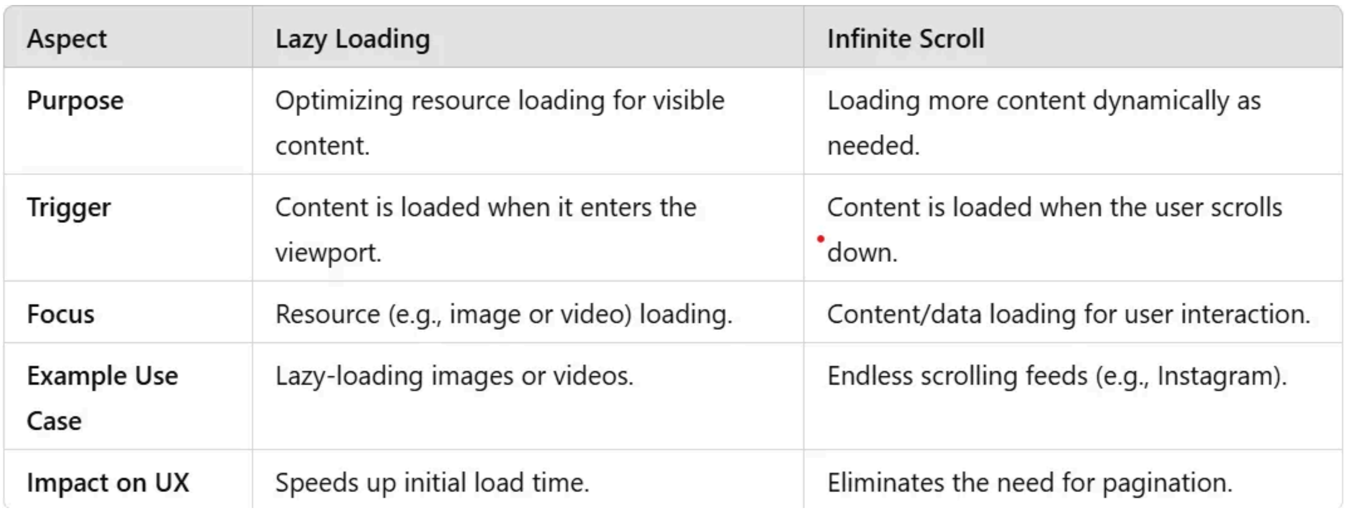
80. ⭐ **Service Workers**: A script that runs in the background of your browser, enabling features like push notifications, background sync, and caching for offline functionality. (**Article 2**)

81. ⭐ **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).



Aspect	Lazy Loading	Infinite Scroll
Purpose	Optimizing resource loading for visible content.	Loading more content dynamically as needed.
Trigger	Content is loaded when it enters the viewport.	Content is loaded when the user scrolls down.
Focus	Resource (e.g., image or video) loading.	Content/data loading for user interaction.
Example Use Case	Lazy-loading images or videos.	Endless scrolling feeds (e.g., Instagram).
Impact on UX	Speeds up initial load time.	Eliminates the need for pagination.

82: ⭐ **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).

83. ⭐ **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.**

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

85. ⭐ **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)**

86. ⭐ **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` .

87. ⭐ **CSR vs. SSR vs. SSG : (Important)**

Web applications can be rendered in different ways depending on their architecture and the desired user experience. The three primary rendering techniques are **Client-Side Rendering (CSR)**, **Server-Side Rendering (SSR)**, and **Static Site Generation (SSG)**.

CSR vs. SSR vs. SSG —For a detailed information, refer to this as well.

ℹ️ If you found this article helpful, you may also enjoy the following related articles : ℹ️

⛔ **Complete Guide to React.js: Basic to Advanced Questions and Concepts — Part 1**

⛔ **Complete Guide to React.js: Basic to Advanced Questions and Concepts — Part 2**

Explore these articles to deepen your understanding of ReactJS, or discover new insights on related topics!

Thats it guys. 😜 ✌️

Learning JavaScript 💻 , or any programming language, can feel overwhelming at first. If you don't understand a topic on the first go, that's completely okay! No one becomes an expert overnight. The key is consistency, practice, and persistence.

Remember, every developer started from the basics. Take your time to understand each topic thoroughly, especially if you're a beginner. Don't rush the process, and don't get discouraged by the number of concepts to master. Keep practicing, and gradually, things will start to click.

Stay patient, stay curious, and most importantly, keep coding !

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!

And don't forget to clap if you found this helpful! 👏

JavaScript Web3 Interview Coding Development