JAVASCRIPT INTERVIEW QUESTIONS - COMPLETE GUIDE

Table of Contents

JAVASCRIPT BASICS ANSWERS

1. What is JavaScript?

JavaScript is a **lightweight, interpreted, high-level** programming language primarily used for making web pages **interactive and dynamic**. It runs in browsers (client-side) and servers (**Node.js**), supports **object-oriented**, **functional**, and **event-driven** programming paradigms. It's single-threaded with **asynchronous** capabilities through event loop, created by **Brendan Eich** in 1995.

2. Is JavaScript interpreted or compiled?

Traditionally considered **interpreted** language (line-by-line execution), but modern JavaScript engines use **Just-In-Time (JIT) compilation**. Code is parsed → compiled to bytecode → optimized during runtime. **V8 engine** (Chrome/Node.js) compiles directly to machine code. Technically now a **"JIT-compiled" or "hybrid"** language.

2.1 What is JIT compilation?

JIT (Just-In-Time) compilation in JavaScript is a runtime optimization technique where the JavaScript engine compiles frequently executed code into optimized machine code during execution, improving performance while still supporting JavaScript's dynamic nature.

3. Difference between Java and JavaScript

- **Origin**: Java by Sun Microsystems, JavaScript by Netscape
- **Typing**: Java is statically typed, JavaScript is dynamically typed
- **Execution**: Java compiles to bytecode for JVM, JavaScript runs in browser/Node.js
- **Paradigm**: Java is class-based OOP, JavaScript is prototype-based
- **Syntax**: Java requires explicit types, JavaScript uses inference
- **Use Case**: Java for enterprise/backend, JavaScript for web/full-stack

4. What are variables in JavaScript?

Variables are **containers for storing data values**, declared using **`var`**, **`let`**, or **`const`** keywords. JavaScript variables are **dynamically typed** (type determined at runtime). Variable names are **case-sensitive** and follow camelCase convention, must start with letter, underscore (_), or dollar sign ($), cannot use reserved keywords.

5. Difference between `var`, `let`, and `const`

- **Scope**: `var` is function-scoped, `let` and `const` are block-scoped
- **Hoisting**: `var` hoisted with undefined, `let`/`const` hoisted but in TDZ
- **Re-declaration**: `var` allows re-declaration, `let`/`const` do not
- **Re-assignment**: `var` and `let` allow re-assignment, `const` does not
- **Global Object**: `var` attaches to window object, `let`/`const` do not
- **Best Practice**: Use `const` by default, `let` when re-assignment needed

6. What are data types in JavaScript?

- **Primitive Types** (7): `string`, `number`, `boolean`, `undefined`, `null`, `symbol`, `bigint`
- **Non-Primitive Type**: `object` (includes arrays, functions, objects)
- JavaScript is **dynamically typed** (type can change at runtime)
- **typeof** operator used to check data type
- **Primitives** are immutable, **objects** are mutable
- ES6 added `Symbol`, ES2020 added `BigInt`

7. Primitive vs non-primitive data types

- **Primitive**: Stored directly in **stack**, accessed by value

- **Non-Primitive**: Stored in **heap**, accessed by reference
- **Primitive**: Immutable, changes create new values
- **Non-Primitive**: Mutable, can be modified in place
- **Comparison**: Primitives compared by value, objects by reference
- **Memory**: Primitives have fixed size, objects have dynamic size

8. What is `typeof`?

typeof is a **unary operator** that returns the data type as a string. Returns: `"string"`, `"number"`, `"boolean"`, `"undefined"`, `"object"`, `"function"`, `"symbol"`, `"bigint"`. **typeof null** returns `"object"` (historical bug), **typeof array** returns `"object"` (use `Array.isArray()` instead), **typeof function** returns `"function"`.

9. What is `NaN`?

NaN stands for **"Not-a-Number"**, represents result of invalid or undefined mathematical operation. Type of NaN is **"number"** (typeof NaN === "number"). **NaN !== NaN** (NaN is not equal to itself!). Use **isNaN()** or **Number.isNaN()** to check for NaN. **Number.isNaN()** is more reliable (doesn't coerce types).

10. What is `undefined` vs `null`?

- **undefined**: Variable declared but not assigned a value
- **null**: Intentional absence of any object value
- **typeof undefined** returns `"undefined"`
- **typeof null** returns `"object"` (historical bug)
- **undefined == null** is `true` (loose equality)
- **undefined === null** is `false` (strict equality)

OPERATORS AND CONTROL FLOW ANSWERS

11. Types of operators in JavaScript

- **Arithmetic**: `+`, `-`, `*`, `/`, `%`, `**`, `++`, `--`
- **Assignment**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- **Comparison**: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=`
- **Logical**: `&&`, `||`, `!`, `??` (nullish coalescing)
- **Bitwise**: `&`, `|`, `^`, `~`, `<<`, `>>`, `>>>`
- **Ternary**: `condition ? value1 : value2`
- **Type**: `typeof`, `instanceof`
- **Spread/Rest**: `...`

12. Difference between == and ===

- **== (Loose Equality)**: Compares values after **type coercion**
- **=== (Strict Equality)**: Compares values **AND** types (no coercion)
- == converts operands to same type before comparison
- === returns false if types are different
- **Best Practice**: Always use === to avoid unexpected results
- Same applies to `!=` (loose) vs `!==` (strict)

13. What are conditional statements?

Conditional statements execute different code blocks based on conditions:

- **if statement**: Executes code if condition is true
- **if-else**: Provides alternative when condition is false
- **else if**: Chains multiple conditions
- **switch**: Tests a value against multiple cases
- **Ternary operator**: Shorthand for simple if-else

14. Difference between `if-else` and `switch`

- **if-else**: Evaluates **boolean expressions**, more flexible
- **switch**: Compares **single value** against multiple cases
- **if-else**: Better for complex conditions and ranges
- **switch**: Better for multiple exact value matches
- **switch** uses **strict equality (===)** for comparison
- **switch** needs **break** to prevent fall-through

15. Difference between `for`, `while`, and `do-while` loops

- **for loop**: Best when iterations are **known in advance**
- **while loop**: Best when iterations depend on a **condition**
- **do-while loop**: Executes **at least once**, then checks condition

- **for**: Initialization, condition, increment in one line
- **while**: Only condition check at start
- **do-while**: Condition checked at **end** of each iteration

16. What is `break` and `continue`?

- **break**: Immediately **exits** the entire loop/switch
- **continue**: **Skips** current iteration and moves to next
- Both work with `for`, `while`, `do-while` loops
- **break** also used in `switch` statements to prevent fall-through
- Can use **labeled statements** for nested loops
- Overuse can make code harder to read

FUNCTIONS ANSWERS

17. What is a function?

A function is a **reusable block of code** designed to perform a specific task. Functions help achieve **DRY (Don't Repeat Yourself)** principle, can accept **parameters** and **return** values, creates its own **scope** for variables. Functions are **first-class citizens** in JavaScript (can be assigned, passed, returned), helps in **modularity** and **code organization**.

18. Types of functions in JavaScript

- **Named Functions**: Have a name identifier
- **Anonymous Functions**: No name, often used as callbacks
- **Arrow Functions**: ES6 shorthand syntax with `=>`
- **IIFE**: Immediately Invoked Function Expression
- **Constructor Functions**: Create objects with `new`
- **Generator Functions**: Use `function*` and `yield`
- **Async Functions**: Use `async/await` for promises

19. Function declaration vs function expression

- **Declaration**: Uses `function` keyword, can be called **before** definition (hoisted)
- **Expression**: Assigned to variable, **NOT hoisted** (only variable is hoisted)
- **Declaration**: Creates named function
- **Expression**: Can be anonymous or named
- **Declaration**: Hoisted to top of scope with function body
- **Expression**: Variable hoisted but assigned `undefined`

20. Arrow functions

ES6 feature providing shorter syntax for functions. **Implicit return** for single expressions (no curly braces). **Lexical `this`** binding (inherits `this` from enclosing scope). **Cannot** be used as constructors (`new` keyword). **No** `arguments` object (use rest parameters instead).

21. Difference between arrow function and normal function

- **`this` binding**: Normal function has own `this`, arrow function inherits `this`
- **`arguments` object**: Normal function has it, arrow function doesn't
- **Constructor**: Normal function can be used with `new`, arrow function cannot
- **Methods**: Arrow functions not suitable for object methods
- **Hoisting**: Function declarations hoisted, arrow functions not
- **Syntax**: Arrow functions are more concise

22. What is a callback function?

A **callback** is a function passed as an **argument** to another function, called/executed **after** the completion of an operation. Enables **asynchronous programming** in JavaScript. Common in event handlers, timers, and array methods. Can lead to **callback hell** (deeply nested callbacks). Foundation for **Promises** and **async/await**.

SCOPE AND HOISTING ANSWERS

23. What is scope?

Scope determines the **accessibility** and **visibility** of variables, defines where variables can be **accessed** or **referenced**. JavaScript has **three types** of scope: Global, Function, and Block. Variables are looked up through **scope chain** (inner to outer). Scope helps prevent **naming conflicts** and manage memory. Created at **compile time** (lexical/static scoping).

24. Global scope vs function scope vs block scope

- **Global Scope**: Variables declared outside any function/block, accessible everywhere
- **Function Scope**: Variables declared inside function, only accessible within function
- **Block Scope**: Variables declared inside `{}` (if, for, while), accessible only in that block
- `var` is **function-scoped**, `let` and `const` are **block-scoped**
- Global variables are attached to **window object** (in browsers)

25. What is hoisting?

Hoisting is JavaScript's behavior of moving **declarations** to the top of scope. Happens during **compilation phase** before execution. **Only declarations** are hoisted, not **initializations**. `var` is hoisted and initialized with **undefined**. `let` and `const` are hoisted but in **Temporal Dead Zone (TDZ)**. **Function declarations** are fully hoisted (body included).

26. Hoisting behavior of `var`, `let`, and `const`

- **var**: Hoisted and initialized with `undefined`, accessible before declaration
- **let**: Hoisted but NOT initialized, throws ReferenceError if accessed before declaration
- **const**: Same as `let`, plus MUST be initialized at declaration
- **Function declarations**: Fully hoisted with body
- **Function expressions**: Only variable hoisted, not the function body
- **Class declarations**: Hoisted but in TDZ (like let/const)

27. What is the temporal dead zone?

TDZ is the time between entering scope and variable **declaration being processed**. Variables exist in TDZ from **start of block** until declaration line. Accessing variable in TDZ throws **ReferenceError**. Applies to `let`, `const`, and **class declarations**. **var** does NOT have TDZ (initialized with undefined). TDZ helps catch **use before declaration** bugs.

ARRAYS AND OBJECTS ANSWERS

28. What is an array?

An array is an **ordered collection** of elements stored in a single variable. Arrays are **zero-indexed** (first element at index 0). Can hold **mixed data types** (numbers, strings, objects, etc.). Arrays are **objects** in JavaScript (`typeof [] === "object"`). **Dynamic length** - can grow or shrink. Have many built-in methods for manipulation.

29. Common array methods (`map`, `filter`, `reduce`, `forEach`)

- **map()**: Creates new array by transforming each element
- **filter()**: Creates new array with elements passing a test
- **reduce()**: Reduces array to single value (accumulator pattern)
- **forEach()**: Executes function for each element, returns undefined
- `map`, `filter`, `reduce` are **immutable** (don't modify original)
- These are **higher-order functions** (take callback as argument)

30. Difference between `map` and `forEach`

- **Return value**: `map` returns new array, `forEach` returns undefined
- **Immutability**: `map` doesn't modify original, creates new array
- **Use case**: `map` for transformation, `forEach` for side effects
- **Chainable**: `map` can be chained, `forEach` cannot
- **Performance**: Similar, but `forEach` slightly faster (no new array)
- **Break**: Neither can be stopped early (use `for...of` or `some`/`every`)

31. Difference between `filter` and `reduce`

- **filter**: Returns **array** of elements matching condition
- **reduce**: Returns **single value** (number, object, array, etc.)
- **filter**: Uses boolean callback, keeps true elements
- **reduce**: Uses accumulator callback, builds up result
- **filter**: Output length ≤ input length
- **reduce**: Can transform array into any data structure

32. What is an object?

An object is a collection of **key-value pairs** (properties). Keys are **strings** (or Symbols), values can be any type. Objects are **reference types** (stored in heap). Can contain **methods** (functions as properties). JavaScript is **object-oriented** (almost everything is an object). Created using **object literals**, **constructors**, or **Object.create()**.

33. How to access object properties?

- **Dot notation**: `object.property` (simpler, common)
- **Bracket notation**: `object["property"]` (dynamic keys)

- **Destructuring**: `const { property } = object`
- Bracket notation required for **dynamic keys** and **special characters**
- Non-existent properties return **undefined**
- Use **optional chaining** (`?.`) for safe access

34. Difference between dot notation and bracket notation

- **Dot notation**: Uses literal property name, cleaner syntax
- **Bracket notation**: Uses string/expression, more flexible
- **Dynamic keys**: Only bracket notation works
- **Special characters**: Only bracket notation (spaces, hyphens)
- **Keywords**: Bracket notation safer for reserved words
- **Performance**: Dot notation slightly faster (direct lookup)

35. What is destructuring?

Destructuring is a syntax for **extracting values** from arrays/objects. Creates variables from object properties or array elements. Supports **default values** for undefined properties. Allows **renaming** variables during extraction. Can be used in **function parameters**. Works with **nested structures**.

EXECUTION CONTEXT AND MEMORY ANSWERS

36. What is execution context?

Execution context is the **environment** where JavaScript code is evaluated and executed. Contains **Variable Environment**, **Scope Chain**, and **this binding**. **Three types**: Global, Function, and Eval execution context. Created when function is **called** (not defined). Pushed onto **call stack** when created. JavaScript starts with **Global Execution Context**.

37. What are the phases of execution context?

- **Creation Phase**: Memory allocated for variables and functions
- **Execution Phase**: Code is executed line by line
- During creation: Variables set to `undefined`, functions stored entirely
- During creation: `this` binding is determined
- During creation: Scope chain is created
- This explains **hoisting** behavior in JavaScript

38. What is call stack?

Call stack is a **data structure** that tracks function execution. Works on **LIFO** principle (Last In, First Out). Stores **execution contexts** of called functions. When function called, its context is **pushed** onto stack. When function returns, its context is **popped** off stack. **Stack overflow** occurs when stack limit exceeded.

39. What is memory heap?

Memory heap is **unstructured memory pool** for storing objects. Where **reference types** (objects, arrays, functions) are stored. Memory allocated **dynamically** as needed. Size is not fixed, can grow and shrink. Managed by **garbage collector** (automatic cleanup). **Primitives** stored in stack, **references** point to heap.

40. What is the event loop?

Event loop is the mechanism that handles **asynchronous operations**. Continuously checks if **call stack is empty**. Moves callbacks from **task queues** to call stack. Enables **non-blocking** behavior in single-threaded JavaScript. Has **microtask queue** (Promises) and **macrotask queue** (setTimeout). Microtasks have **higher priority** than macrotasks.

41. How is JavaScript single-threaded?

JavaScript has **one call stack** (one task at a time). **Single main thread** for executing code. Only one piece of code can run at any moment. **Async operations** don't block the main thread. Browser provides **Web APIs** for async (setTimeout, fetch). Node.js uses **libuv** for async operations. Event loop enables **concurrency** without multiple threads.

ASYNCHRONOUS JAVASCRIPT ANSWERS

42. What is synchronous vs asynchronous execution?

- **Synchronous**: Code executes **line by line**, blocking until complete
- **Asynchronous**: Code can **run in background**, doesn't block execution
- Synchronous waits for each operation to complete
- Asynchronous allows other code to run while waiting
- JavaScript uses **event loop** for async operations

- Network requests, timers, file I/O are typically async

43. What is callback hell?

Callback hell is **deeply nested callbacks** that form a pyramid shape. Also called **"Pyramid of Doom"**. Makes code **hard to read, maintain, and debug**. Occurs when handling multiple sequential async operations. **Error handling** becomes complex and repetitive. Solved by **Promises** and **async/await**.

44. What is a `Promise`?

A Promise is an object representing **eventual completion or failure** of async operation. Has three states: **pending**, **fulfilled**, **rejected**. Created using `new Promise((resolve, reject) => {})` **resolve()** for success, **reject()** for failure. Use **.then()** for success, **.catch()** for errors. Solves callback hell with **cleaner chaining**.

45. Promise states

- **Pending**: Initial state, neither fulfilled nor rejected
- **Fulfilled**: Operation completed successfully (resolved)
- **Rejected**: Operation failed (rejected with error)
- Promise state can only change **once** (pending → fulfilled/rejected)
- Once settled (fulfilled/rejected), state is **immutable**
- Fulfilled calls **.then()** handlers, Rejected calls **.catch()**

46. Promise chaining

Chaining connects multiple `.then()` calls sequentially. Each `.then()` returns a **new Promise**. Return value becomes the next `.then()`'s input. Allows **sequential async operations** without nesting. Single `.catch()` at end handles **any error** in chain. Can return **values** or **Promises** from `.then()`.

47. What is `async` and `await`?

- **async**: Declares a function that returns a Promise
- **await**: Pauses execution until Promise resolves
- Makes async code look and behave like **synchronous** code
- `await` can only be used inside **async functions** (or top-level modules)
- **Error handling** with try-catch instead of .catch()
- Introduced in **ES2017 (ES8)**

48. Difference between `async/await` and promises

- **Syntax**: async/await looks synchronous, Promises use .then() chains
- **Readability**: async/await more readable for sequential operations
- **Error handling**: async/await uses try-catch, Promises use .catch()
- **Debugging**: async/await easier to debug (better stack traces)
- **Underlying**: async/await is **syntactic sugar** over Promises
- **Both**: Work with Promises, can be used together

CLOSURES AND `this` KEYWORD ANSWERS

49. What is a closure?

A closure is a function that **remembers** its outer scope even after outer function returns. Inner function has access to **outer function's variables**. Variables are kept alive in memory (not garbage collected). Closures "close over" the **lexical environment**. Created every time a function is created. Foundation for many JavaScript patterns (modules, currying, etc.).

50. Real-time use cases of closures

- **Data privacy/Encapsulation**: Hide internal state from outside access
- **Function factories**: Create specialized functions with preset values
- **Callbacks with state**: Maintain state in event handlers
- **Module pattern**: Create private/public interface
- **Currying and partial application**: Transform multi-argument functions
- **Memoization**: Cache expensive computation results

51. What is the `this` keyword?

`this` refers to the **object that is executing the current function**. Value of `this` is determined at **runtime**, not definition time. In global scope: `this` = window (browser) or global (Node.js). In object method: `this` = the object calling the method. In event handler: `this` = the element that triggered event. Can be explicitly set using **call, apply, bind**.

52. How does `this` behave in different scenarios?

- **Global context**: `this` = window/global object
- **Object method**: `this` = object owning the method
- **Simple function**: `this` = window (non-strict) or undefined (strict)
- **Constructor (new)**: `this` = newly created object
- **Event handler**: `this` = element that fired event
- **Arrow function**: `this` = inherited from enclosing scope
- **call/apply/bind**: `this` = explicitly provided object

53. `this` in arrow functions vs normal functions

- **Arrow function**: Does NOT have its own `this`
- **Arrow function**: Inherits `this` from **lexical scope** (enclosing context)
- **Normal function**: Has its own `this`, determined by how it's called
- Arrow functions **cannot** have `this` changed by call/apply/bind
- Arrow functions **cannot** be used as constructors
- Arrow functions ideal for **callbacks** to preserve `this`

BROWSER AND WEB CONCEPTS ANSWERS

54. What is the DOM?

DOM stands for **Document Object Model**, tree-like **representation** of HTML document in memory. Allows JavaScript to **access and manipulate** HTML elements. Each HTML element becomes a **node** in the DOM tree. DOM is an **API** provided by the browser. Changes to DOM are **reflected** in the webpage immediately.

55. What is BOM?

BOM stands for **Browser Object Model**, allows JavaScript to **interact with the browser** (not the document). Includes **window**, **navigator**, **screen**, **location**, **history** objects. **window** is the global object in browsers. BOM is **not standardized** (varies slightly between browsers). Provides browser information, navigation, and control.

56. Difference between `localStorage`, `sessionStorage`, and cookies

- **localStorage**: Persists **forever** until manually cleared, ~5-10MB
- **sessionStorage**: Cleared when **tab/browser closes**, ~5-10MB
- **Cookies**: Can set **expiry**, sent with every HTTP request, ~4KB
- **Scope**: localStorage/sessionStorage are per origin, cookies can be domain-wide
- **Access**: localStorage/sessionStorage via JS only, cookies via JS and HTTP headers
- **Use case**: localStorage for preferences, sessionStorage for session data, cookies for auth

57. What is event bubbling?

Event bubbling is when an event **propagates from target to ancestors**. Event first triggers on **innermost element**, then bubbles up. Goes from **child → parent → grandparent → ... → document → window**. Most events bubble (click, change, keydown, etc.). Some events don't bubble (focus, blur, load). Can be stopped with **event.stopPropagation()**.

58. What is event capturing?

Event capturing is when event propagates from **ancestors to target**. Goes from **window → document → ... → grandparent → parent → target**. Also called **"trickling down"** phase. Capturing happens **BEFORE** bubbling. Enable with `addEventListener(event, handler, true)` or `{capture: true}`. Rarely used, bubbling is more common.

59. What is event delegation?

Event delegation is attaching **one handler to parent** for multiple children. Uses **event bubbling** to catch events from descendants. More **efficient** than attaching handlers to each child. Works for **dynamically added elements**. Use `event.target` to identify which child triggered event. Common pattern for lists, tables, and dynamic content.

VERY IMPORTANT QUESTIONS (SERVICE COMPANIES) ANSWERS

60. Explain JavaScript execution flow step by step

JavaScript engine receives code → **Parsing** - Code is parsed into AST (Abstract Syntax Tree) → **Creation Phase** - Global Execution Context created, variables hoisted → **Execution Phase** - Code executed line by line → Functions create new execution contexts when called → Event loop handles async operations after sync code completes.

61. Difference between synchronous and asynchronous code

- **Synchronous**: Executes **sequentially**, blocks until complete
- **Asynchronous**: Executes **independently**, doesn't block
- Sync code runs on **main thread** one at a time
- Async operations handled by **Web APIs/Node APIs**
- Async callbacks processed via **event loop**
- JavaScript is **single-threaded** but **non-blocking**

62. How does the event loop work?

Event loop is the **mechanism** enabling async in single-threaded JS. Continuously monitors **call stack** and **callback queues**. Moves callbacks to stack only when **stack is empty**. **Microtask queue** (Promises) has priority over **macrotask queue** (setTimeout). Each iteration of loop is called a **"tick"**. Enables **non-blocking** behavior in JavaScript.

63. Explain closure with a real-time example

Closure = **Function + its lexical environment**. Inner function **remembers** outer function's variables. Variables **persist** even after outer function returns. Used for **data privacy**, **state management**, **callbacks**. Created every time a function is defined inside another. See real-time examples: Counter Module, API Rate Limiter, Debounce Function, Event Handlers, etc.

64. Difference between `var`, `let`, and `const` with example

- **Scope**: `var` function-scoped, `let`/`const` block-scoped
- **Hoisting**: `var` initialized as undefined, `let`/`const` in TDZ
- **Re-declaration**: `var` allows, `let`/`const` don't
- **Re-assignment**: `var`/`let` allow, `const` doesn't
- **Global object**: `var` attaches to window, `let`/`const` don't
- **Best practice**: `const` default, `let` when re-assignment needed

65. Difference between == and === with example

- **== (Loose)**: Compares values after **type coercion**
- **=== (Strict)**: Compares values AND types, **no coercion**
- == can have **unexpected results** due to coercion
- === is **predictable** and recommended
- Same applies to `!=` vs `!==`
- Always use **strict equality** in production code

66. Explain promises and `async/await`

- **Promise**: Object representing future completion/failure of async operation
- **States**: pending → fulfilled (resolved) or rejected
- **async/await**: Syntactic sugar over Promises for cleaner code
- **async**: Makes function return a Promise
- **await**: Pauses execution until Promise resolves
- Both are used for handling asynchronous operations cleanly

67. How did you use JavaScript in your project?

- **DOM Manipulation**: Interactive UI elements, form validation
- **API Integration**: Fetch data from REST APIs
- **Event Handling**: User interactions (clicks, inputs, scrolls)
- **Form Validation**: Client-side validation before submission
- **State Management**: Managing application state
- **Asynchronous Operations**: Loading data, file uploads

FREQUENTLY ASKED QUESTIONS (SERVICE COMPANIES) ANSWERS

68. Is JavaScript single-threaded or multi-threaded?

JavaScript is **single-threaded** - one call stack, one task at a time. Main thread executes one piece of code at a time. **Event loop** enables concurrency without multiple threads. Web Workers provide **true multi-threading** for heavy tasks. Node.js uses **libuv** thread pool for I/O operations. Single-threaded but **non-blocking** through async mechanisms.

69. How does JavaScript handle asynchronous operations if it is single-threaded?

Uses **Event Loop** mechanism for concurrency. **Browser/Node.js APIs** handle async operations (setTimeout, fetch). Callbacks queued in **task queues** when operations complete. Event loop moves callbacks to stack when **stack is empty**. **Microtasks** (Promises) have priority over **macrotasks** (timers). This enables **non-blocking** behavior without multiple threads.

70. What happens when you declare a variable without `var`, `let`, or `const`?

Creates a **global variable** (attaches to window object). This is called **implicit global** - very dangerous! In **strict mode**, throws **ReferenceError**. Pollutes global namespace. Can cause **hard-to-debug bugs**. Always use `let`, `const`, or `var` to declare variables.

71. What is the difference between `undefined` and not defined?

- **undefined**: Variable declared but **not assigned** a value
- **not defined**: Variable **never declared** - ReferenceError
- `undefined` is a **type** and a **value** in JavaScript
- Accessing not defined variable throws **ReferenceError**
- `typeof undefined` returns "undefined"
- `typeof notDeclared` also returns "undefined" (safe check)

72. What is the use of strict mode?

Strict mode enables **stricter parsing** and error handling. Activated with `"use strict";` at file or function start. **Prevents** accidental globals, duplicate parameters. **Throws errors** for silent failures. **Disables** dangerous features (with, eval scope leaking). Makes code **safer and more optimizable**.

73. What is closure and where is it stored in memory?

Closure is **function + its lexical environment**. Stored in **heap memory** (not stack). Variables in closure are kept alive (not garbage collected). Each closure gets its own **environment record**. Accessible via function's internal **[[Environment]]** property. Memory released when closure is no longer referenced.

74. What is the difference between deep copy and shallow copy?

- **Shallow copy**: Copies only **top-level** properties
- **Deep copy**: Copies **all nested levels** recursively
- Shallow: Nested objects share **same reference**
- Deep: Nested objects are **completely independent**
- Shallow methods: `Object.assign()`, spread operator
- Deep methods: `JSON.parse(JSON.stringify())`, `structuredClone()`

75. How do you clone an object in JavaScript?

- **Shallow clone**: `{ ...obj }`, `Object.assign({}, obj)`
- **Deep clone**: `structuredClone(obj)`, JSON method
- For arrays: `[...arr]`, `arr.slice()`, `Array.from(arr)`
- Libraries: Lodash `_.cloneDeep()`
- Consider what needs cloning (methods, symbols, dates)
- `structuredClone()` is the modern standard for deep cloning

76. What is the difference between `null` and `undefined`?

- **undefined**: Variable declared but **not assigned**
- **null**: Intentional **absence of value** (assigned explicitly)
- `typeof undefined` → `"undefined"`
- `typeof null` → `"object"` (historical bug)
- `undefined == null` → `true` (loose equality)
- `undefined === null` → `false` (strict equality)

77. What is the difference between `call`, `apply`, and `bind`?

All three set `this` context explicitly:

- **call**: Invokes immediately, arguments passed **individually**
- **apply**: Invokes immediately, arguments passed as **array**
- **bind**: Returns **new function** with bound `this`, doesn't invoke
- Use `call` for known arguments, `apply` for arrays
- Use `bind` when you need function reference for later

78. What is event delegation and why is it used?

Attaching **single handler to parent** instead of multiple children. Uses **event bubbling** to catch events from descendants. **Why**: Memory efficiency, handles dynamic elements. Check `event.target` to identify actual clicked element. Reduces number of event listeners. Perfect for lists, tables, dynamic content.

79. What is the difference between `map`, `filter`, and `reduce`?

- **map**: Transforms each element, returns **new array of same length**
- **filter**: Keeps elements passing test, returns **filtered array**
- **reduce**: Combines all elements into **single value**
- All three don't modify original array
- Can be chained together

- `map`/`filter` more readable, `reduce` more powerful

80. What is hoisting and why does it happen?

Hoisting moves **declarations** to top of scope during compilation. Only **declarations** hoisted, not **initializations**. `var` hoisted with `undefined`, `let`/`const` in TDZ. **Functions** fully hoisted (can call before declaration). **Why**: JavaScript's two-phase execution (compile then execute). Helps with mutual recursion and flexible code organization.

81. What is temporal dead zone?

TDZ is time between **scope entry** and **variable declaration**. Variables exist but **cannot be accessed** in TDZ. Applies to `let`, `const`, and class declarations. Accessing throws **ReferenceError**. Helps catch bugs from using variables before declaration. `var` doesn't have TDZ (initialized with undefined).

82. How does garbage collection work in JavaScript?

Automatic memory management - no manual deallocation. Uses **mark-and-sweep** algorithm. Marks all **reachable** objects starting from roots. **Sweeps** (frees) unmarked objects. **Roots**: global object, call stack, closures. Memory leaks happen when unneeded references persist.

CURRYING AND PARTIAL APPLICATION ANSWERS

What is Currying?

Currying is a technique of transforming a function that takes **multiple arguments** into a sequence of functions that take **one argument each**. Named after mathematician **Haskell Curry**. Returns a new function that accepts remaining arguments. Creates **partial functions** - functions with some arguments pre-filled. Enables **functional programming** patterns. Improves **code reusability** and **composition**. Different from **partial application** (though related).

Difference between Currying and Partial Application

- **Currying**: Transforms function into sequence of **single-argument functions**
- **Partial Application**: Creates new function by **fixing some arguments** of existing function
- **Currying**: Always returns a **unary function** (takes 1 argument)
- **Partial Application**: Can return function taking **multiple arguments**
- **Currying**: More formal/structured approach
- **Partial Application**: More flexible/practical approach
- **Both** use **closures** to preserve fixed arguments

Use Cases of Currying in Real Projects

- **API Configuration**: Pre-configure base URLs and headers
- **Event Handlers**: Create specialized handlers with context
- **Validation**: Build validators for specific rules
- **Logging**: Create loggers with specific contexts/levels
- **Data Transformation**: Build reusable data pipelines
- **Testing**: Create mock functions with preset arguments
- **Configuration Management**: Store app configuration

END OF JAVASCRIPT INTERVIEW QUESTIONS GUIDE