🧠 SECTION 1: JavaScript Basics (1–25)

1. **What is JavaScript?**
   → JavaScript is a high-level, interpreted programming language used to make web pages interactive.
2. **Who developed JavaScript and when?**
   → Brendan Eich, in 1995, while working at Netscape.
3. **Is JavaScript case sensitive?**
   → Yes, JavaScript is case-sensitive.
4. **What is the difference between Java and JavaScript?**
   → Java is a compiled, object-oriented language; JavaScript is interpreted and primarily used for web scripting.
5. **What are variables in JavaScript?**
   → Variables store data values.
6. **How do you declare variables in JavaScript?**
   → Using `var`, `let`, or `const`.
7. **Difference between `var`, `let`, and `const`?**
   - `var` → function-scoped
   - `let` → block-scoped
   - `const` → block-scoped and immutable
8. **What are data types in JavaScript?**
   → Number, String, Boolean, Undefined, Null, Object, Symbol, BigInt.
9. **What is the type of `null`?**
   → It is an object (a historical bug in JS).
10. **What is `typeof` operator used for?**
    → To check the type of a variable.
11. **What is NaN?**
    → “Not-a-Number” — returned when a mathematical operation fails.
12. **What is the difference between `==` and `===`?**
    → `==` checks value only, `===` checks value and type (strict equality).
13. **What are truthy and falsy values?**
    → Values that evaluate to `true` or `false` in Boolean context.
    Falsy: `false`, `0`, `""`, `null`, `undefined`, `NaN`.
14. **What is a function?**
    → A reusable block of code that performs a specific task.
15. **How to define a function in JS?**

```
function greet() {
  console.log("Hello!");
```

```
}
```

16. **What is a parameter and argument?**
    → Parameter = variable in function definition.
    Argument = value passed during function call.
17. **What is a return statement?**
    → It returns a value from a function.
18. **What are arrow functions?**
    → Shorter syntax for functions.
    Example: `const add = (a, b) => a + b;`
19. **What is a callback function?**
    → A function passed as an argument to another function.
20. **What is an object in JavaScript?**
    → A collection of key-value pairs.
21. **How to create an object?**

```
const person = { name: "Atiar", age: 22 };
```

22. **How to access object properties?**
    → `person.name` or `person["name"]`
23. **What is an array?**
    → A list-like object that stores multiple values.
    Example: `[1, 2, 3]`
24. **How to find array length?**
    → `arr.length`
25. **What are template literals?**
    → Strings with backticks that allow variable embedding.
    Example: `` `Hello ${name}` ``

⚙️ SECTION 2: Intermediate JavaScript (26–60)

26. **What is DOM?**
    → Document Object Model — represents the structure of a web page.
27. **How to select an element by ID in DOM?**
    → `document.getElementById("id")`
28. **How to select elements by class?**
    → `document.getElementsByClassName("class")`
29. **How to select elements using query selector?**
    → `document.querySelector(".class")`
30. **How to change HTML content using JS?**
    → `element.innerHTML = "New Text";`

31. **How to change CSS with JS?**
    → `element.style.color = "red";`
32. **How to add an event listener?**
    → `element.addEventListener("click", myFunction);`
33. **What are JavaScript events?**
    → Actions that happen on a web page (e.g., click, hover, keypress).
34. **What is event bubbling?**
    → Event propagation from child to parent.
35. **What is event capturing?**
    → Event propagation from parent to child.
36. **What is JSON?**
    → JavaScript Object Notation — a lightweight data format for communication.
37. **How to convert JSON to object?**
    → `JSON.parse(jsonString)`
38. **How to convert object to JSON?**
    → `JSON.stringify(object)`
39. **What is the difference between null and undefined?**
    → `null` = empty value, `undefined` = variable declared but not assigned.
40. **What is a closure?**
    → A function that remembers variables from its outer scope.

```
function outer() {
  let x = 10;
  return function inner() {
    console.log(x);
  };
}
```

41. **What is hoisting?**
    → JS moves variable and function declarations to the top before execution.
42. **What is scope?**
    → The current context of code — determines variable access.
43. **Types of scope?**
    → Global, Function, and Block scope.
44. **What is IIFE (Immediately Invoked Function Expression)?**
    → A function executed immediately after definition.

```
(function() { console.log("Hello"); })();
```

45. **What is the difference between synchronous and asynchronous JS?**
    → Synchronous runs line-by-line; asynchronous can run concurrently.
46. **What is a Promise?**
    → Represents a value that may be available now, later, or never.

47. **What are the states of a Promise?**
    → Pending, Fulfilled, Rejected.
48. **Example of a Promise:**

```
new Promise((resolve, reject) => {
  resolve("Success");
});
```

49. **What are async/await keywords?**
    → Used for writing asynchronous code in a synchronous way.
50. **Example of async/await:**

```
async function getData() {
  let res = await fetch("data.json");
  return res.json();
}
```

51. **What is the use of `this` keyword?**
    → Refers to the current object in context.
52. **What is the difference between `call()`, `apply()`, and `bind()`?**
    - `call()` → calls with arguments list
    - `apply()` → calls with array of arguments
    - `bind()` → returns a new function
53. **What is destructuring in JS?**
    → Unpacking values from arrays or objects.

```
const [a, b] = [1, 2];
const {name, age} = person;
```

54. **What is the spread operator (`...`)?**
    → Expands iterable elements.
    Example: `[...arr1, ...arr2]`
55. **What is rest operator?**
    → Collects remaining arguments.
    Example: `function sum(...nums) { }`
56. **What are higher-order functions?**
    → Functions that take other functions as arguments or return functions.
57. **What is array `map()` used for?**
    → Transforms each array element and returns a new array.
58. **What is array `filter()` used for?**
    → Returns elements that meet a condition.
59. **What is array `reduce()` used for?**
    → Reduces an array to a single value.
60. **What is `forEach()` used for?**
    → Iterates over array elements without returning anything.

⚡ SECTION 3: Advanced JavaScript (61–100)

61. **What is ES6?**
    → ECMAScript 2015 — introduced new JS features like `let`, `const`, arrow functions, classes.
62. **What are template literals?**
    → Strings allowing embedded expressions using backticks.
63. **What are default parameters?**
    → Function parameters with default values.
    Example: `function greet(name = "User") { }`
64. **What is object shorthand?**
    → `{name, age}` instead of `{name: name, age: age}`.
65. **What is object destructuring?**
    → Extracting properties into variables.
    Example: `const {x, y} = point;`
66. **What is the difference between deep and shallow copy?**
    → Shallow copy shares references; deep copy duplicates objects.
67. **How to deep copy an object?**
    → `JSON.parse(JSON.stringify(obj))`
68. **What are classes in JavaScript?**
    → Syntax sugar for creating objects using constructors.
69. **How to define a class?**

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

70. **What is inheritance in JS classes?**
    → Using `extends` keyword to inherit another class.
71. **What are modules in JS?**
    → Used to export and import code between files.
72. **How to export a function?**
    → `export function greet() {}`
73. **How to import a function?**
    → `import { greet } from './file.js';`
74. **What is the difference between named and default export?**
    → Default = one export; Named = multiple.
75. **What is an event loop?**
    → Handles asynchronous tasks by pushing them into the callback queue.

76. **What is a callback queue?**
    → Queue where async tasks wait to be executed.
77. **What is the difference between stack and heap?**
    → Stack = stores primitive values; Heap = stores objects.
78. **What is garbage collection?**
    → Automatic memory cleanup in JS.
79. **What are prototypes in JS?**
    → Mechanism by which objects inherit properties.
80. **What is prototypal inheritance?**
    → Inheriting properties from another object via its prototype.
81. **What is the difference between `Object.create()` and class-based inheritance?**
    → `Object.create()` links directly to another object; classes use constructors.
82. **What is event delegation?**
    → Handling events on parent element instead of child.
83. **What is debouncing?**
    → Delays function execution until after some time passes.
84. **What is throttling?**
    → Limits function execution rate.
85. **What is a promise chain?**
    → Linking multiple `.then()` calls.
86. **What is microtask queue?**
    → Queue that holds promise callbacks (higher priority than task queue).
87. **What is difference between `innerHTML` and `textContent`?**
    → `innerHTML` parses HTML; `textContent` returns plain text.
88. **What is the difference between `==` and `Object.is()`?**
    → `Object.is()` compares values more strictly (handles `NaN` correctly).
89. **What is an Immediately Invoked Arrow Function?**

```
(() => console.log("Run immediately"))();
```

90. **What is the purpose of `try...catch`?**
    → Handles runtime errors gracefully.
91. **What is `finally` block used for?**
    → Executes code after `try` / `catch`, regardless of result.
92. **What are JavaScript generators?**
    → Functions that can pause execution using `yield`.
93. **What is async iterator?**
    → Used to loop over asynchronous data sources.
94. **What is the difference between synchronous and asynchronous iteration?**
    → Sync → simple loop; Async → waits for promises.
95. **What is optional chaining (`?.`)?**
    → Safely access nested properties.

Example: `user?.address?.city`

96. **What is nullish coalescing (`??`)?**
    → Returns right-hand value only if left-hand is `null` or `undefined`.
97. **What is the difference between `map()` and `forEach()`?**
    → `map()` returns new array; `forEach()` doesn't.
98. **What is the difference between `slice()` and `splice()`?**
    → `slice()` = copy; `splice()` = modify original array.
99. **What are WeakMap and WeakSet?**
    → Collections that hold weak references (don't prevent garbage collection).
100. **What is the difference between JS runtime and JS engine?**
    → Engine executes JS code (like V8), runtime provides APIs (like browser or Node.js).

---

collection of 100 deep JavaScript interview questions and answers, divided into concept levels: Core Mechanics, Advanced Concepts, Performance, and `ES6+` Features.

This set is designed for **mid to senior developer interviews**, or if you want to *master JS deeply.*

⚙️ 1. JavaScript Core Mechanics (1–25)

1. **What is the execution context in JavaScript?**
   → The environment in which JavaScript code is evaluated and executed (global, function, or eval context).
2. **What are the types of execution contexts?**
   → Global, Function, and Eval.
3. **What is the call stack?**
   → A stack data structure that tracks function calls in order of execution.
4. **Explain the event loop in JavaScript.**
   → The event loop continuously checks the call stack and callback queue to manage asynchronous operations.
5. **Difference between the call stack and task queue?**
   → The stack executes synchronous code; the queue stores asynchronous callbacks waiting to be pushed to the stack.
6. **What is the `microtask` queue?**
   → A queue that holds promises and `process.nextTick()` callbacks — executed before the task queue.

7. **What is hoisting?**
   → Moving variable and function declarations to the top of their scope before execution.
8. **Why is `var` hoisted but not `let` or `const`?**
   → `let` and `const` are hoisted but remain in a “temporal dead zone” until initialized.
9. **What is the temporal dead zone?**
   → The phase between variable hoisting and initialization when accessing the variable causes an error.
10. **What is the lexical environment?**
    → The environment where variables and functions are defined, forming scope chains.
11. **How does the scope chain work?**
    → JavaScript looks for variables in the current scope, then parent scopes, up to the global scope.
12. **What is the difference between static and dynamic scope?**
    → JS uses *lexical (static)* scoping — scope depends on where code is written, not where it’s called.
13. **What is closure in JavaScript?**
    → A function that retains access to its outer scope variables even after the outer function has returned.
14. **Example of closure use:**

```
function counter() {
  let count = 0;
  return function() {
    return ++count;
  };
}
const add = counter();
add(); // 1
```

15. **How are closures used in real-world apps?**
    → Data privacy, memoization, event handling, and function factories.
16. **What is the difference between deep and shallow copy in JS?**
    → Shallow copy copies references; deep copy duplicates nested objects.
17. **How to deep copy an object?**
    - `structuredClone(obj)`
    - `JSON.parse(JSON.stringify(obj))`
    - or libraries like Lodash `cloneDeep`.
18. **What happens when you compare two objects with `==` or `===`?**
    → Both check *references*, not *values.*
19. **What is `this` keyword in JavaScript?**
    → Refers to the context where a function is executed.
20. **What does `this` refer to in arrow functions?**
    → Lexically inherited from the parent scope (not bound dynamically).
21. **Difference between `call()`, `apply()`, and `bind()`?**

- `call()` → invoke with arguments list
- `apply()` → invoke with argument array
- `bind()` → returns new bound function

22. **How does JavaScript handle memory management?**
    → Through garbage collection — removing unreferenced objects automatically.
23. **What are WeakMaps and WeakSets used for?**
    → To store weak references — useful for caching and avoiding memory leaks.
24. **What is prototype chaining?**
    → When accessing a property, JS searches in the object, then its prototype, up the chain.
25. **How does `Object.create()` differ from class inheritance?**
    → `Object.create()` directly links to a prototype object; classes use constructor functions.

⚡ 2. Asynchronous JavaScript & Concurrency (26–50)

26. **What is asynchronous programming?**
    → A way to execute non-blocking code that runs independently of the main thread.
27. **Difference between blocking and non-blocking code?**
    → Blocking code halts execution until completion; non-blocking code continues executing other tasks.
28. **How do callbacks work in JavaScript?**
    → Functions passed as arguments that execute after another function finishes.
29. **What are callback hell and its solution?**
    → Nested callbacks causing poor readability — solved with Promises or async/await.
30. **What is a Promise in JS?**
    → An object representing the eventual completion or failure of an async operation.
31. **Promise states?**
    → Pending → Fulfilled → Rejected.
32. **What are `.then()` and `.catch()` used for?**
    → `.then()` handles resolved promises; `.catch()` handles rejections.
33. **What is `Promise.all()`?**
    → Executes multiple promises in parallel and returns results when all are resolved.
34. **What is `Promise.race()`?**
    → Returns the first promise (resolved or rejected) that completes.
35. **What is `Promise.any()`?**
    → Returns the first successfully resolved promise (ignores rejections).
36. **What is `Promise.allSettled()`?**
    → Returns results of all promises, regardless of resolution or rejection.

37. **What is async/await?**
    → Syntax sugar over Promises that allows writing asynchronous code in a synchronous style.
38. **Can `await` be used outside async functions?**
    → No, except in top-level ES modules.
39. **What happens if you forget `await` inside async?**
    → A Promise is returned instead of the resolved value.
40. **What is event loop priority between microtasks and macrotasks?**
    → Microtasks (Promises) are executed before macrotasks (setTimeout, setInterval).
41. **What is `setImmediate()` in Node.js?**
    → Executes code after the current event loop phase (macrotask queue).
42. **Difference between `setTimeout(fn, 0)` and `process.nextTick()`?**
    → `process.nextTick()` runs before the next event loop tick (microtask queue).
43. **What is an async iterator?**
    → Allows iterating over async data sources using `for await...of`.
44. **Explain concurrency vs parallelism in JS.**
    → Concurrency: managing multiple tasks at once.
    Parallelism: actually running them simultaneously (not in single-threaded JS).
45. **What is the worker thread in JS?**
    → Allows running scripts in background threads (Web Workers or Node.js Workers).
46. **What is `fetch()` API?**
    → Promise-based API to make network requests.
47. **Difference between `fetch()` and `XMLHttpRequest`?**
    → `fetch()` uses Promises and cleaner syntax; `XHR` is older and callback-based.
48. **What is CORS and why is it important?**
    → Cross-Origin Resource Sharing — controls access between different domains.
49. **How do you handle errors in async/await?**
    → Use `try...catch` block.
50. **What is a race condition in async JS?**
    → When two async operations interfere or complete in an unexpected order.

🧩 3. Advanced Functional Programming (51–75)

51. **What are higher-order functions?**
    → Functions that take or return other functions.
52. **What is functional composition?**
    → Combining multiple functions into one.
53. **What is currying?**
    → Transforming a function with multiple arguments into nested single-argument

functions.

Example: `const add = a => b => a + b;`

54. **What is partial application?**
    → Fixing some arguments of a function and returning a new function.
55. **What is immutability?**
    → Data that cannot be changed after creation.
56. **How do you enforce immutability?**
    → `Object.freeze(obj)` or using spread copies.
57. **What is memoization?**
    → Caching function results to improve performance.
58. **Example of memoization:**

```
const memo = fn => {
  const cache = {};
  return x => cache[x] || (cache[x] = fn(x));
};
```

59. **What is recursion in JS?**
    → A function calling itself to solve smaller subproblems.
60. **Tail call optimization (TCO)?**
    → JS optimization where recursive calls don't consume stack space (rarely implemented).
61. **What is the difference between imperative and declarative programming?**
    → Imperative describes *how* to do; declarative describes *what* to do.
62. **What is the difference between pure and impure functions?**
    → Pure = no side effects, same input → same output.
63. **What is function composition in JS?**
    → Combining multiple functions:

```
const compose = (f, g) => x => f(g(x));
```

64. **What are first-class functions?**
    → Functions treated as values — can be passed, returned, or assigned.
65. **What is the difference between `map()` and `reduce()`?**
    → `map()` transforms each element; `reduce()` accumulates values.
66. **What is the difference between `filter()` and `find()`?**
    → `filter()` returns all matches; `find()` returns the first.
67. **What are generators in JavaScript?**
    → Functions that yield multiple values over time with `yield` keyword.
68. **How do you pause and resume a generator?**
    → Using `next()` method.
69. **How to pass values into generators?**
    → Pass argument to `next(value)`.

70. **What is the difference between async function and generator?**
    → Async returns Promise; generator returns iterator.
71. **How to make generators asynchronous?**
    → Use `async function*`.
72. **What is the use of Symbol.iterator?**
    → Allows objects to be iterable (for-of loop compatible).
73. **What is the difference between iterator and iterable?**
    → Iterable: object implementing `[Symbol.iterator]`.
    Iterator: object returned by that function.
74. **What is the difference between `for...in` and `for...of`?**
    → `for...in` → keys (object); `for...of` → values (iterables).
75. **What is `Proxy` in JavaScript?**
    → Allows intercepting and redefining fundamental operations on objects.

🚀 4. ES6+ and Performance/Optimization (76–100)

76. **What are ES modules?**
    → JavaScript files that can `export` and `import` functions or values.
77. **Difference between default and named export?**
    → Default = one export; Named = many exports.
78. **What is dynamic import?**
    → `import()` function used for lazy-loading modules.
79. **What is destructuring assignment?**
    → Extracting properties/values directly into variables.
80. **What is spread vs rest operator?**
    → Spread expands; rest collects.
81. **What are template literals?**
    → Backtick strings allowing interpolation.
82. **What is optional chaining (`?.`)?**
    → Safe access of nested properties.
83. **What is nullish coalescing (`??`)?**
    → Returns right value if left is `null` or `undefined`.
84. **Difference between `==`, `===`, and `Object.is()`?**
    → `Object.is()` handles `NaN` and `-0` correctly.
85. **What is reflow and repaint in JS rendering?**
    → Reflow: recalculating layout.
    Repaint: updating visual elements.
86. **How can you prevent reflows?**
    → Minimize DOM changes; use `documentFragment`.

87. **What is debouncing?**
    → Delaying execution until user stops triggering an event.
88. **What is throttling?**
    → Limiting the execution rate of a function.
89. **How to optimize large DOM manipulation?**
    → Batch updates, virtual DOM, or requestAnimationFrame.
90. **What is tree shaking?**
    → Removing unused code during bundling.
91. **What are Service Workers?**
    → Scripts that enable offline caching and background sync.
92. **What is shadow DOM?**
    → A DOM subtree encapsulated from main DOM (used in Web Components).
93. **What are web components?**
    → Reusable encapsulated custom HTML elements.
94. **What is BigInt in JS?**
    → Primitive type for representing integers beyond `Number.MAX_SAFE_INTEGER`.
95. **What is the `Intl` object used for?**
    → Internationalization (dates, numbers, currencies).
96. **What is `WeakRef`?**
    → Creates weak references to objects without preventing garbage collection.
97. **What is an AbortController?**
    → Used to cancel ongoing fetch or async operations.
98. **What are the new JS features in ES2023+?**
    → Array `findLast()`, `toSorted()`, `toReversed()`, `Array.fromAsync()`.
99. **How to handle large data efficiently in JS?**
    → Use streams, generators, and lazy evaluation.
100. **What is the difference between JS runtime (Node, Browser, Deno)?**
    → All use JS engines (like V8), but provide different APIs and environments.

Got it ✅ — you're asking for **JavaScript coding-level interview questions and answers** — meaning **practical coding challenges**, not just theory.

Below is a **comprehensive set of 100 coding interview questions (with solutions)** arranged from **Beginner → Intermediate → Advanced** — perfect for frontend, MERN, or full-stack interviews.

🧩 Section 1: Beginner JavaScript Coding Questions (1–30)

1. Reverse a string

```
function reverseString(str) {
  return str.split('').reverse().join('');
}
console.log(reverseString("hello")); // "olleh"
```

---

2. Check if a string is a palindrome

```
function isPalindrome(str) {
  str = str.toLowerCase().replace(/[^a-z0-9]/g, '');
  return str === str.split('').reverse().join('');
}
console.log(isPalindrome("Madam")); // true
```

---

3. Find the largest number in an array

```
function largest(arr) {
  return Math.max( ...arr);
}
console.log(largest([2, 45, 67, 23])); // 67
```

---

4. Find factorial of a number

```
function factorial(n) {
  return n === 0 ? 1 : n * factorial(n - 1);
}
console.log(factorial(5)); // 120
```

---

5. Count vowels in a string

```
function countVowels(str) {
  return (str.match(/[aeiou]/gi) || []).length;
}
console.log(countVowels("education")); // 5
```

---

6. Find the sum of all numbers in an array

```
function sum(arr) {
  return arr.reduce((a, b) => a + b, 0);
}
console.log(sum([1, 2, 3, 4])); // 10
```

---

7. Check if a number is prime

```
function isPrime(n) {
  if (n <= 1) return false;
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) return false;
  }
  return true;
}
console.log(isPrime(7)); // true
```

---

8. Find Fibonacci sequence up to N

```
function fibonacci(n) {
  let a = 0, b = 1, res = [];
  for (let i = 0; i < n; i++) {
    res.push(a);
    [a, b] = [b, a + b];
  }
  return res;
}
console.log(fibonacci(7)); // [0,1,1,2,3,5,8]
```

---

9. Remove duplicates from array

```
function removeDuplicates(arr) {
  return [...new Set(arr)];
}
console.log(removeDuplicates([1, 2, 2, 3, 4, 4])); // [1,2,3,4]
```

---

10. Check if two strings are anagrams

```
function isAnagram(a, b) {
  return a.split('').sort().join('') === b.split('').sort().join('');
}
console.log(isAnagram("listen", "silent")); // true
```

---

11. Find the second largest number

```
function secondLargest(arr) {
  let unique = [ ...new Set(arr)];
  unique.sort((a, b) => b - a);
  return unique[1];
}
console.log(secondLargest([5, 3, 9, 9, 2])); // 5
```

---

12. Reverse words in a sentence

```
function reverseWords(str) {
  return str.split(' ').reverse().join(' ');
}
console.log(reverseWords("I love JS")); // "JS love I"
```

---

13. Find the missing number from an array (1–N)

```
function missingNumber(arr) {
  let n = arr.length + 1;
  let sum = (n * (n + 1)) / 2;
  return sum - arr.reduce((a, b) => a + b, 0);
}
console.log(missingNumber([1,2,4,5])); // 3
```

---

14. Find intersection of two arrays

```
function intersection(a, b) {
  return a.filter(x => b.includes(x));
```

```
}
console.log(intersection([1,2,3], [2,3,4])); // [2,3]
```

---

15. Flatten an array

```
function flatten(arr) {
  return arr.flat(Infinity);
}
console.log(flatten([1, [2, [3, [4]]]])); // [1,2,3,4]
```

---

16. Count occurrences of elements

```
function countOccur(arr) {
  return arr.reduce((acc, val) => {
    acc[val] = (acc[val] || 0) + 1;
    return acc;
  }, {});
}
console.log(countOccur(["a","b","a","c"])); // {a:2,b:1,c:1}
```

---

17. Find even and odd numbers separately

```
function evenOdd(arr) {
  return {
    even: arr.filter(x => x % 2 === 0),
    odd: arr.filter(x => x % 2 !== 0)
  };
}
console.log(evenOdd([1,2,3,4,5])); // {even:[2,4], odd:[1,3,5]}
```

---

18. Reverse an array without using `.reverse()`

```
function reverseArr(arr) {
  let res = [];
  for (let i = arr.length - 1; i >= 0; i--) res.push(arr[i]);
```

```
  return res;
}
```

---

19. Find maximum character in a string

```
function maxChar(str) {
  let map = {};
  for (let c of str) map[c] = (map[c] || 0) + 1;
  return Object.keys(map).reduce((a, b) => map[a] > map[b] ? a : b);
}
console.log(maxChar("javascript")); // "a"
```

---

20. Convert first letter of each word to uppercase

```
function capitalize(str) {
  return str.replace(/\b\w/g, c => c.toUpperCase());
}
console.log(capitalize("hello world")); // "Hello World"
```

⚙️ Section 2: Intermediate JavaScript Coding (31–70)

31. Check if array is sorted

```
function isSorted(arr) {
  return arr.every((x, i) => i === 0 || arr[i - 1] <= x);
}
```

---

32. Implement custom `.map()` function

```
Array.prototype.myMap = function(cb) {
  let res = [];
  for (let i = 0; i < this.length; i++) res.push(cb(this[i], i, this));
  return res;
};
console.log([1,2,3].myMap(x => x * 2)); // [2,4,6]
```

---

33. Implement custom `.filter()`

```
Array.prototype.myFilter = function(cb) {
  let res = [];
  for (let i = 0; i < this.length; i++) if (cb(this[i]))
res.push(this[i]);
  return res;
};
```

---

34. Implement `.reduce()`

```
Array.prototype.myReduce = function(cb, acc) {
  for (let i = 0; i < this.length; i++) acc = cb(acc, this[i]);
  return acc;
};
```

---

35. Shuffle an array

```
function shuffle(arr) {
  for (let i = arr.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1));
    [arr[i], arr[j]] = [arr[j], arr[i]];
  }
  return arr;
}
```

---

36. Remove falsy values

```
function removeFalsy(arr) {
  return arr.filter(Boolean);
}
```

---

37. Find union of two arrays

```
function union(a, b) {
  return [...new Set([...a, ...b])];
```

```
}
```

---

38. Find difference of arrays

```
function difference(a, b) {
  return a.filter(x => !b.includes(x));
}
```

---

39. Find common prefix of array strings

```
function commonPrefix(arr) {
  return arr.reduce((a, b) => {
    while (!b.startsWith(a)) a = a.slice(0, -1);
    return a;
  });
}
console.log(commonPrefix(["flower","flow","flight"])); // "fl"
```

---

40. Count words in a sentence

```
function wordCount(str) {
  return str.trim().split(/\s+/).length;
}
```

---

41. Implement debounce function

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}
```

---

42. Implement throttle

```
function throttle(fn, delay) {
  let last = 0;
  return (...args) => {
    let now = Date.now();
    if (now - last > delay) {
      last = now;
      fn(...args);
    }
  };
}
```

---

43. Convert object to query string

```
function toQuery(obj) {
  return Object.entries(obj)
    .map(([k, v]) => `${k}=${encodeURIComponent(v)}`).join('&');
}
```

---

44. Parse query string to object

```
function parseQuery(q) {
  return Object.fromEntries(new URLSearchParams(q));
}
```

---

45. Deep clone an object

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

---

46. Find the first non-repeating character

```
function firstUnique(str) {
  for (let c of str)
```

```
    if (str.indexOf(c) === str.lastIndexOf(c)) return c;
}
```

---

47. Implement a stack

```
class Stack {
  constructor() { this.items = []; }
  push(x) { this.items.push(x); }
  pop() { return this.items.pop(); }
}
```

---

48. Implement a queue

```
class Queue {
  constructor() { this.items = []; }
  enqueue(x) { this.items.push(x); }
  dequeue() { return this.items.shift(); }
}
```

---

49. Merge two sorted arrays

```
function merge(a, b) {
  return [...a, ...b].sort((x, y) => x - y);
}
```

---

50. Find duplicate elements

```
function findDup(arr) {
  return arr.filter((v, i) => arr.indexOf(v) !== i);
}
```

🚀 Section 3: Advanced JavaScript Coding (71–100)

71. Implement memoization

```
function memoize(fn) {
  const cache = {};
  return function( ...args) {
    const key = JSON.stringify(args);
    return cache[key] || (cache[key] = fn( ...args));
  };
}
```

---

72. Implement curry function

```
function curry(fn) {
  return function curried( ...args) {
    return args.length >= fn.length
      ? fn( ...args)
      : ( ...next) => curried( ...args,  ...next);
  };
}
```

---

73. Compose functions

```
const compose = ( ...fns) => arg => fns.reduceRight((v, f) => f(v), arg);
```

---

74. Implement binary search

```
function binarySearch(arr, target) {
  let l = 0, r = arr.length - 1;
  while (l <= r) {
    let m = Math.floor((l + r) / 2);
    if (arr[m] === target) return m;
    arr[m] < target ? l = m + 1 : r = m - 1;
  }
  return -1;
}
```

---

75. Generate all permutations

```
function permute(arr) {
  if (arr.length === 0) return [[]];
  return arr.flatMap((x, i) =>
    permute([ ...arr.slice(0, i),  ...arr.slice(i + 1)]).map(p => [x,
...p])
  );
}
```

---

76. Implement deep equality check

```
function deepEqual(a, b) {
  if (a === b) return true;
  if (typeof a !== "object" || typeof b !== "object") return false;
  const keys = Object.keys(a);
  return keys.length === Object.keys(b).length &&
         keys.every(k => deepEqual(a[k], b[k]));
}
```

---

77. Implement sleep/delay function

```
const sleep = ms => new Promise(res => setTimeout(res, ms));
```

---

78. Create async retry mechanism

```
async function retry(fn, times) {
  while (times--) {
    try { return await fn(); } catch (e) { if (!times) throw e; }
  }
}
```

---

79. Convert callback to Promise

```
function promisify(fn) {
  return (...args) =>
    new Promise((res, rej) => fn(...args, (err, data) => err ? rej(err) :
```

```
res(data)));
}
```

---

80. Limit concurrent async calls

```
async function limitConcurrency(tasks, limit) {
  const pool = [];
  for (const task of tasks) {
    const p = task().finally(() => pool.splice(pool.indexOf(p), 1));
    pool.push(p);
    if (pool.length >= limit) await Promise.race(pool);
  }
  return Promise.all(pool);
}
```

---

deep into the Advanced JavaScript Coding Interview Questions (81–100) — focused on recursion, async patterns, data structures, and problem-solving that test real coding ability.

🚀 Advanced JavaScript Coding Interview Questions (81–100)

---

81. Implement `Promise.all()` manually

```
function myPromiseAll(promises) {
  return new Promise((resolve, reject) => {
    let results = [], completed = 0;
    promises.forEach((p, i) => {
      Promise.resolve(p).then(val => {
        results[i] = val;
        if (++completed === promises.length) resolve(results);
      }).catch(reject);
    });
  });
}
```

✅ *Checks understanding of Promises and async flow.*

---

82. Implement `Promise.race()` manually

```
function myPromiseRace(promises) {
  return new Promise((resolve, reject) => {
    promises.forEach(p => Promise.resolve(p).then(resolve, reject));
  });
}
```

---

83. Deep flatten an object

```
function flattenObject(obj, parent = '', res = {}) {
  for (let key in obj) {
    const prop = parent ? `${parent}.${key}` : key;
    if (typeof obj[key] === 'object' && obj[key] !== null)
      flattenObject(obj[key], prop, res);
    else res[prop] = obj[key];
  }
  return res;
}

console.log(flattenObject({ a: { b: { c: 5 } }, d: 6 }));
// { "a.b.c": 5, "d": 6 }
```

---

84. Implement event emitter

```
class EventEmitter {
  constructor() { this.events = {}; }

  on(event, listener) {
    (this.events[event] = this.events[event] || []).push(listener);
  }

  emit(event, ...args) {
    (this.events[event] || []).forEach(fn => fn(...args));
  }

  off(event, listener) {
    this.events[event] = (this.events[event] || []).filter(fn => fn !==
```

```
listener);
  }
}
```

---

85. Create a function to chain promises sequentially

```
async function runSequentially(tasks) {
  const results = [];
  for (let task of tasks) results.push(await task());
  return results;
}
```

---

86. Implement LRU Cache

```
class LRUCache {
  constructor(limit) {
    this.limit = limit;
    this.cache = new Map();
  }
  get(key) {
    if (!this.cache.has(key)) return -1;
    const val = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, val);
    return val;
  }
  put(key, val) {
    if (this.cache.has(key)) this.cache.delete(key);
    else if (this.cache.size >= this.limit)
      this.cache.delete(this.cache.keys().next().value);
    this.cache.set(key, val);
  }
}
```

✅ *Tests Map usage and algorithmic thinking.*

---

87. Implement a function to check balanced parentheses

```
function isBalanced(str) {
  const stack = [];
```

```
  const map = { ')':'(', ']':'[', '}':'{' };
  for (let c of str) {
    if (['(', '[', '{'].includes(c)) stack.push(c);
    else if (map[c] && stack.pop() !== map[c]) return false;
  }
  return !stack.length;
}
```

---

88. Implement binary tree traversal (inorder)

```
function inorder(root) {
  if (!root) return [];
  return [ ... inorder(root.left), root.val,  ... inorder(root.right)];
}
```

---

89. Find all subsets of an array

```
function subsets(arr) {
  let res = [[]];
  for (let num of arr)
    res = [ ... res,  ... res.map(r => [ ... r, num])];
  return res;
}
console.log(subsets([1,2]));
// [[],[1],[2],[1,2]]
```

---

90. Implement `bind()` manually

```
Function.prototype.myBind = function(context,  ... args) {
  const fn = this;
  return function( ... rest) {
    return fn.apply(context, [ ... args,  ... rest]);
  };
};
```

---

91. Detect a cycle in a linked list

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

---

92. Implement binary search recursively

```
function binarySearchRec(arr, target, low = 0, high = arr.length - 1) {
  if (low > high) return -1;
  let mid = Math.floor((low + high) / 2);
  if (arr[mid] === target) return mid;
  return arr[mid] > target
    ? binarySearchRec(arr, target, low, mid - 1)
    : binarySearchRec(arr, target, mid + 1, high);
}
```

---

93. Implement `new` keyword manually

```
function myNew(constructor, ...args) {
  const obj = Object.create(constructor.prototype);
  const res = constructor.apply(obj, args);
  return res instanceof Object ? res : obj;
}
```

---

94. Create a retryable API fetch

```
async function fetchWithRetry(url, retries = 3) {
  while (retries--) {
    try { return await fetch(url); }
    catch (err) {
      if (!retries) throw err;
    }
  }
}
```

95. Detect deep nested object key

```
function hasKey(obj, key) {
  if (obj.hasOwnProperty(key)) return true;
  return Object.values(obj).some(v =>
    typeof v === 'object' && hasKey(v, key)
  );
}
```

---

96. Group an array of objects by property

```
function groupBy(arr, key) {
  return arr.reduce((acc, obj) => {
    (acc[obj[key]] = acc[obj[key]] || []).push(obj);
    return acc;
  }, {});
}
```

---

97. Implement observer pattern

```
class Observer {
  constructor() { this.subs = []; }
  subscribe(fn) { this.subs.push(fn); }
  unsubscribe(fn) { this.subs = this.subs.filter(f => f !== fn); }
  notify(data) { this.subs.forEach(fn => fn(data)); }
}
```

---

98. Implement async pool (limit concurrent promises)

```
async function asyncPool(poolLimit, tasks) {
  const results = [];
  const executing = [];

  for (const task of tasks) {
    const p = Promise.resolve().then(() => task());
    results.push(p);

    if (poolLimit <= tasks.length) {
```

```
      const e = p.then(() => executing.splice(executing.indexOf(e), 1));
      executing.push(e);
      if (executing.length >= poolLimit) await Promise.race(executing);
    }
  }
  return Promise.all(results);
}
```

---

99. Implement cloneDeep without JSON

```
function cloneDeep(obj, map = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') return obj;
  if (map.has(obj)) return map.get(obj);
  const copy = Array.isArray(obj) ? [] : {};
  map.set(obj, copy);
  for (let key in obj) copy[key] = cloneDeep(obj[key], map);
  return copy;
}
```

---

100. Implement custom `setInterval()` using `setTimeout()`

```
function customInterval(fn, delay) {
  let timer;
  const loop = () => {
    fn();
    timer = setTimeout(loop, delay);
  };
  loop();
  return () => clearTimeout(timer);
}

const stop = customInterval(() => console.log('Run'), 1000);
setTimeout(stop, 5000);
```

🧠 Bonus Topics Tested in Real Interviews

- Event loop, microtasks, macrotasks
- Closures and lexical scope
- Prototype inheritance
- Async/await internal mechanics

- Debounce vs Throttle
- Hoisting and temporal dead zone
- Garbage collection
- Time complexity analysis
- Currying and composition in React
