# 70+ JavaScript Coding Round

Interview Questions and Answers

Sai Reddy 5/26/25 Interview Tips

# 70+ JavaScript Coding Round Interview Q&A

1. **Reverse a String**

```
function reverseStr(str) {
  let rev = "";
  for (let i = str.length - 1; i >= 0; i--) rev += str[i];
  return rev;
}
```

*We loop from the end and build a new string character by character.*

2. **Check if a string is a palindrome**

```
function isPalindrome(str) {
  let len = str.length;
  for (let i = 0; i < len / 2; i++)
    if (str[i] !== str[len - 1 - i]) return false;
  return true;
}
```

*Compare characters from both ends toward the center.*

3. **Find largest number in array**

```
function findMax(arr) {
  let max = arr[0];
  for (let i = 1; i < arr.length; i++)
    if (arr[i] > max) max = arr[i];
  return max;
}
```

*Iterate and compare each element to update max value.*

4. **Remove duplicates from array**

```
function removeDup(arr) {
  let result = [], seen = {};
  for (let i = 0; i < arr.length; i++)
    if (!seen[arr[i]]) result.push(arr[i]), seen[arr[i]] = true;
  return result;
}
```

*Use an object to track already added elements.*

5. **Flatten a nested array (1 level)**

```
function flatten(arr) {
  let flat = [];
  for (let i = 0; i < arr.length; i++)
    if (Array.isArray(arr[i]))
      for (let j = 0; j < arr[i].length; j++) flat.push(arr[i][j]);
    else flat.push(arr[i]);
  return flat;
}
```

*Loop through and handle nested arrays manually.*

6. **FizzBuzz**

```
function fizzBuzz(n) {
  for (let i = 1; i <= n; i++) {
    if (i % 15 === 0) console.log("FizzBuzz");
    else if (i % 3 === 0) console.log("Fizz");
    else if (i % 5 === 0) console.log("Buzz");
    else console.log(i);
  }
}
```

*Use conditional logic with modulo checks.*

7. **Sum of array elements**

```
function arraySum(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) sum += arr[i];
  return sum;
}
```

*Add each element in loop.*

8. **Factorial using recursion**

```
function fact(n) {
  if (n <= 1) return 1;
  return n * fact(n - 1);
}
```

*Recursive function multiplying* `n` *with factorial of* `n-1`.

9. **Debounce function**

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

*Clears previous timer and resets delay before calling.*

## 10. Throttle function

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

*Limits function calls to once per delay interval.*

## 11. Count vowels in string

```
function countVowels(str) {
  let count = 0, vowels = "aeiouAEIOU";
  for (let i = 0; i < str.length; i++)
    if (vowels.indexOf(str[i]) !== -1) count++;
  return count;
}
```

*Check each char against known vowels.*

## 12. Second largest in array

```
function secondMax(arr) {
  let max = -Infinity, second = -Infinity;
  for (let i = 0; i < arr.length; i++) {
    if (arr[i] > max) second = max, max = arr[i];
    else if (arr[i] > second && arr[i] !== max) second = arr[i];
  }
  return second;
}
```

*Track both max and second max.*

## 13. Anagram check

```
function isAnagram(a, b) {
  if (a.length !== b.length) return false;
  let count = {};
  for (let i = 0; i < a.length; i++) count[a[i]] = (count[a[i]] || 0) + 1;
  for (let i = 0; i < b.length; i++) if (!count[b[i]]--) return false;
  return true;
}
```

*Use char frequency counters.*

## 14. Merge two sorted arrays

```
function mergeSorted(a, b) {
  let res = [], i = 0, j = 0;
  while (i < a.length && j < b.length)
    res.push(a[i] < b[j] ? a[i++] : b[j++]);
  while (i < a.length) res.push(a[i++]);
  while (j < b.length) res.push(b[j++]);
  return res;
}
```

*Two-pointer technique.*

## 15. Convert to title case

```
function toTitleCase(str) {
  let result = "", cap = true;
  for (let i = 0; i < str.length; i++) {
    let ch = str[i];
    if (ch === " ") cap = true, result += ch;
    else result += cap ? ch.toUpperCase() : ch.toLowerCase(), cap = false;
  }
  return result;
}
```

*Capitalize first letter after space.*

## 16. Check if number is prime

```
function isPrime(n) {
  if (n < 2) return false;
  for (let i = 2; i * i <= n; i++)
    if (n % i === 0) return false;
  return true;
}
```

*Divide by numbers up to sqrt(n).*

## 17. Fibonacci using recursion

```
function fib(n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}
```

*Each term is sum of previous two.*

## 18. Find missing number in 1–N

```
function findMissing(arr, n) {
  let total = (n * (n + 1)) / 2;
  let sum = 0;
  for (let i = 0; i < arr.length; i++) sum += arr[i];
  return total - sum;
}
```

*Use sum formula and subtract.*

## 19. Swap two vars without temp

```
function swap(a, b) {
  a = a + b;
  b = a - b;
  a = a - b;
  return [a, b];
}
```

*Use math to swap.*

## 20. Simple email regex validation

```
function validateEmail(email) {
  let regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return regex.test(email);
}
```

*Basic structure check for email.*

## 21. Object ↔ Array conversion

```
function objToArr(obj) {
  let res = [];
  for (let key in obj) res.push([key, obj[key]]);
  return res;
}
function arrToObj(arr) {
  let obj = {};
  for (let i = 0; i < arr.length; i++) obj[arr[i][0]] = arr[i][1];
  return obj;
}
```

*Manual conversion using loops.*

## 22. Sort array of numbers

```
function bubbleSort(arr) {
  for (let i = 0; i < arr.length - 1; i++)
    for (let j = 0; j < arr.length - i - 1; j++)
      if (arr[j] > arr[j + 1]) [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
  return arr;
}
```

*Classic bubble sort implementation.*

## 23. Unique values from array of objects

```
function getUnique(arr, key) {
  let result = [], seen = {};
  for (let i = 0; i < arr.length; i++)
    if (!seen[arr[i][key]]) {
      seen[arr[i][key]] = true;
      result.push(arr[i]);
    }
  return result;
}
```

*Use object to track unique keys.*

## 24. Deep clone an object

```
function deepClone(obj) {
  if (obj === null || typeof obj !== 'object') return obj;
  let copy = Array.isArray(obj) ? [] : {};
  for (let key in obj) copy[key] = deepClone(obj[key]);
  return copy;
}
```

*Recursive traversal and copy.*

## 25. == vs === difference

```
let a = "5", b = 5;
console.log(a == b);  // true, because of type coercion
console.log(a === b); // false, checks type + value
```

*== allows coercion; === is strict check.*

**3-7 y Exp**

## 1. Deep Clone Method

Function deepClone(obj) {

If (obj === null || typeof obj !== 'object') return obj;

Let copy = Array.isArray(obj) ? [] : {};

For (let key in obj) copy[key] = deepClone(obj[key]);

```
  Return copy;
}
```

Recursively clone each property. Handles objects/arrays deeply.

## 2. Custom Array.map()

```
Array.prototype.myMap = function(cb) {
  Let res = [];
  For (let i = 0; i < this.length; i++) res.push(cb(this[i], i, this));
  Return res;
};
```

Loop through array and apply callback to each item manually.

## 3. Custom Promise.all()

```
Function myPromiseAll(promises) {
  Return new Promise((res, rej) => {
    Let out = [], count = 0;
    Promises.forEach((p, i) => Promise.resolve(p).then(val => {
      Out[i] = val; if (++count === promises.length) res(out);
    }, rej));
  });
}
```

Track resolution and index, resolve all or reject one.

## 4. Memoization

```
Function memo(fn) {
  Let cache = {};
  Return function(n) {
    If (cache[n]) return cache[n];
    Return cache[n] = fn(n);
  };
}
```

Store results in cache to avoid re-computation.

## 5. Event Delegation

```
Document.getElementById("parent").addEventListener("click", e => {
  If (e.target.matches("button")) alert("Button clicked: " + e.target.textContent);
});
```

Attach one listener to parent, check target element.

## 6. Currying Function

```
Function curry(a) {
  Return function(b) {
    Return function(c) {
      Return a + b + c;
    };
  };
}
```

Returns nested functions each taking one argument.

## 7. Debounce vs Throttle

```
Function debounce(fn, delay) {
  Let t; return function() { clearTimeout(t); t = setTimeout(fn, delay); };
}
Function throttle(fn, delay) {
  Let last = 0; return function() {
    Let now = Date.now(); if (now – last > delay) fn(), last = now;
  };
}
```

Debounce delays after stop, throttle limits rate.

## 8. Chaining

```
Class Calc {
  Constructor(v = 0) { this.val = v; }
  Add(n) { this.val += n; return this; }
  Sub(n) { this.val -= n; return this; }
  Result() { return this.val; }
}
```

Each method returns this to allow chaining calls.

## 9. LRU Cache

```
Class LRU {
  Constructor(size) { this.map = new Map(); this.size = size; }
  Get(k) { if (!this.map.has(k)) return -1;
    Let v = this.map.get(k); this.map.delete(k); this.map.set(k, v); return v; }
  Put(k, v) { if (this.map.has(k)) this.map.delete(k);
    If (this.map.size === this.size) this.map.delete(this.map.keys().next().value);
    This.map.set(k, v); }
}
```

Map used for quick access and insertion order tracking.

**10. Custom Bind**

```
Function.prototype.myBind = function(ctx, ...args1) {
  Let fn = this;
  Return function(...args2) {
    Return fn.apply(ctx, [...args1, ...args2]);
  };
};
```

Capture context and arguments, return bound function.

**11. Async/Await Error Handling**

```
Async function fetchData() {
  Try {
    Let res = await fetch('url');
    Let data = await res.json();
  } catch (err) {
    Console.error("Error:", err);
  }
}
```

Wrap in try/catch to handle async errors gracefully.

## 12. Longest Substring Without Repeat

```
Function longestSubstr(str) {
  Let set = {}, max = 0, start = 0;
  For (let i = 0; i < str.length; i++) {
    If (set[str[i]] >= start) start = set[str[i]] + 1;
    Set[str[i]] = i; max = Math.max(max, i – start + 1);
  }
  Return max;
}
```

Sliding window using character index map.

## 13. Detect Cycle in 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;
}
```

Floyd's Tortoise & Hare cycle detection.

## 14. Call, Apply, Bind Differences

```
Function greet(msg) { console.log(msg + " " + this.name); }

Let obj = { name: "Sam" };

Greet.call(obj, "Hi"); greet.apply(obj, ["Hi"]); let f = greet.bind(obj, "Hi"); f();
```

Call: args comma. Apply: args array. Bind: returns function.

## 15. Flatten Deep Object

```
Function flatten(obj, path = '', res = {}) {
  For (let key in obj) {
    Let newKey = path ? path + '.' + key : key;
    If (typeof obj[key] === 'object') flatten(obj[key], newKey, res);
    Else res[newKey] = obj[key];
  }
  Return res;
}
```

Recursively build dot notation keys.

## 16. Sleep Using Promise

```
Function sleep(ms) {
  Return new Promise(resolve => setTimeout(resolve, ms));
}
```

Wrap setTimeout in a promise to await sleep.

## 17. Promisify Callback

```
Function promisify(fn) {
  Return function(...args) {
    Return new Promise((res, rej) => fn(...args, (e, d) => e ? rej(e) : res(d)));
  };
}
```

Convert callback (err, data) to promise format.

### 18. Throttled Scroll Event

```
Window.addEventListener("scroll", throttle(() => {
  Console.log("Scroll fired");
}, 200));
```

Throttled function limits scroll handler execution.

### 19. Closure for Private Variable

```
Function counter() {
  Let count = 0;
  Return { inc: () => ++count, dec: () => --count, get: () => count };
}
```

Count remains private via closure, accessed via methods.

### 20. Rate Limiter

```
Function rateLimiter(fn, limit) {
  Let calls = 0, queue = [];
  Return function(...args) {
    If (calls < limit) calls++, fn(...args), setTimeout(() => calls--, 1000);
    Else queue.push(() => fn(...args));
  };
}
```

Allow limit calls/sec and queue others.

## 21. Array.reduce Polyfill

```
Array.prototype.myReduce = function(cb, init) {
  Let acc = init, i = 0;
  If (acc === undefined) acc = this[i++];
  For (; i < this.length; i++) acc = cb(acc, this[i], i, this);
  Return acc;
};
```

Iterate and accumulate manually using initial value.

## 22. Set and Map Use

```
Let set = new Set([1, 2, 2]); set.add(3);
Let map = new Map(); map.set("a", 1); map.get("a");
```

Set stores unique values, Map stores key-value pairs efficiently.

## 23. Sort Array of Objects by Key

```
Function sortByKey(arr, key) {
  For (let i = 0; i < arr.length – 1; i++)
    For (let j = 0; j < arr.length – i – 1; j++)
      If (arr[j][key] > arr[j + 1][key]) [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
  Return arr;
}
```

Bubble sort objects based on key.

## 24. Binary Search

```
Function binarySearch(arr, t) {
  Let l = 0, r = arr.length – 1;
  While (l <= r) {
    Let m = Math.floor((l + r) / 2);
    If (arr[m] === t) return m;

    If (arr[m] < t) l = m + 1; else r = m – 1;

  }

  Return -1;

}
```

Divide and conquer search in sorted array.

### 25. Debounce using setTimeout

```
Function debounce(fn, delay) {

  Let timer;

  Return function() {

    clearTimeout(timer);

    timer = setTimeout(fn, delay);

  };

}
```

Delay execution until user stops triggering for delay ms.

## **7 to 14 years Experience**

### 1. Polyfill for Promise

```
function MyPromise(exec) {
  this.thenCb = null;
  const resolve = val => this.thenCb && this.thenCb(val);
  exec(resolve);
}
MyPromise.prototype.then = function(cb) { this.thenCb = cb; };
```

*Stores callback and invokes after resolution.*

### 2. Async Task Queue with Concurrency

```
class TaskQueue {
  constructor(limit) { this.tasks = []; this.running = 0; this.limit =
limit; }
  add(task) {
    this.tasks.push(task); this.run();
  }
  run() {
    if (this.running >= this.limit || this.tasks.length === 0) return;
    this.running++; this.tasks.shift()().finally(() => { this.running--;
this.run(); });
  }
}
```

*Controls number of tasks running concurrently.*

### 3. Custom Observable

```
class Observable {
  constructor(sub) { this.sub = sub; }
  subscribe(obs) { this.sub(obs); }
}
new Observable(obs => { obs.next(1); }).subscribe({ next: v =>
console.log(v) });
```

*Encapsulates producer logic, invokes observer.*

### 4. Event Emitter

```
class Emitter {
  constructor() { this.events = {}; }
  on(e, fn) { (this.events[e] ||= []).push(fn); }
  emit(e, ...a) { (this.events[e] || []).forEach(f => f(...a)); }
}
```

*Stores event listeners and invokes them on emit.*

### 5. In-memory Key-Value Store

```
class Store {
  constructor() { this.data = {}; }
  set(k, v) { this.data[k] = v; }
  get(k) { return this.data[k]; }
  delete(k) { delete this.data[k]; }
}
```

*Simple object-backed memory storage.*

### 6. Virtual DOM & Diffing

```
function diff(oldNode, newNode) {
  if (oldNode !== newNode) console.log("Changed:", newNode);
}
diff("div", "span");
```

*Compares nodes and logs change; extend for full VDOM.*

### 7. Web Worker Use Case

```
// worker.js
onmessage = e => postMessage(e.data * 2);

// main.js
let w = new Worker("worker.js");
w.postMessage(10); w.onmessage = e => console.log(e.data);
```

*Runs heavy computation off main thread.*

## 8. Middleware Pipeline

```
function compose(mws) {
  return ctx => mws.reduce((a, f) => () => f(ctx, a), () => {})(ctx);
}
```

*Composes functions like Express middlewares.*

## 9. State Management Store

```
function createStore(r, init) {
  let state = init, subs = [];
  return {
    dispatch: a => { state = r(state, a); subs.forEach(fn => fn()); },
    subscribe: fn => subs.push(fn),
    getState: () => state
  };
}
```

*Reducer pattern with subscription system.*

## 10. Custom Hook-like Function

```
function useState(init) {
  let val = init;
  return [() => val, v => val = v];
}
```

*Manages internal state with getter/setter.*

## 11. Pub/Sub System

```
class PubSub {
  constructor() { this.events = {}; }
  subscribe(e, fn) { (this.events[e] ||= []).push(fn); }
  publish(e, data) { (this.events[e] || []).forEach(fn => fn(data)); }
}
```

*Similar to emitter but for broadcast-based logic.*

## 12. Code Splitting (Dynamic Import)

```
button.onclick = () => import('./module.js').then(m => m.run());
```

*Load modules on-demand to reduce bundle size.*

## 13. Memory Leak Detection

```
let arr = [];
function leaky() { arr.push(new Array(1000000)); }
// Use Chrome DevTools to monitor heap
```

*Detect unfreed memory via DevTools profiling.*

## 14. Performance Optimization

```
console.time("loop");
for (let i = 0; i < 1e6; i++) {}
console.timeEnd("loop");
```

*Use console timers or `performance.now()` to benchmark.*

## 15. Service Worker to Cache Assets

```
self.addEventListener("install", e => {
  e.waitUntil(caches.open("v1").then(c => c.addAll(["/index.html"])));
});
```

*Intercepts network and caches static files.*

## 16. Lazy Loading Component

```
function lazyLoad(view) {
  return import(`./${view}.js`).then(m => m.default());
}
```

*Loads UI module when needed.*

## 17. Custom Validation Schema

```
function validate(obj, schema) {
  for (let key in schema) {
    if (!schema[key](obj[key])) return false;
  }
  return true;
}
```

*Schema defines validators; applied to object fields.*

## 18. Chunked File Upload with Retry

```
async function upload(file, size = 1024 * 1024) {
  let i = 0;
  while (i < file.size) {
    let chunk = file.slice(i, i + size);
    await fetch('/upload', { method: "POST", body: chunk });
    i += size;
  }
}
```

*Reads file by chunks and uploads sequentially.*

## 19. WebSocket Client

```
let ws = new WebSocket("wss://server");
ws.onopen = () => ws.send("Hello");
ws.onmessage = e => console.log(e.data);
```

*Two-way communication with server using WebSocket.*

## 20. Retry with Exponential Backoff

```
async function retry(fn, retries = 3, delay = 500) {
  try { return await fn(); }
  catch (e) { if (retries === 0) throw e; await new Promise(r =>
setTimeout(r, delay)); return retry(fn, retries - 1, delay * 2); }
}
```

*Retries failed task with increasing delay.*

## 21. Circuit Breaker Pattern

```
class Circuit {
  constructor(fn, failLimit = 3) {
    this.fn = fn; this.fail = 0; this.limit = failLimit; this.open = false;
  }
  async call(...a) {
    if (this.open) throw "Circuit Open";
    try { return await this.fn(...a); this.fail = 0; }
    catch { if (++this.fail >= this.limit) this.open = true; throw "Fail";
}
  }
}
```

*Blocks calls after threshold failures.*

## 22. Throttle/Debounce with Cancel

```
function debounce(fn, d) {
  let t; const wrapper = function() {
    clearTimeout(t); t = setTimeout(() => fn(), d);
  };
  wrapper.cancel = () => clearTimeout(t);
  return wrapper;
}
```

*Attach `cancel()` to cancel pending debounce.*

## 23. Intersection Observer Example

```
let obs = new IntersectionObserver(entries => {
  entries.forEach(e => { if (e.isIntersecting) console.log("Visible"); });
});
obs.observe(document.querySelector("#target"));
```

*Triggers callback when element enters viewport.*

## 24. Polyfill for Object.create

```
function create(proto) {
  function F() {} F.prototype = proto;
  return new F();
}
```

*Mimics object instantiation with prototype linkage.*

## 25. Infinite Scroll Design

```
window.onscroll = () => {
  if (window.innerHeight + window.scrollY >= document.body.offsetHeight)
    loadMoreItems();
};
```

*Loads more content when scrolled to bottom.*