Module 16 of 16

# Interview idioms

The coding-round template and the JavaScript decisions to make before being asked; the twelve algorithm patterns; the idiom sheet and the trap behind each idiom; the reimplementation questions and the spec details that decide them; the theory repertoire in two-sentence answers.

---

The last module turns fifteen modules of language knowledge into interview performance. A coding round is a conversation with a program in the middle, and it is won or lost on process: restating the problem, probing its edges, naming the brute force, coding aloud, tracing an example, stating complexity. A theory round is a finite repertoire answered crisply. A reimplementation question hinges on one spec detail the interviewer already has in mind. This module makes each of those explicit.

The first lesson is the template and the JavaScript-specific decisions — `Map` over object, comparators, index-pointer queues, code points, `BigInt`, `Array.from` grids. The second names the twelve patterns behind most problems, the signs that identify each, and its JavaScript shape. The third is the idiom sheet: the collection, string, object, number and function one-liners, each with the trap inside it. The fourth covers the reimplementation questions — `bind`, `new`, the array methods, `Promise.all`, a `Promise` class, `JSON.stringify`, deep clone and equality — and the details that decide them. The fifth is the theory repertoire as flashcards with two-sentence answers and pointers to depth.

The exercises are interview problems done the interview way — two-sum three ways with operation counts, a test table with edge cases, sliding windows and two pointers, stacks and BFS, the idiom sheets as command interpreters, `bind` / `new` / `instanceof` and the array methods checked against the natives, a Leitner flashcard scheduler and an answer grader — then the three senior-round reimplementations: a `Promise` class with adoption and the four combinators, a `JSON.stringify` that matches the native one, and a deep-equality and deep-clone toolkit reported as TAP.

CONTENTS


| | | |
|---|---|---|
| 1 | The template — running a JavaScript coding round from your side of the table | 12 min |
| 2 | The patterns — the dozen algorithm shapes behind most JavaScript rounds | 14 min |
| 3 | The idiom sheet — the one-liners worth knowing by heart, and the traps inside them | 11 min |

| | | |
|---|---|---|
| 4 | Implement the built-in — the reimplementation questions and the details that decide them | 14 min |
| 5 | Theory drill — the questions, the two-sentence answers, and where each one goes deeper | 14 min |
| ✓ | Checkpoint — Interview idioms | 26 min |

Exercises in this module compile on Node 16. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

# The template — running a JavaScript coding round from your side of the table

A coding interview is a forty-five-minute conversation with a program in the middle, and most of it is lost before the first line of code: on a problem misread, an edge case never asked about, a brute force never stated, or a solution typed in silence. This lesson is the template that senior candidates follow without thinking — restate, probe, plan, code aloud, trace, analyse — and the JavaScript-specific decisions you should make before the interviewer has to ask. Nothing here is a trick; it is the order of operations that makes the rest of this module usable under pressure.

---

## The six steps

**Restate** the problem in one sentence in your own words, including the return type. "Given an array of integers and a target, return the indices of the two numbers that sum to it — one answer guaranteed." This catches misreadings immediately and shows you separate the *what* from the *how*.

**Probe** — ask the questions that change the solution: input size (does an O(n²) pass?), value ranges (negatives? duplicates? overflow beyond $2^{53}$?), empty and single-element inputs, whether the input can be mutated, whether the output order matters, what to return when there is no answer (`-1`, `null`, an empty array, throw?), and — for strings — case, whitespace and Unicode. Write the answers down as **examples**: a normal case, an edge case, and one you expect to trip a naive solution.

**Plan aloud** — state the brute force and its complexity first, even when you already see something better ("brute force is every pair, O(n²) time, O(1) space"). Then improve it: what repeated work is there? What data structure removes it? Name the target complexity before coding. If you cannot see an improvement, say what you would try and pick the brute force — a working O(n²) beats a broken O(n).

**Code aloud** — narrate the intent of each block, not each character ("I'll count frequencies in a Map, then walk the string once more"). Write helpers as you need them; leave `// TODO` for validation you will return to; do not pause to golf.

**Trace** — before declaring done, run your second example through the code line by line with real values in a comment or on the whiteboard. This is where off-by-ones, wrong

initial values and a `<` that should be `⩽` surface; finding them yourself is worth more than never making them.

**Analyse** — time and space, with a sentence of justification ("one pass, constant work per element; the Map holds up to n entries"). Then offer the follow-ups you can see: streaming input, memory limits, concurrency, what changes if the array is sorted.

---

## Choices to make before you are asked

`Map` **over an object** for counting or lookup keyed by data: keys keep their type (`1` and `"1"` differ), there is no prototype to collide with, `size` is free, insertion order is preserved. `Set` for membership.

**Sorting numbers** needs a comparator — `[10, 9, 1].sort()` gives `[1, 10, 9]`. `arr.sort((a, b) ⇒ a - b)`, and `localeCompare` for strings when the order must be human.

**Queues** are an index pointer over an array or two stacks — never `shift()` in a loop (O(n) each; module 15).

**Strings** are immutable; build output in an array and `join`. Iterate with `for…of` or `[...s]` to get code points, not UTF-16 halves (module 10). Say which one you mean when the problem involves "characters".

**Integers** are safe to $2^{53}$; sums of large arrays, factorials and hashes need `BigInt` or modular arithmetic. Bitwise operators truncate to 32 bits — say so if you use them.

**Grid and matrix**: `Array.from({ length: rows }, () ⇒ Array(cols).fill(0))` — never `Array(rows).fill([])` (one shared row).

**Recursion** depth is ~$10^4$ frames; a DFS over a $10^6$-node graph needs an explicit stack.

**Mutation**: ask whether you may mutate the input; if not, copy once (`[...arr]`) and say why.

**Output**: match the requested shape exactly — indices versus values, sorted or not, a string versus an array of strings.

---

## Templates worth having memorised

```
// Frequency count
const freq = new Map();
for (const x of items) freq.set(x, (freq.get(x) ?? 0) + 1);

// Two pointers on a sorted array
let lo = 0, hi = arr.length - 1;
while (lo < hi) { const s = arr[lo] + arr[hi]; if (s === target) …; else if (s

// Sliding window
for (let right = 0, left = 0; right < s.length; right++) {
  add(s[right]);
  while (windowInvalid()) remove(s[left++]);
  best = Math.max(best, right - left + 1);
}

// BFS with an index-pointer queue
const queue = [start]; const seen = new Set([key(start)]);
for (let i = 0; i < queue.length; i++) { const cur = queue[i]; for (const nex

// Memoised recursion
const memo = new Map();
const f = (n) => { if (n < 2) return n; if (memo.has(n)) return memo.get(n); c
```

Each is a shape you fill in, not a solution; the value is that you never spend interview minutes remembering how a sliding window shrinks.

---

## When stuck

Say so, and say what you know: "I can do this in O(n²); I suspect a hash map removes the inner loop but I'm not sure what to key it on." Then try the standard moves: sort the input and see what becomes easy; walk a small example by hand and watch what you compute repeatedly; think of the problem in reverse; consider what the answer looks like for n = 1 and how it grows. Interviewers give hints to candidates who expose their reasoning; they cannot help a silent one.

---

## Testing without a runner

You will usually not have a test framework. Write a tiny table at the bottom — `[[input, expected], …]` — and a loop that prints `ok` / `FAIL` with the actual value. Include the edge

cases you asked about: empty, single element, all equal, negative, the maximum size you discussed. If you can run code, run it; if you cannot, trace the table by hand. Either way the interviewer sees that testing is part of your definition of done.

---

## Common mistakes

- Coding before restating; solving the problem you assumed rather than the one posed.
- Skipping the brute force — then having no fallback when the clever idea stalls.
- Typing in silence; abandoning a working approach for an unfinished better one with five minutes left.
- `sort()` without a comparator; `shift()` queues; `Array(n).fill([])`; string indexing where code points matter.
- Declaring done without tracing an example; forgetting to state complexity.

---

## What the interviewer is listening for

- Whether you separate understanding from solving, and check both.
- Whether you can name the brute force and *why* the better solution is better.
- Whether you know the language: the right structure, the right comparator, the right iteration.
- Whether you test your own code before being asked.
- Whether you communicate under uncertainty — thinking aloud, asking for hints well.

---

## Key takeaways

- Restate → probe → plan (brute force first) → code aloud → trace → analyse.
- Decide before asked: `Map` / `Set`, numeric comparators, index-pointer queues, code-point iteration, `BigInt` for big sums, `Array.from` grids.
- Keep the five template shapes ready: frequency count, two pointers, sliding window, BFS, memoised recursion.
- When stuck, expose your reasoning and try the standard moves; a working brute force beats a broken optimum.
- Test with a small table of cases including every edge you asked about.

EXERCISE 1

## Two-sum, three ways

Line 1 is a list of integers, line 2 the target. Solve two-sum three times and count the work each way does: brute force over every pair ( `pairs checked` ), sort with the original indices attached and walk two pointers ( `steps` ), and a single pass with a `Map` ( `lookups` ). Print one line per approach — `<name>: <counter>=<n> → indices <i>,<j> (<a>+<b>) | <complexity>` or `→ none` — with the complexities `O(n^2) time, O(1) space` , `O(n log n) time, O(n) space for the index pairs` and `O(n) time, O(n) space` . Finish with `all three found a pair summing to <target>` or `no pair sums to <target>: all three agree` .

Example: `2 7 11 15` / `9` →

```
brute force: pairs checked=1 → indices 0,1 (2+7) | O(n^2) time, O(1) space
two pointers: sort, then steps=3 → indices 0,1 (2+7) | O(n log n) time, O(n) space
hash map: lookups=2 → indices 0,1 (2+7) | O(n) time, O(n) space
all three found a pair summing to 9
```

INPUT 1

```
2 7 11 15
9
```

OUTPUT 1

```
brute force: pairs checked=1 → indices
0,1 (2+7) | O(n^2) time, O(1) space
two pointers: sort, then steps=3 →
indices 0,1 (2+7) | O(n log n) time,
O(n) space for the index pairs
hash map: lookups=2 → indices 0,1 (2+7)
| O(n) time, O(n) space
all three found a pair summing to 9
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const nums = lines[0].trim().split(/\s+/).map(Number);
const target = Number(lines[1]);
const show = (pair) => (pair ? `indices ${pair.join(",")} (${nums[pair[0]]}+${nums[p
let checks = 0, brute = null;
outer: for (let i = 0; i < nums.length; i++) for (let j = i + 1; j < nums.length; j+
console.log(`brute force: pairs checked=${checks} → ${show(brute)} | O(n^2) time, O
const order = nums.map((v, i) => [v, i]).sort((a, b) => a[0] - b[0]);       // keep
let lo = 0, hi = order.length - 1, steps = 0, twoPointers = null;
while (lo < hi) { steps++; const s = order[lo][0] + order[hi][0]; if (s === target)
console.log(`two pointers: sort, then steps=${steps} → ${show(twoPointers)} | O(n l
const seen = new Map();
let lookups = 0, hash = null;
for (let i = 0; i < nums.length && !hash; i++) { lookups++; const need = target - nu
console.log(`hash map: lookups=${lookups} → ${show(hash)} | O(n) time, O(n) space`)
const sums = [brute, twoPointers, hash].map((p) => (p ? nums[p[0]] + nums[p[1]] ===
console.log(sums.every((s) => s === true) ? `all three found a pair summing to ${tar
```

EXERCISE 2

## A test table for isAnagram

Write `isAnagram(a, b)` the way you would defend it in a round: NFC-normalise, lower-case, drop everything that is not a letter or digit (Unicode-aware), compare the sorted **code points**. Then run the test table from stdin — each line `<a> | <b> | <expected>` — printing `ok isAnagram(<a JSON>, <b JSON>) → <actual>` or `FAIL … → <actual> (table expected <expected>)`, and finally `<n> passed, <m> failed`. The table includes the edge cases you would ask about: punctuation and spaces, empty strings, a precomposed versus a decomposed `é`, emoji, and one row whose expectation is wrong so that a FAIL is visible.

Example rows: `Dormitory | dirty room! | true` → `ok isAnagram("Dormitory", "dirty room!") → true`; `rat | car | true` → `FAIL isAnagram("rat", "car") → false (table expected true)`.

INPUT 1

```
Listen | Silent | true
Dormitory | dirty room! | true
Astronomer | Moon starer | true
abc | ab | false
 | | true
école | école | true
👍😀 | 😀👍 | true
rat | car | true
```

OUTPUT 1

```
ok   isAnagram("Listen", "Silent") → true
ok   isAnagram("Dormitory", "dirty room!") → true
ok   isAnagram("Astronomer", "Moon starer") → true
ok   isAnagram("abc", "ab") → false
ok   isAnagram("", "") → true
ok   isAnagram("école", "école") → true
ok   isAnagram("👍😀", "😀👍") → true
FAIL isAnagram("rat", "car") → false
(table expected true)
7 passed, 1 failed
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Each line: `<a> | <b> | <expected>` — a test table for isAnagram, which must ignc
const key = (s) => [...s.normalize("NFC").toLowerCase().replace(/[^\p{L}\p{N}]/gu, '
const isAnagram = (a, b) => key(a) === key(b);
let pass = 0, fail = 0;
for (const line of lines) {
  const [a, b, expected] = line.split("|").map((p) => p.trim());
  const actual = isAnagram(a, b);
  const ok = String(actual) === expected;
  ok ? pass++ : fail++;
  console.log(`${ok ? "ok  " : "FAIL"} isAnagram(${JSON.stringify(a)}, ${JSON.string
}
console.log(`${pass} passed, ${fail} failed`);
```

---

QUIZ

**1.** The first thing to do after hearing a problem is…

A. Start coding the brute force

B. Restate it in your own words including the return type, then probe edge cases with examples

C. Ask for the optimal complexity

D. Write the tests

**2.** `[10, 9, 1].sort()` returns…

A. `[1, 9, 10]`

B. `[1, 10, 9]` — the default comparator sorts as strings; pass `(a, b) ⇒ a - b`

C. `[10, 9, 1]`

D. A `TypeError`

**3.** Why prefer a `Map` over a plain object for counting in an interview?

A. It is faster to type

B. Keys keep their type, there is no prototype to collide with, `size` is free and insertion order is kept

C. Objects cannot hold numbers

D. `Map` is immutable

**4.** `Array(3).fill([])` as a grid…

A. Makes three independent rows

B. Shares one array between all three rows — use `Array.from({ length: 3 }, () ⇒ [])`

C. Throws

D. Makes a typed array

**5.** You have a working O(n²) solution and see a possible O(n) one with five minutes left. You should…

A. Delete the working one and start over

B. Keep the working solution, state the improvement you see and its complexity, and only implement it if there is time

C. Say nothing

D. Ask for more time

ANSWER KEY · THE TEMPLATE — RUNNING A JAVASCRIPT CODING ROUND FROM YOUR SIDE OF THE TABLE

**1. B** Restate → probe → plan (brute force first) → code aloud → trace → analyse.

**2. B** One of the decisions to make before the interviewer has to ask.

**3. B** `Set` for membership, `Map` for lookup — say it before you are asked.

**4. B** `fill` writes the same reference into every slot.

**5. B** A working brute force beats a broken optimum; naming the better approach still earns credit.

LESSON 2 · 14 MIN

# The patterns — the dozen algorithm shapes behind most JavaScript rounds

Coding questions are not infinite. Behind the thousands of published problems sit a dozen patterns, and recognising which one a problem belongs to is most of solving it. This lesson names the patterns, gives the tell-tale signs that identify each, the JavaScript shape of its solution, and the mistakes specific to writing it in this language. Depth on any one of them is a data-structures course; the goal here is the recognition step — "this is a sliding window" — that turns a blank page into a template to fill.

---

## 1. Hash map counting

**Signs**: "how many times", "first unique", "anagram", "two numbers that sum to", "group by". **Shape**: one pass building a `Map` (value → count, value → index, key → group), a second pass reading it. O(n) time, O(n) space. **JS notes**: `Map` keeps key types; `freq.get(x) ?? 0`; anagram keys are the sorted characters (`[...w].sort().join("")`) or a 26-count array joined — say which alphabet you assume.

---

## 2. Two pointers

**Signs**: a sorted array, "pair with sum", "remove duplicates in place", "reverse", "palindrome", "merge two sorted". **Shape**: `lo` / `hi` moving inward, or a slow/fast pair moving the same way (`write` / `read` for in-place compaction). O(n), O(1). **JS notes**: destructuring swap `[a[i], a[j]] = [a[j], a[i]]`; for palindromes over Unicode use `[...s]` first.

---

## 3. Sliding window

**Signs**: "longest/shortest substring or subarray with…", "at most k distinct", "maximum sum of size k". **Shape**: `right` expands, a `while` loop moves `left` until the window is valid again, best is recorded each step; state is a count `Map` or a running sum. O(n). **JS notes**: delete a key when its count drops to 0 so `map.size` is the number of distinct elements; fixed-size windows add `s[right]` and remove `s[right - k]`.

## 4. Stack

**Signs**: "valid parentheses", "nearest greater/smaller element", "evaluate an expression", "simplify a path", "undo", nesting of any kind. **Shape**: an array with `push` / `pop`; for **monotonic stacks** pop while the top violates the order and record the popped element's answer. O(n). **JS notes**: `stack.at(-1)` for peek; an empty-stack pop returns `undefined`, so check `length` first.

---

## 5. Queue and BFS

**Signs**: "shortest path in an unweighted graph/grid", "minimum number of steps", "level order", "nearest". **Shape**: a queue seeded with the start, a `seen` set, expand level by level; the first time you reach the target is the shortest. O(V + E). **JS notes**: index-pointer queue (`for (let i = 0; i < queue.length; i++)`) — never `shift()`; grid keys as `` `${r},${c}` `` or `r * cols + c`; the four directions as `[[1,0],[-1,0],[0,1],[0,-1]]`.

---

## 6. DFS, recursion and backtracking

**Signs**: "all combinations/permutations/subsets", "number of islands", "path exists", "generate parentheses", "N-queens", tree traversals. **Shape**: a recursive function that chooses, recurses, and **undoes** the choice (`path.push(x); go(); path.pop()`). Exponential in the output size for enumeration problems — say so. **JS notes**: recursion depth ~$10^4$ — an iterative stack for deep graphs; copy the path when recording an answer (`result.push([...path])`), not the live array; mark visited cells before recursing and unmark after if paths must be re-explorable.

---

## 7. Binary search

**Signs**: a sorted array, "minimum x such that", "first/last position", "square root", "peak", any monotonic predicate over a range. **Shape**: `lo` / `hi` bounds, `mid = lo + ((hi - lo) >> 1)`, shrink toward the side where the predicate flips. O(log n). **JS notes**: `>> 1` is fine below $2^{31}$; use `Math.floor((lo + hi) / 2)` otherwise; get the boundary convention right (half-open `[lo, hi)` and return `lo`) and trace it on a two-element array.

## 8. Prefix sums and difference arrays

**Signs**: "sum of a range many times", "subarray sum equals k", "count of subarrays with…". **Shape**: `prefix[i + 1] = prefix[i] + a[i]`, range sum = `prefix[r + 1] - prefix[l]`; for "subarray sum equals k" combine with a hash map of prefix counts. O(n). **JS notes**: sums beyond $2^{53}$ need `BigInt`; the map is keyed by number, so `0` must be seeded (`map.set(0, 1)`).

## 9. Sorting and greedy

**Signs**: "intervals" (merge, overlap, minimum rooms), "schedule", "assign", "minimum number of…" where a local choice is provably safe. **Shape**: sort by the right key (start, end, or ratio), then one linear pass making the greedy choice. O(n log n). **JS notes**: `sort((a, b) ⇒ a[0] - b[0])`; multi-key `a[0] - b[0] || a[1] - b[1]`; `sort` is stable since 2019, so a previous order survives ties.

## 10. Heaps and top-k

**Signs**: "k largest/smallest", "k most frequent", "merge k sorted", "median of a stream". **Shape**: a min-heap of size k; JavaScript has no built-in heap — implement a small binary heap (push: append and sift up; pop: swap root with last, sift down) or, for one-shot top-k, sort and slice with an O(n log n) note. **JS notes**: interviewers accept `sort().slice(0, k)` for small inputs when you say what the heap would change; a heap class is ~30 lines and worth having practised once.

## 11. Dynamic programming

**Signs**: "number of ways", "minimum cost", "longest … subsequence", "can it be partitioned", overlapping subproblems and optimal substructure. **Shape**: write the recursive relation first, memoise it (a `Map` or an array), then optionally convert to a bottom-up table and compress rows. **JS notes**: `Array(n + 1).fill(0)` for 1-D tables, `Array.from({ length: m + 1 }, () ⇒ Array(n + 1).fill(0))` for 2-D; memo keys for two parameters as `` `${i},${j}` `` or `i * (n + 1) + j`; the recursive version may exceed the stack for n ≈ $10^4$ — mention the iterative conversion.

## 12. Linked lists, trees and graphs as objects

**Signs**: anything about nodes. **Shape**: `{ val, next }`, `{ val, left, right }`, adjacency lists as `Map<node, node[]>` or arrays of arrays. Common moves: dummy head for list edits, slow/fast pointers for the middle or a cycle, recursion mirroring tree structure, `seen` sets for graphs. **JS notes**: build adjacency with `(adj.get(u) ?? adj.set(u, []).get(u)).push(v)`; a tree's height recursion is fine, a list's recursion is not (lists are deep).

## Complexity you should know cold

| Structure / operation | Cost |
|---|---|
| Array index, `push` / `pop` | O(1) |
| Array `shift` / `unshift` / `splice` / `includes` / `indexOf` | O(n) |
| `Map` / `Set` get/set/has/delete | O(1) average |
| `sort` | O(n log n) |
| String concatenation in a loop | fine (ropes), but build arrays for clarity |
| Binary search | O(log n) |
| BFS/DFS | O(V + E) |
| Enumerating subsets / permutations | $O(2^n)$ / O(n!) |

State the complexity of the *whole* solution — the sort you did before the linear pass counts.

## Common mistakes

- Reaching for DFS when BFS is asked for (shortest path needs BFS on unweighted graphs).
- Sliding windows that never shrink; monotonic stacks that pop the wrong way; binary searches with an off-by-one boundary.
- Recording the live `path` array instead of a copy in backtracking.
- `shift()` queues, `includes` inside loops, recomputing what a prefix sum would give.

Not naming the pattern — the interviewer cannot tell whether you recognised the problem or guessed.

## What the interviewer is listening for

The recognition step: "this looks like a sliding window because…".

Correct complexity for the whole solution, including sorting and the data structures.

Handling the pattern's standard edge cases: empty input, k larger than n, a single node, disconnected graphs.

Knowing which JavaScript structure implements the pattern well — and that a heap has to be written.

## Key takeaways

Twelve patterns cover most rounds: counting, two pointers, sliding window, stack, BFS, DFS/backtracking, binary search, prefix sums, sort + greedy, heaps/top-k, DP, node structures.

Recognise by the signs, then fill the template; say the pattern's name and complexity out loud.

JavaScript specifics: `Map` / `Set`, index-pointer queues, `at(-1)` peeks, copies of paths, `Array.from` tables, `BigInt` for big sums, no built-in heap.

Know the complexity table cold, and count the sort.

EXERCISE 1

## Sliding windows and two pointers

Commands, one per line. `longest-unique <s>` — the longest substring without repeating characters (code points), printed `longest-unique <s>: length=<n> substring=<first such substring>`. `at-most-k <k> <s>` — the longest substring with at most k distinct characters. `max-window <k> <nums…>` — the maximum sum over windows of size k: `max-window k=<k>: sum=<s> window=[a,b,c] at <start>`, or `k larger than n=<n>`. `pair-sorted <target> <sorted nums…>` — two pointers from both ends: `pair-sorted target=<t>: <a>+<b> steps=<n>` or `none steps=<n>`.

Example: `longest-unique pwwkew` → `longest-unique pwwkew: length=3 substring=wke`; `at-most-k 2 eceba` → `at-most-k k=2 eceba: length=3 substring=ece`; `max-window 3 2 1 5 1 3 2`

→ `max-window k=3: sum=9 window=[5,1,3] at 2`; `pair-sorted 9 1 2 4 5 7 11` → `pair-sorted target=9: 2+7 steps=3`.

INPUT 1

```
longest-unique abcabcbb
longest-unique pwwkew
longest-unique 😀a😀
at-most-k 2 eceba
max-window 3 2 1 5 1 3 2
max-window 9 1 2
pair-sorted 9 1 2 4 5 7 11
pair-sorted 100 1 2 3
```

OUTPUT 1

```
longest-unique abcabcbb: length=3
substring=abc
longest-unique pwwkew: length=3
substring=wke
longest-unique 😀a😀: length=2
substring=😀a
at-most-k k=2 eceba: length=3
substring=ece
max-window k=3: sum=9 window=[5,1,3] at
2
max-window k=9: k larger than n=2
pair-sorted target=9: 2+7 steps=3
pair-sorted target=100: none steps=2
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// longest-unique <s> | at-most-k <k> <s> | max-window <k> <nums…> | pair-sorted <ta
function longestUnique(chars) {
  const last = new Map();
  let best = [0, 0];
  for (let right = 0, left = 0; right < chars.length; right++) {
    if (last.has(chars[right]) && last.get(chars[right]) >= left) left = last.get(ch
    last.set(chars[right], right);
    if (right - left + 1 > best[1] - best[0]) best = [left, right + 1];
  }
  return best;
}
function atMostK(chars, k) {
  const count = new Map();
  let best = [0, 0];
  for (let right = 0, left = 0; right < chars.length; right++) {
    count.set(chars[right], (count.get(chars[right]) ?? 0) + 1);
    while (count.size > k) { const c = chars[left++]; count.set(c, count.get(c) - 1)
    if (right - left + 1 > best[1] - best[0]) best = [left, right + 1];
  }
  return best;
}
for (const line of lines) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  if (cmd === "longest-unique") { const chars = [...rest[0]]; const [l, r] = longest
  else if (cmd === "at-most-k") { const k = Number(rest[0]), chars = [...rest[1]]; c
  else if (cmd === "max-window") {
    const k = Number(rest[0]), nums = rest.slice(1).map(Number);
    if (k > nums.length) { console.log(`max-window k=${k}: k larger than n=${nums.le
    let sum = 0, best = -Infinity, at = 0;
    for (let i = 0; i < nums.length; i++) { sum += nums[i]; if (i >= k) sum -= nums[
    console.log(`max-window k=${k}: sum=${best} window=[${nums.slice(at, at + k).joi
  } else if (cmd === "pair-sorted") {
    const target = Number(rest[0]), nums = rest.slice(1).map(Number);
    let lo = 0, hi = nums.length - 1, steps = 0, found = null;
    while (lo < hi) { steps++; const s = nums[lo] + nums[hi]; if (s === target) { fo
    console.log(`pair-sorted target=${target}: ${found ? `${found[0]}+${found[1]}` :
  }
}
```

EXERCISE 2

## Stacks and breadth-first search

Commands. `brackets <s>` — validate `()[]{}` with a stack: `brackets "<s>": valid`, `invalid at <i> (unexpected '<c>')` or `invalid at <i> (unclosed '<c>')`. `next-greater <nums…>` — the

next greater element for each position via a monotonic stack, `-1` when none: `next-greater [2,1,2,4,3] → [4,2,4,-1,-1]`. `simplify-path <path>` — collapse `.`, `..` and repeated slashes with a stack: `simplify-path /a/./b/../../c/ → /c`. `bfs <rows>` followed by that many grid lines (`S` start, `E` end, `#` wall, `.` open) — the shortest path in four directions with an index-pointer queue: `bfs <cols>x<rows>: shortest=<steps> visited=<n>` or `unreachable visited=<n>`.

Example: the 4×4 grid `S.#.` / `.##.` / `....` / `#..E` → `bfs 4x4: shortest=6 visited=11`.

INPUT 1

```
brackets ([]{})
brackets ([)]
brackets ((
brackets ())
next-greater 2 1 2 4 3
simplify-path /a/./b/../../c/
simplify-path /../x//y/
bfs 4
S.#.
.##.
....
#..E
bfs 2
S#
#E
```

OUTPUT 1

```
brackets "([]{})": valid
brackets "([)]": invalid at 2
(unexpected ')')
brackets "((": invalid at 1 (unclosed
'(')
brackets "())": invalid at 2 (unexpected
')')
next-greater [2,1,2,4,3] →
[4,2,4,-1,-1]
simplify-path /a/./b/../../c/ → /c
simplify-path /../x//y/ → /x/y
bfs 4x4: shortest=6 visited=11
bfs 2x2: unreachable visited=1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// brackets <s> | next-greater <nums…> | simplify-path <path> | bfs <rows> followed
const PAIRS = { ")": "(", "]": "[", "}": "{" };
function brackets(s) {
  const stack = [];
  for (let i = 0; i < s.length; i++) {
    const c = s[i];
    if ("([{".includes(c)) stack.push([c, i]);
    else if (c in PAIRS) { if (stack.length === 0 || stack.at(-1)[0] !== PAIRS[c]) r
  }
  return stack.length ? `invalid at ${stack.at(-1)[1]} (unclosed '${stack.at(-1)[0]}
}
function nextGreater(nums) {
  const out = new Array(nums.length).fill(-1), stack = [];                  // stack
  for (let i = 0; i < nums.length; i++) { while (stack.length && nums[stack.at(-1)]
  return out;
}
function simplifyPath(p) {
  const stack = [];
  for (const part of p.split("/")) { if (part === "" || part === ".") continue; if (
  return "/" + stack.join("/");
}
function bfs(grid) {
  const rows = grid.length, cols = grid[0].length;
  let start, end;
  grid.forEach((row, r) => [...row].forEach((c, col) => { if (c === "S") start = [r,
  const queue = [[...start, 0]], seen = new Set([`${start[0]},${start[1]}`]);
  for (let i = 0; i < queue.length; i++) {                                    // inc
    const [r, c, d] = queue[i];
    if (r === end[0] && c === end[1]) return `shortest=${d} visited=${seen.size}`;
    for (const [dr, dc] of [[1, 0], [-1, 0], [0, 1], [0, -1]]) {
      const nr = r + dr, nc = c + dc, key = `${nr},${nc}`;
      if (nr < 0 || nc < 0 || nr >= rows || nc >= cols || grid[nr][nc] === "#" || se
      seen.add(key); queue.push([nr, nc, d + 1]);
    }
  }
  return `unreachable visited=${seen.size}`;
}
for (let i = 0; i < lines.length; i++) {
  const [cmd, ...rest] = lines[i].trim().split(/\s+/);
  if (cmd === "brackets") console.log(`brackets "${rest[0] ?? ""}": ${brackets(rest[
  else if (cmd === "next-greater") console.log(`next-greater [${rest.join(",")}] →
  else if (cmd === "simplify-path") console.log(`simplify-path ${rest[0]} → ${simpl
  else if (cmd === "bfs") { const n = Number(rest[0]); const grid = lines.slice(i +
}
```

QUIZ

**1.** "Shortest path in an unweighted grid" is a sign for…

A. DFS

B. BFS — level-by-level expansion reaches the target first along a shortest path

C. Binary search

D. Dynamic programming

**2.** In a sliding window, when the window becomes invalid you…

A. Restart from `right + 1`

B. Move `left` forward in a `while` loop until it is valid again, then record the best

C. Move `right` backward

D. Reset the counts

**3.** In backtracking, recording an answer with `result.push(path)` …

A. Is correct

B. Stores the live array that later `pop`s mutate — push a copy: `result.push([...path])`

C. Is faster

D. Throws

**4.** JavaScript's built-in priority queue is…

A. `PriorityQueue`

B. Nonexistent — write a ~30-line binary heap, or sort and slice for a one-shot top-k while saying what the heap would change

C. `Array.prototype.heapify`

D. `Map`

**5.** A solution that sorts and then does one linear pass is…

A. O(n)

B. O(n log n) — the sort counts; state the complexity of the whole solution

C. O(log n)

D. O(n²)

ANSWER KEY · THE PATTERNS — THE DOZEN ALGORITHM SHAPES BEHIND MOST JAVASCRIPT ROUNDS

**1. B** DFS finds *a* path; BFS finds the shortest one on unweighted graphs.

**2. B** Each index enters and leaves once, so the whole pass is O(n).

**3. B** Choose, recurse, undo — and copy what you keep.

**4. B** Interviewers accept the sort for small inputs when you name the trade-off.

**5. B** Same for building a Map before a loop: the total, not the last step.

LESSON 3 · 11 MIN

# The idiom sheet — the one-liners worth knowing by heart, and the traps inside them

Every language has a set of small moves that experienced practitioners write without thinking, and an interviewer notices when you have them: they free your attention for the problem, and they signal fluency more reliably than any theory answer. This is the JavaScript sheet — collection idioms, string idioms, object idioms, number idioms — each with the trap that catches people who half-remember it. Read it once for recognition, then let the exercises make them yours.

## Collections

```
const freq = new Map(); for (const x of xs) freq.set(x, (freq.get(x) ?? 0) + 1
const groups = new Map(); for (const x of xs) (groups.get(key(x)) ?? groups.se
const unique = [...new Set(xs)];                                    // dedupe p
const byKey = [...new Map(xs.map((x) => [x.id, x])).values()];      // dedupe
const sorted = [...xs].sort((a, b) => a.age - b.age || a.name.localeCompare(b.
const top = [...freq].sort((a, b) => b[1] - a[1] || a[0].localeCompare(b[0]))
const chunks = Array.from({ length: Math.ceil(xs.length / n) }, (_, i) => xs.s
const zipped = a.map((x, i) => [x, b[i]]);                          // zip (le
const range = Array.from({ length: n }, (_, i) => start + i);
const flat = nested.flat(Infinity);
const [min, max] = xs.reduce(([lo, hi], x) => [Math.min(lo, x), Math.max(hi, x
const sum = xs.reduce((a, b) => a + b, 0);                          // always
const last = xs.at(-1);
const shuffled = xs.map((x) => [Math.random(), x]).sort((a, b) => a[0] - b[0])
```

Traps: `sort()` without a comparator sorts numbers as strings and **mutates**; `Math.max(...big)` throws `RangeError` on argument-count limits; `reduce` on an empty array with no initial value throws; `Array(3).map(f)` does nothing (holes are skipped — use `Array.from({ length: 3 }, f)`); `new Set` deduplicates by SameValueZero, so two equal-looking objects stay.

## Strings

```
const chars = [...s];                                                    // code po
const reversed = [...s].reverse().join("");
const isPalindrome = (s) => { const t = s.toLowerCase().replace(/[^\p{L}\p{N}]
const words = s.trim().split(/\s+/).filter(Boolean);
const title = words.map((w) => w[0].toUpperCase() + w.slice(1).toLowerCase())
const camel = kebab.replace(/-(\w)/g, (_, c) => c.toUpperCase());
const kebab = camel.replace(/[A-Z]/g, (c) => "-" + c.toLowerCase());
const padded = String(n).padStart(3, "0");
const count = (s.match(/a/g) ?? []).length;                              // or [...
const truncate = (s, n) => ([...s].length <= n ? s : [...s].slice(0, n - 1).jo
const rle = s.replace(/(.)\1*/gs, (run, c) => run.length + c);           // run-len
const template = `${name} has ${count} item${count === 1 ? "" : "s"}`;
```

Traps: `s.length` counts UTF-16 units (`"👍".length === 2`); `s[0]` on an empty string is `undefined`, not an error; `split("")` splits surrogate pairs; `replace` with a string pattern replaces the first occurrence only (`replaceAll` or a `/g` regex); `"b" + "a" + +"a" + "a"` is `"baNaNa"` — unary plus on a non-numeric string is `NaN`.

---

## Objects

```
const { a, b: renamed = 2, ...rest } = obj;                              // pick,
const picked = Object.fromEntries(Object.entries(obj).filter(([k]) => keys.inc
const inverted = Object.fromEntries(Object.entries(obj).map(([k, v]) => [v, k]
const merged = { ...defaults, ...overrides };                            // shallo
const shallow = { ...obj }, deep = structuredClone(obj);                 // Node 1
const has = Object.hasOwn(obj, key);                                     // not `h
const isEmpty = Object.keys(obj).length === 0;
const byId = Object.fromEntries(list.map((x) => [x.id, x]));
const frozen = Object.freeze({ ...obj });                                // shallo
const safeGet = obj?.a?.b ?? fallback;
```

Traps: object keys are strings or symbols — `obj[1]` and `obj["1"]` are the same key, and integer-like keys enumerate first in ascending order; spread copies own enumerable properties only (no prototype, getters become values); `JSON.parse(JSON.stringify(x))` drops `undefined`, functions and `Map` / `Set`, turns `Date` into a string and `NaN` into `null`, and throws on cycles; `Object.freeze` is shallow.

---

## Numbers

```
const isInt = Number.isInteger(x);                                    // not x
const rounded = Math.round(x * 100) / 100;                            // toFixe
const clamp = (x, lo, hi) => Math.min(hi, Math.max(lo, x));
const randInt = (lo, hi) => lo + Math.floor(Math.random() * (hi - lo + 1));
const safe = Number.isSafeInteger(n);                                 // beyond
const parsed = Number(str);                                           // strict
const digits = [...String(n)].map(Number);
const mod = ((a % m) + m) % m;                                        // % keep
const gcd = (a, b) => (b ? gcd(b, a % b) : a);
```

Traps: `0.1 + 0.2 !== 0.3` (compare with `Math.abs(a - b) < Number.EPSILON` or work in integers); `parseInt` with `map` passes the index as the radix (`["1","2","3"].map(parseInt)` → `[1, NaN, NaN]`); `typeof NaN === "number"` and `NaN !== NaN` — use `Number.isNaN`; `-7 % 3 === -1`; `~x` / `x | 0` truncate to 32 bits.

---

## Functions

```
const once = (fn) => { let done = false, v; return (...a) => (done ? v : ((dor
const memo = (fn) => { const c = new Map(); return (x) => (c.has(x) ? c.get(x)
const sleep = (ms) => new Promise((r) => setTimeout(r, ms));
const debounce = (fn, ms) => { let t; return (...a) => { clearTimeout(t); t =
const pipe = (...fns) => (x) => fns.reduce((v, f) => f(v), x);
const tryCatch = (fn, fallback) => { try { return fn(); } catch { return fallb
```

Traps: arrow functions have no `this` / `arguments` of their own and cannot be constructors; a memoiser keyed by a single argument breaks for several (join them, or nest Maps); `debounce` returning a new function per render means the timer is never shared.

---

## Common mistakes

`sort()` without a comparator; `Math.max(...huge)`; `Array(n).map`; `reduce` with no seed.

`s.length` / `s[i]` for characters; `split("")` on emoji; `replace` without `/g`.

`key in obj` for own properties; JSON round trips as a deep clone of anything with Dates or `undefined`.

`parseInt` as a `map` callback; `x % 1 === 0`; `toFixed` treated as rounding a number.

Memoising with a single-key Map for multi-argument functions.

## What the interviewer is listening for

Whether the small moves are automatic — the comparator appears without a pause, the queue is a pointer, the string is spread before it is reversed.

Whether you know *why* the idiom is shaped that way (the trap it avoids) — that is the difference between fluency and copy-paste.

Whether you reach for `Map` / `Set` and `Object.fromEntries` / `entries` transformations rather than manual loops for everything.

## Key takeaways

Collections: `Map` counting and grouping, `Set` dedupe, comparator sorts on copies, `Array.from` for chunks/ranges/tables, `reduce` with a seed.

Strings: spread for code points, regex with `/g` and `/u`, `padStart`, run-length via backreference.

Objects: destructuring with rename/default/rest, `entries` / `fromEntries` transforms, `Object.hasOwn`, shallow spread versus a real deep clone.

Numbers: `Number.isInteger` / `isNaN` / `isSafeInteger`, epsilon comparison, sign-safe modulo, `Number()` over `parseInt`.

Know the trap behind each idiom; it is what the interviewer will ask about.

EXERCISE 1

## The collection idioms as a command interpreter

Line 1 lists records `name:dept:age`. Then commands: `count <field>` → `count dept: eng=2 ops=2 hr=1` (a `Map`, insertion order); `group <field>` → `group dept: eng=[ann,bob] …`; `unique <field>` → `unique age: 30,25,41` (a `Set`); `sort <f1> <f2>` → a **copy** sorted by the first field (numeric or `localeCompare`) then the second: `sort age,name: cid(25) ann(30) …`; `chunk <n>` → `chunk 2: [ann,bob] [cid,dee] [eve]`; `top <k> <field>` → the k most frequent values, ties broken alphabetically; `range <a> <b>` → `range 3..7: 3,4,5,6,7`; `zip <f1> <f2>` → `zip name,age: ann=30 …`; `minmax <field>` → via one `reduce` (never `Math.max(...huge)`): `minmax age: 25..41`; `dedupe-by <field>` → the last record per value through a `Map`: `dedupe-by dept: bob(eng) dee(ops) eve(hr)`.

INPUT 1

```
ann:eng:30 bob:eng:30 cid:ops:25
dee:ops:41 eve:hr:30
count dept
group dept
unique age
sort age name
chunk 2
top 2 dept
range 3 7
zip name age
minmax age
dedupe-by dept
```

OUTPUT 1

```
count dept: eng=2 ops=2 hr=1
group dept: eng=[ann,bob] ops=[cid,dee]
hr=[eve]
unique age: 30,25,41
sort age,name: cid(25) ann(30) bob(30)
eve(30) dee(41)
chunk 2: [ann,bob] [cid,dee] [eve]
top 2 dept: eng=2 ops=2
range 3..7: 3,4,5,6,7
zip name,age: ann=30 bob=30 cid=25
dee=41 eve=30
minmax age: 25..41
dedupe-by dept: bob(eng) dee(ops)
eve(hr)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Line 1: records `name:dept:age`; then commands: count <field> | group <field> | u
const records = lines[0].trim().split(/\s+/).map((r) => { const [name, dept, age] =
const countBy = (f) => { const m = new Map(); for (const r of records) m.set(r[f], (
const groupBy = (f) => { const m = new Map(); for (const r of records) (m.get(r[f])
const cmp = (f) => (a, b) => (typeof a[f] === "number" ? a[f] - b[f] : a[f].localeCo
for (const line of lines.slice(1)) {
  const [cmd, ...args] = line.trim().split(/\s+/);
  if (cmd === "count") console.log(`count ${args[0]}: ${[...countBy(args[0])].map(([
  else if (cmd === "group") console.log(`group ${args[0]}: ${[...groupBy(args[0])].m
  else if (cmd === "unique") console.log(`unique ${args[0]}: ${[...new Set(records.m
  else if (cmd === "sort") { const sorted = [...records].sort((a, b) => cmp(args[0])
  else if (cmd === "chunk") { const n = Number(args[0]); const chunks = Array.from({
  else if (cmd === "top") { const k = Number(args[0]); const top = [...countBy(args[
  else if (cmd === "range") { const [a, b] = args.map(Number); console.log(`range ${
  else if (cmd === "zip") console.log(`zip ${args[0]},${args[1]}: ${records.map((r)
  else if (cmd === "minmax") { const [lo, hi] = records.reduce(([lo, hi], r) => [Mat
  else if (cmd === "dedupe-by") { const last = [...new Map(records.map((r) => [r[arg
}
```

EXERCISE 2

## The string idioms as a command interpreter

Commands: `reverse <s>` by code points ( `reverse: 👍cba` for `abc👍` ); `palindrome <text…>` ignoring case and anything that is not a letter or digit; `camel <kebab-case>` ; `kebab <camelCase>` ; `title <text…>` ; `truncate <n> <text…>` to n code points with a trailing `…` counting as one; `rle`

`<s>` run-length encoding via a backreference regex ( `3a1b2c4d` ); `count <char> <text…>` ; `pad <width> <n>` with `padStart` ; `words <text…>` → `words: <count> longest=<word>` .

Example: `palindrome A man, a plan, a canal: Panama` → `palindrome: true` ; `truncate 5 hello world` → `truncate 5: hell…` ; `rle 👍👍👍x` → `rle: 3👍1x` .

INPUT 1

```
reverse abc👍
palindrome A man, a plan, a canal:
Panama
palindrome not one
camel background-color-hover
kebab backgroundColorHover
title hELLO wORLD from js
truncate 5 hello world
truncate 20 short
rle aaabccdddd
count a banana bread
pad 5 42
words the quick brown fox jumps
```

OUTPUT 1

```
reverse: 👍cba
palindrome: true
palindrome: false
camel: backgroundColorHover
kebab: background-color-hover
title: Hello World From Js
truncate 5: hell…
truncate 20: short
rle: 3a1b2c4d
count a: 4
pad: 00042
words: 5 longest=quick
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// reverse <s> | palindrome <text…> | camel <kebab-case> | kebab <camelCase> | title
for (const line of lines) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  const text = rest.join(" ");
  if (cmd === "reverse") console.log(`reverse: ${[...rest[0]].reverse().join("")}`);
  else if (cmd === "palindrome") { const t = text.toLowerCase().replace(/[^\p{L}\p{N
  else if (cmd === "camel") console.log(`camel: ${rest[0].replace(/-(\w)/g, (_, c) =
  else if (cmd === "kebab") console.log(`kebab: ${rest[0].replace(/[A-Z]/g, (c) => '
  else if (cmd === "title") console.log(`title: ${text.split(/\s+/).map((w) => w[0].
  else if (cmd === "truncate") { const n = Number(rest[0]), s = rest.slice(1).join('
  else if (cmd === "rle") console.log(`rle: ${rest[0].replace(/(.)\1*/gsu, (run, c)
  else if (cmd === "count") { const s = rest.slice(1).join(" "); console.log(`count
  else if (cmd === "pad") console.log(`pad: ${String(rest[1]).padStart(Number(rest[0
  else if (cmd === "words") { const words = text.trim().split(/\s+/).filter(Boolean)
}
```

QUIZ

**1.** `Math.max(...arr)` on an array of 200 000 numbers…

A. Returns the maximum

B. Can throw a `RangeError` (too many arguments) — use a `reduce`

C. Returns `NaN`

D. Is O(log n)

**2.** `["1", "2", "3"].map(parseInt)` gives…

A. `[1, 2, 3]`

B. `[1, NaN, NaN]` — `map` passes the index as `parseInt`'s radix; use `Number` or `(s) ⇒ parseInt(s, 10)`

C. `["1", "2", "3"]`

D. A `TypeError`

**3.** `[...s].reverse().join("")` rather than `s.split("").reverse().join("")` because…

A. It is shorter

B. Spread iterates code points, so surrogate pairs (emoji) stay intact; `split("")` splits them

C. `split` is deprecated

D. It is faster

**4.** `JSON.parse(JSON.stringify(obj))` as a deep clone…

A. Is exact

B. Loses `undefined`, functions, `Map` / `Set`, turns `Date` into a string and `NaN` into `null`, and throws on cycles

C. Keeps prototypes

D. Is the fastest possible

**5.** `[].reduce((a, b) ⇒ a + b)` …

A. Returns `0`

B. Throws a `TypeError` — no seed and no elements; always pass an initial value

C. Returns `undefined`

D. Returns `[]`

## ANSWER KEY · THE IDIOM SHEET — THE ONE-LINERS WORTH KNOWING BY HEART, AND THE TRAPS INSIDE THEM

**1. B** Spread into a call is bounded by the argument limit.

**2. B** Every idiom on the sheet has a trap like this one.

**3. B** `"👍".length` is 2; the string is UTF-16 underneath.

**4. B** `structuredClone` or a recursive clone with a seen-map is the real answer.

**5. B** The seed also fixes the accumulator's type.

LESSON 4 · 14 MIN

# Implement the built-in — the reimplementation questions and the details that decide them

"Implement `Promise.all`." "Write `bind`." "How would you implement `Array.prototype.reduce`?" These questions test whether you understand the semantics of things you use every day — not whether you can recite them. The trap is that the obvious ten-line version is wrong in ways the interviewer knows about: `bind` must work with `new`, `map` must skip holes, `reduce` must throw on an empty array without a seed, `Promise.all` must preserve order and reject on the first failure, `JSON.stringify` must drop `undefined` in objects but write `null` in arrays. This lesson lists the questions that recur, the spec details each one hinges on, and the shape of a correct answer; the exercises and the checkpoint have you write them.

---

## The recurring list

| Ask | The detail that decides it |
| --- | --- |
| `call` / `apply` / `bind` | Set `this` via a temporary symbol-keyed property (or `Reflect.apply`); `bind` pre-applies arguments and **is ignored as `this` when the bound function is called with `new`** — detect via `this instanceof bound` or `new.target` |
| `new` | Create an object whose prototype is `Ctor.prototype`, call `Ctor` with it as `this`, return the object **unless the constructor returned an object** |
| `instanceof` | Walk `Object.getPrototypeOf` from the value until `null`, comparing to `Ctor.prototype`; primitives are `false`; honour `Symbol.hasInstance` if asked |
| `Object.create` | An object with the given prototype (`null` allowed) plus optional descriptors |
| `Array.prototype.map/filter/forEach` | Iterate `0…length-1`, **skip holes** (`i in arr`), pass `(value, index, array)`, honour `thisArg`; `map` preserves length and holes |
| `reduce` | With no initial value start from the first *present* element; **throw `TypeError` on an empty array without a seed** |
| `flat(depth)` | Recursive with `depth - 1`, `Infinity` allowed, holes removed, `depth` default 1; `flatMap` is `map` then `flat(1)` |
| `includes` versus `indexOf` | `includes` uses SameValueZero (finds `NaN`); `indexOf` uses `===` (never finds `NaN`) |
| `Promise.all` | Result array **in input order**, resolve when the counter hits `length`, reject on the **first** rejection, non-promise inputs are wrapped (`Promise.resolve`), an empty input resolves `[]` immediately |
| `Promise.allSettled` / `race` / `any` | `allSettled` never rejects — `{status, value |
| A `Promise` class | State machine (pending → fulfilled/rejected, once); `then` returns a **new** promise; handlers run **asynchronously** (microtask) even when already settled; a handler's return value resolves the next promise, a thrown error rejects it, a returned thenable is **adopted** |

| Ask | The detail that decides it |
| --- | --- |
| `JSON.stringify` | `undefined`/functions/symbols are **omitted** as object values and become `null` in arrays; `NaN` / `Infinity` → `null`; `toJSON` honoured (Dates); strings escaped; cycles throw `TypeError`; replacer function/array and indent |
| `debounce` / `throttle` | Shared timer across calls; trailing versus leading; `cancel`; preserve `this` and arguments |
| `EventEmitter` | `on` / `off` / `once` / `emit`; copy the listener list before emitting so `off` during emit is safe |
| LRU cache | `Map` insertion order: delete + re-set on access; evict `map.keys().next().value` |
| `deepClone` | Primitives, arrays, plain objects, `Date`, `RegExp`, `Map`, `Set`; a `WeakMap` of seen objects for **cycles and shared references**; class instances lose their prototype unless you `Object.create(Object.getPrototypeOf(x))` |
| `deepEqual` | Same type; `NaN` equals `NaN`; arrays by length and elements; objects by own key sets; `Date` by time; `Map` / `Set` by entries; cycles via a visited pair set if asked |
| `curry` / `memoize` / `once` / `compose` | Arity via `fn.length`; memo keys for multiple arguments; `once` caches the first result |

## Worked shape: `bind`

```
Function.prototype.myBind = function (thisArg, ...preset) {
  const target = this;
  function bound(...args) {
    // Called with `new`? Then `this` is the fresh instance and thisArg is igr
    const useNew = this instanceof bound;
    return target.apply(useNew ? this : thisArg, [...preset, ...args]);
  }
  bound.prototype = Object.create(target.prototype ?? null);   // so instanceo
  return bound;
};
```

Interviewers probe exactly the `new` case and the partial-application case; mention both before they ask.

---

## Worked shape: `Promise.all`

```
function all(iterable) {
  return new Promise((resolve, reject) => {
    const items = [...iterable];
    const results = new Array(items.length);
    let pending = items.length;
    if (pending === 0) return resolve(results);
    items.forEach((item, i) => {
      Promise.resolve(item).then((value) => {
        results[i] = value;                    // index, not push order
        if (--pending === 0) resolve(results);
      }, reject);                              // first rejection wins; late
    });
  });
}
```

The two things people get wrong: pushing in completion order (breaks input order) and forgetting the empty case (never resolves).

---

## Worked shape: a `Promise`

```
class MyPromise {
  #state = "pending"; #value; #handlers = [];
  constructor(executor) {
    const settle = (state) => (value) => { if (this.#state !== "pending") retu
    try { executor(settle("fulfilled"), settle("rejected")); } catch (e) { set
  }
  then(onOk, onErr) {
    return new MyPromise((resolve, reject) => {
      const run = () => queueMicrotask(() => {
        const handler = this.#state === "fulfilled" ? onOk : onErr;
        if (typeof handler !== "function") return this.#state === "fulfilled"
        try { const out = handler(this.#value); out && typeof out.then === "fu
        catch (e) { reject(e); }
      });
      this.#state === "pending" ? this.#handlers.push(run) : run();
    });
  }
  catch(onErr) { return this.then(undefined, onErr); }
}
```

The points that score: settling is one-shot; `then` returns a new promise; handlers run in a microtask even when the promise is already settled; a missing handler passes the value/reason through; returned thenables are adopted. `resolve(thenable)` inside the executor should adopt too — mention it if you skip it.

---

## Worked shape: `reduce` with holes and the empty case

```
Array.prototype.myReduce = function (fn, ...seed) {
  let i = 0, acc;
  if (seed.length) acc = seed[0];
  else { while (i < this.length && !(i in this)) i++; if (i >= this.length) th
  for (; i < this.length; i++) if (i in this) acc = fn(acc, this[i], i, this);
  return acc;
};
```

Using rest for the seed distinguishes "no seed" from "seed is `undefined`" — the arity check the spec makes with `arguments.length`.

---

## How to answer these live

Say the contract first — what the built-in guarantees — in three or four bullets, including the edge case you know the interviewer is thinking of.

Write the simple path.

Add the edge cases one at a time, naming each ("now the `new` case").

Test with two calls: the normal one and the edge one.

If you do not know a detail, say what you would check ("I believe `map` skips holes; I'd confirm on MDN"). Guessing confidently and wrong is the worst outcome.

---

## Common mistakes

`bind` that ignores `new`; `new` that ignores an object returned by the constructor.

`map` / `filter` on `arr.length` without `i in arr`; `reduce` seeded with `undefined` instead of checking arity.

`Promise.all` pushing in completion order or never resolving on `[]`; `then` that runs handlers synchronously.

`JSON.stringify` writing `undefined` or leaving `NaN`; forgetting `toJSON`.

`deepClone` without a seen-map (infinite recursion on cycles); `deepEqual` that fails on `NaN` or compares `Date`s by identity.

---

## What the interviewer is listening for

Whether you state the contract before coding, and whether it includes the famous edge case.

Whether the asynchronous ones stay asynchronous and one-shot.

Whether the array ones respect holes, `thisArg` and the callback signature.

Whether you test the edge case unprompted.

---

## Key takeaways

The recurring reimplementation questions each hinge on one or two spec details — know the detail, not just the sketch.

`bind` and `new`; `map` and holes; `reduce` and the empty array; `Promise.all` and order/empty; `then` and microtasks/adoption; `JSON.stringify` and

`undefined` / `NaN` / `toJSON` /cycles.

Answer in the order contract → simple path → edge cases named one by one → two tests.

Say what you would verify rather than guessing a detail.

---

EXERCISE 1

## bind, call, apply, new and instanceof by hand

Implement `Function.prototype.myCall` / `myApply` (bind `this` through a temporary symbol-keyed method), `myBind` (pre-applied arguments **and** the `new` case — a bound function called with `new` ignores the bound `this` and keeps the prototype chain), `myNew(Ctor, …args)` (an object returned by the constructor replaces `this`) and `myInstanceOf(value, Ctor)` (walk the prototype chain; primitives and `null` are `false`). The starter's checks compare each against the native behaviour; stdin lists check names or `all`. Print `ok <check>: <detail> match=true` per check and `<n>/<total> match the native behaviour`.

Example (`all`) →

```
ok call: hi, Ada! match=true
ok apply: 9 match=true
ok bind-partial: add(10)(5)=15 match=true
ok bind-this: n=2 match=true
ok bind-new: {x:1,y:2} instanceof Point=true sum=3 match=true
ok new-plain: {x:1,y:2} sum=3 match=true
ok new-returns-object: {"custom":true} match=true
ok instanceof: dog→Dog,Animal,Object,Array | [],5,null,nullProto→Object = true,tru
8/8 match the native behaviour
```

INPUT 1

```
all
```

OUTPUT 1

```
ok call: hi, Ada! match=true
ok apply: 9 match=true
ok bind-partial: add(10)(5)=15 match=true
ok bind-this: n=2 match=true
ok bind-new: {x:1,y:2} instanceof
Point=true sum=3 match=true
ok new-plain: {x:1,y:2} sum=3 match=true
ok new-returns-object: {"custom":true}
match=true
ok instanceof: dog→Dog,Animal,Object,Array
| [],5,null,nullProto→Object =
true,true,true,false,true,false,false,false
match=true
8/8 match the native behaviour
```

## REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Implement myCall, myApply, myBind (partial application AND `new` support), myNew
// Input: check names, or `all`. Checks: call apply bind-partial bind-this bind-new
Function.prototype.myCall = function (thisArg, ...args) {
  const ctx = thisArg == null ? globalThis : Object(thisArg);                // pri
  const key = Symbol("fn");
  ctx[key] = this;
  try { return ctx[key](...args); } finally { delete ctx[key]; }             // a m
};
Function.prototype.myApply = function (thisArg, args = []) { return this.myCall(this
Function.prototype.myBind = function (thisArg, ...preset) {
  const target = this;
  function bound(...args) { return target.myApply(this instanceof bound ? this : thi
  bound.prototype = Object.create(target.prototype ?? null);
  return bound;
};
function myNew(Ctor, ...args) {
  const obj = Object.create(Ctor.prototype);
  const out = Ctor.apply(obj, args);
  return out !== null && (typeof out === "object" || typeof out === "function") ? ou
}
function myInstanceOf(value, Ctor) {
  if (value === null || (typeof value !== "object" && typeof value !== "function"))
  for (let p = Object.getPrototypeOf(value); p !== null; p = Object.getPrototypeOf(p
  return false;
}
function greet(greeting, punct) { return `${greeting}, ${this.name}${punct}`; }
function add(a, b) { return a + b; }
function Point(x, y) { this.x = x; this.y = y; }
Point.prototype.sum = function () { return this.x + this.y; };
function Weird() { this.ignored = true; return { custom: true }; }
class Animal {} class Dog extends Animal {}
const checks = {
  call() { const mine = greet.myCall({ name: "Ada" }, "hi", "!"), native = greet.cal
  apply() { const mine = Math.max.myApply(null, [3, 9, 2]), native = Math.max.apply(
  "bind-partial"() { const mine = add.myBind(null, 10)(5), native = add.bind(null, 1
  "bind-this"() { const counter = { n: 0, inc() { return ++this.n; } }; const f = co
  "bind-new"() { const Bound = Point.myBind({ ignored: true }, 1); const p = new Bou
  "new-plain"() { const p = myNew(Point, 1, 2), q = new Point(1, 2); return [`{x:${p
  "new-returns-object"() { const w = myNew(Weird), n = new Weird(); return [`${JSON.
  instanceof() { const d = new Dog(); const pairs = [[d, Dog], [d, Animal], [d, Obje
    const mine = pairs.map(([v, C]) => myInstanceOf(v, C)), native = pairs.map(([v,
    return [`dog→Dog,Animal,Object,Array | [],5,null,nullProto→Object = ${mine.joi
};
const wanted = lines.map((l) => l.trim()).includes("all") ? Object.keys(checks) : li
let pass = 0;
for (const name of wanted) {
  if (!checks[name]) { console.log(`?? ${name}: unknown check`); continue; }
```

```
  const [detail, ok] = checks[name]();
  if (ok) pass++;
  console.log(`${ok ? "ok" : "FAIL"} ${name}: ${detail} match=${ok}`);
}
console.log(`${pass}/${wanted.length} match the native behaviour`);
```

EXERCISE 2

## The array methods from scratch

Line 1 lists elements — numbers, `NaN`, words, JSON arrays like `[3,[4,[5]]]` and `_` for a **hole**. Implement `myForEach`, `myMap`, `myFilter`, `myReduce`, `myFlat(depth = 1)`, `myFlatMap`, `myIncludes` and `myIndexOf` on `Array.prototype` with the spec details: holes are skipped by the callbacks (`map` keeps them as holes), `thisArg` is honoured, `reduce` without a seed starts from the first present element and throws `TypeError` on an empty array, `includes` uses SameValueZero (finds `NaN`, reads a hole as `undefined`) while `indexOf` uses `===` and skips holes. The starter's `show` renders holes as `<hole>`; print each comparison as `<label>: <mine> match=<bool>`.

Example: `1 _ 2 NaN hello [3,[4,[5]]] _` → `map x2: [2,<hole>,4,NaN,"hello",[3,[4,[5]]],<hole>] match=true`, `flat(Infinity): [1,2,NaN,"hello",3,4,5] match=true`, `includes(NaN): true match=true`, `indexOf(NaN): -1 match=true`, `includes(undefined) (a hole reads as undefined): true match=true`.

INPUT 1

```
1 _ 2 NaN hello [3,[4,[5]]] _
```

OUTPUT 1

```
forEach visits (holes skipped): 5
match=true
map x2: [2,<hole>,4,NaN,"hello",[3,[4,
[5]]],<hole>] match=true
filter numbers: [1,2] match=true
reduce sum: 3 match=true
reduce on [] without a seed: TypeError
match=true
reduce on [] with seed: 0 match=true
flat(): [1,2,NaN,"hello",3,[4,[5]]]
match=true
flat(Infinity): [1,2,NaN,"hello",3,4,5]
match=true
flatMap [x,x]:
[1,1,2,2,NaN,NaN,"hello","hello",3,[4,
[5]]] match=true
includes(NaN): true match=true
indexOf(NaN): -1 match=true
includes(undefined) (a hole reads as
undefined): true match=true
indexOf(undefined) (holes skipped): -1
match=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Line 1: elements — numbers, NaN, words, JSON arrays like [1,[2]] and `_` for a ho
const tokens = lines[0].trim().split(/\s+/);
const arr = [];
tokens.forEach((t, i) => { if (t !== "_") arr[i] = t === "NaN" ? NaN : t.startsWith(
arr.length = tokens.length;
const show = (a) => `[${Array.from({ length: a.length }, (_, i) => (i in a ? (typeof
Array.prototype.myForEach = function (fn, thisArg) { for (let i = 0; i < this.length
Array.prototype.myMap = function (fn, thisArg) { const out = new Array(this.length);
Array.prototype.myFilter = function (fn, thisArg) { const out = []; for (let i = 0;
Array.prototype.myReduce = function (fn, ...seed) {
  let i = 0, acc;
  if (seed.length) acc = seed[0];                                          // ar
  else { while (i < this.length && !(i in this)) i++; if (i >= this.length) throw ne
  for (; i < this.length; i++) if (i in this) acc = fn(acc, this[i], i, this);
  return acc;
};
Array.prototype.myFlat = function (depth = 1) { const out = []; for (let i = 0; i <
Array.prototype.myFlatMap = function (fn, thisArg) { return this.myMap(fn, thisArg).
Array.prototype.myIncludes = function (x) { for (let i = 0; i < this.length; i++) if
Array.prototype.myIndexOf = function (x) { for (let i = 0; i < this.length; i++) if
const isNum = (x) => typeof x === "number" && !Number.isNaN(x);
const compare = (label, mine, native) => console.log(`${label}: ${mine} match=${mine
let visitsMine = 0, visitsNative = 0;
arr.myForEach(() => visitsMine++); arr.forEach(() => visitsNative++);
compare(`forEach visits (holes skipped)`, String(visitsMine), String(visitsNative));
compare(`map x2`, show(arr.myMap((x) => (isNum(x) ? x * 2 : x))), show(arr.map((x) =
compare(`filter numbers`, show(arr.myFilter(isNum)), show(arr.filter(isNum)));
compare(`reduce sum`, String(arr.myReduce((a, x) => a + (isNum(x) ? x : 0), 0)), Str
const empty = (fn) => { try { return String(fn()); } catch (e) { return e.name; } };
compare(`reduce on [] without a seed`, empty(() => [].myReduce((a, b) => a + b)), em
compare(`reduce on [] with seed`, empty(() => [].myReduce((a, b) => a + b, 0)), empt
compare(`flat()`, show(arr.myFlat()), show(arr.flat()));
compare(`flat(Infinity)`, show(arr.myFlat(Infinity)), show(arr.flat(Infinity)));
compare(`flatMap [x,x]`, show(arr.myFlatMap((x) => (Array.isArray(x) ? x : [x, x]))
compare(`includes(NaN)`, String(arr.myIncludes(NaN)), String(arr.includes(NaN)));
compare(`indexOf(NaN)`, String(arr.myIndexOf(NaN)), String(arr.indexOf(NaN)));
compare(`includes(undefined) (a hole reads as undefined)`, String(arr.myIncludes(unc
compare(`indexOf(undefined) (holes skipped)`, String(arr.myIndexOf(undefined)), Stri
```

---

QUIZ

**1.** A hand-written `bind` is most often wrong about…

A. Argument order

B. The `new` case — a bound function called with `new` must ignore the bound `this` and keep the prototype chain

C. Return values

D. Arrow functions

**2.** `Promise.all` implemented by pushing results as they arrive…

A. Is correct

B. Breaks the contract — results must be in **input** order; write `results[i] = value` and count settled promises

C. Is faster

D. Rejects on the first success

**3.** A `then` handler registered on an already-settled promise runs…

A. Synchronously, right away

B. Asynchronously in a microtask — always, so ordering is predictable

C. On the next timer tick

D. Never

**4.** `JSON.stringify({ a: undefined, b: [undefined] })` …

A. `{"a":undefined,"b":[undefined]}`

B. `{"b":[null]}` — `undefined` is omitted from objects but becomes `null` in arrays

C. Throws

D. `{"a":null,"b":[null]}`

**5.** `[1, , 3].map(f)` calls `f` …

A. Three times

B. Twice — holes are skipped and preserved in the result; `includes(undefined)` on that array is nonetheless `true`

C. Once

D. Zero times

### ANSWER KEY · IMPLEMENT THE BUILT-IN — THE REIMPLEMENTATION QUESTIONS AND THE DETAILS THAT DECIDE THEM

**1. B** Detect it with `this instanceof bound` (or `new.target`) inside the wrapper.

**2. B** Also: an empty input must resolve `[]` immediately, and the first rejection wins.

**3. B** Handlers also run in a fresh promise's context: their return value resolves the next promise, a thrown error rejects it.

**4. B** Functions and symbols behave the same; `NaN` / `Infinity` become `null`; cycles throw.

**5. B** `i in arr` is the hole test a reimplementation needs.

LESSON 5 · 14 MIN

# Theory drill — the questions, the two-sentence answers, and where each one goes deeper

The theory part of a JavaScript interview is a fixed repertoire: perhaps forty questions, asked in slightly different words, for which the interviewer wants a crisp answer and then one level of depth. The failure mode is not ignorance — you have covered every one of these in the previous fifteen modules — but a rambling answer that never lands, or a crisp answer that cannot go one level further. This lesson is the repertoire as flashcards: the question, the answer in two sentences, the follow-up the interviewer is likely to add, and the module that holds the depth. Drill it aloud; the exercises give you a spaced-repetition scheduler and an answer grader to practise with.

## Language core

| Question | Two-sentence answer | Likely follow-up → depth |
| --- | --- | --- |
| `var`, `let`, `const`? | `var` is function-scoped and hoisted as `undefined`; `let` / `const` are block-scoped with a temporal dead zone. `const` fixes the binding, not the value. | Loop closures with `var` → module 2 |
| Hoisting? | Declarations are processed before execution: functions fully, `var` as `undefined`, `let` / `const` / `class` uninitialised (TDZ). Accessing a TDZ binding throws `ReferenceError`. | Function versus arrow expressions → module 3 |
| `==` versus `===`? | `===` compares type and value; `==` coerces first by an algorithm (null/undefined equal each other only; numbers win over strings; objects to primitives). Use `===` and `Object.is` for `NaN` / `±0`. | `[] == ![]` → module 2 |
| Coercion and falsy values? | Seven falsy values: `false, 0, -0, 0n, "", null, undefined`; everything else is truthy including `[]` and `{}`. `+` concatenates when either side is a string; other operators go numeric. | `"5" - 2` versus `"5" + 2` → module 2 |
| `typeof null`? | `"object"` — a bug preserved for compatibility. Test with `x === null`. | `typeof` of a function/array → module 2 |
| `null` versus `undefined`? | `undefined` means "no value assigned" (missing property, no return); `null` is a deliberate "empty". Default parameters trigger on `undefined` only. | `??` versus `\|\|` → module 14 |
| Closures? | A function keeps access to the variables of the scope where it was created, even after that scope has returned. They implement private state, memoisation, `once`, and module patterns. | Loop-closure classic; memory leaks via closures → modules 3, 15 |
| `this`? | Decided at call time by the call form: `new` → the instance; `call` / `apply` / `bind` → explicit; | Losing `this` in callbacks and the three |

| Question | Two-sentence answer | Likely follow-up → depth |
| --- | --- | --- |
| | `obj.method()` → `obj`; plain call → `undefined` in strict mode. Arrows have no `this` and use the enclosing one. | fixes → module 3 |
| Prototypes and inheritance? | Every object has a prototype link; property lookup walks the chain. `class` is syntax over constructor functions and prototypes; `extends` links the prototypes and `super` calls the parent. | `Object.create`, `instanceof` implementation → module 5 |
| Shallow versus deep copy? | Spread/ `Object.assign` copy one level; nested objects are shared. Deep copies need `structuredClone` or a recursive clone that handles cycles. | What `JSON.parse(JSON.stringify())` loses → modules 4, 14 |
| Immutability? | Not changing an object after creation — copy-on-write with spread and non-mutating array methods. `Object.freeze` is shallow; `const` does not freeze. | Structural sharing, `Readonly` in TS → modules 11, 13 |
| `Map` versus object? | `Map` takes any key type, preserves insertion order, has `size`, no prototype collisions, and is faster for frequent add/delete. Objects for fixed-shape records and JSON. | `WeakMap` and leaks → modules 6, 15 |
| Symbols? | Unique, non-enumerable-by-default keys used for metadata and protocols (`Symbol.iterator`). `Symbol.for` gives a global registry. | Well-known symbols → module 6 |
| Iterators and generators? | An iterable has `[Symbol.iterator]()` returning `{ next() }`; `for…of`, spread and destructuring use it. Generators (`function*`) build iterators lazily with `yield`. | Async generators and `for await` → modules 6, 8 |
| Modules: ESM versus CommonJS? | ESM is static (`import` / `export`, hoisted, live bindings, async loading); CJS is dynamic (`require`, a copied `module.exports` value at call time). Node picks by `"type"` /extension. | Interop, `__dirname` in ESM → module 9 |

| Question | Two-sentence answer | Likely follow-up → depth |
| --- | --- | --- |
| Strict mode? | Turns silent errors into thrown ones: undeclared assignment, writes to read-only, duplicate parameters; `this` is `undefined` in plain calls. Modules and classes are strict by default. | Module 1 |

## Asynchrony

| Question | Two-sentence answer | Likely follow-up → depth |
| --- | --- | --- |
| The event loop? | One thread runs tasks from queues: the current script, then all microtasks (promise reactions, `queueMicrotask`), then one macrotask (timer, I/O), repeat. Node adds `process.nextTick` before microtasks and phases for timers, I/O, `setImmediate`. | Order of `setTimeout(0)`, `Promise.then`, `nextTick` → modules 1, 8 |
| Microtask versus macrotask? | Microtasks drain completely before the loop continues (starving rendering if unbounded); macrotasks run one per turn. Promise callbacks are microtasks; timers are macrotasks. | Why `await` in a loop yields → module 8 |
| Promises? | An object representing a future value in one of three states, settled once; `then` returns a new promise, chaining transformations and flattening returned promises. Errors propagate to the nearest `catch`. | Implement one → module 16 checkpoint |
| `async` / `await`? | Syntax over promises: an `async` function returns a promise; `await` pauses it (yielding to the loop) until the awaited value settles. Sequential awaits serialise; `Promise.all` runs them concurrently. | `forEach` with `await`; `return await` in `try` → module 8 |
| Callback hell and its fixes? | Deep nesting of callbacks with duplicated error handling. Promises flatten it, `async` / `await` makes it read sequentially, and `util.promisify` converts Node-style APIs. | Module 8 |
| `Promise.all` versus `allSettled` versus `race` versus `any`? | `all` fails fast and preserves order; `allSettled` reports every outcome; `race` takes the first to settle; `any` takes the first to fulfil and fails only when all fail. | Timeouts with `race` and `AbortSignal` → module 8 |
| Debounce versus throttle? | Debounce runs once after activity stops for the delay (search box); throttle runs at most once per interval during activity (scroll). Both share one timer across calls. | Leading/trailing options → module 8 |

## Browser and platform

| Question | Two-sentence answer | Likely follow-up → depth |
|---|---|---|
| Event bubbling, capturing, delegation? | Events run capture (down), target, bubble (up); `addEventListener`'s third argument picks the phase. Delegation registers one listener on an ancestor and dispatches on `event.target.closest(selector)`. | `stopPropagation` versus `preventDefault` → module 12 |
| `localStorage`, `sessionStorage`, cookies? | Storage is 5–10 MB, string-only, synchronous, per origin, never sent to the server; cookies are ~4 KB, sent with every request, and carry `HttpOnly` / `Secure` / `SameSite` flags. Auth tokens belong in `HttpOnly` cookies. | XSS and CSRF → module 12 |
| CORS? | The browser blocks cross-origin *responses* unless the server opts in with `Access-Control-Allow-Origin`; preflight `OPTIONS` requests precede non-simple requests. It protects users, not servers. | Credentials and `SameSite` → module 12 |
| XSS and CSP? | Injecting script through untrusted HTML; prevent with escaping/ `textContent`, sanitisation, and a Content-Security-Policy that forbids inline and foreign scripts. | Module 12 |
| Reflow versus repaint? | Reflow recomputes layout (geometry changes); repaint redraws pixels (colour). Interleaving DOM writes and layout reads forces synchronous reflows — batch reads, then writes. | `requestAnimationFrame` → module 12 |
| Web Workers? | Background threads without DOM access, communicating by message passing (structured clone or transfer). Use for CPU-heavy work to keep the UI responsive. | `SharedArrayBuffer` → module 15 |

## Engineering

| Question | Two-sentence answer | Likely follow-up → depth |
| --- | --- | --- |
| Memory leaks in JavaScript? | References you forgot: unbounded caches, listeners and timers never removed, closures over large scopes, detached DOM nodes, growing arrays. Find with two heap snapshots and retainer paths. | Generational GC → module 15 |
| Why is my code slow? | Measure first (profile, percentiles), then fix complexity and data structures, then allocation and shape stability in the hot path; on servers count round trips and cache composed payloads. | Module 15 |
| Bundling and tree-shaking? | A bundler follows the import graph into deployable files; tree-shaking removes unused ESM exports (static, side-effect-free). Code splitting loads routes lazily. | Source maps, `sideEffects` → module 14 |
| TypeScript's benefits? | Types catch mismatches at compile time, document contracts, and drive tooling; structural typing and narrowing model real JavaScript. Types vanish at run time — validate at boundaries. | `unknown` versus `any`, discriminated unions → module 13 |
| Error handling patterns? | Throw `Error` subclasses with `cause`; catch at boundaries (request handler, `main`); reject promises with errors, never values. Expected failures can be `Result` values. | Unhandled rejections → module 7 |
| Testing? | Unit tests with arrange–act–assert on behaviour, not implementation; mock only the boundary (network, clock); integration tests for the wiring; a pre-commit gate. | Module 14 |

## How to use these

Answer in the two-sentence form first, then stop. Silence invites the follow-up, and the follow-up is where you demonstrate depth — do not pre-empt it with a monologue. When a question is one you have not seen, reduce it to one you have ("that is asking about closures with an event loop twist") and say so.

## Common mistakes

Answering with an example instead of a definition ("closures are like when you have a function inside a function…"). Give the definition, then an example if asked.

Rambling past the answer; contradicting yourself on the way to the point.

Confusing microtasks and macrotasks; `null` and `undefined`; shallow and deep copies; CORS as a server protection.

Not knowing where the depth is when the follow-up comes.

## What the interviewer is listening for

A precise first sentence.

Correct vocabulary: TDZ, microtask, prototype chain, structural typing, retained size.

The ability to go one level deeper on request — and to say "I don't know that one" cleanly when you cannot.

## Key takeaways

The theory repertoire is finite; know each answer in two sentences and where its depth lives.

Core: scoping and hoisting, coercion, closures, `this`, prototypes, copies and immutability, `Map`, symbols, iterators, modules.

Async: the loop, micro/macro tasks, promises, `async` / `await`, the four combinators, debounce/throttle.

Platform and engineering: events, storage, CORS/XSS, reflow, workers, leaks, performance method, bundling, TypeScript, errors, tests.

Answer, stop, take the follow-up.

EXERCISE 1

## A Leitner flashcard scheduler

Line 1 is `cards: a b c …` (every card starts in box 1). Each following line is `session <n>: a=ok b=miss …`. Box 1 is due every session, box 2 every second session, box 3 every fourth; box 4 is mastered and retired (printed `M`). A due card answered `ok` moves up one box; `miss` sends it back

to box 1; a due card with no result is `skipped`; a result for a card that was not due is `ignored`. Print per session `session <n>: due=[…] ok=[…] miss=[…]` (plus `skipped=[…]` / `ignored=[…]` when non-empty) `→ a=2 b=1 …`, then `mastered: … | box3: … | box2: … | box1: …` (`-` for empty) and `next due (session <n+1>): […]`.

Example: four cards, four sessions →

```
session 1: due=[a,b,c,d] ok=[a,b,d] miss=[c] → a=2 b=2 c=1 d=2
session 2: due=[a,b,c,d] ok=[a,c,d] miss=[b] → a=3 b=1 c=2 d=3
session 3: due=[b] ok=[] miss=[] skipped=[b] ignored=[c,a] → a=3 b=1 c=2 d=3
session 4: due=[a,b,c,d] ok=[a,b,c,d] miss=[] → a=M b=2 c=3 d=M
mastered: a,d | box3: c | box2: b | box1: -
next due (session 5): []
```

INPUT 1

```
cards: a b c d
session 1: a=ok b=ok c=miss d=ok
session 2: a=ok b=miss c=ok d=ok
session 3: c=ok a=ok
session 4: a=ok b=ok c=ok d=ok
```

OUTPUT 1

```
session 1: due=[a,b,c,d] ok=[a,b,d]
miss=[c] → a=2 b=2 c=1 d=2
session 2: due=[a,b,c,d] ok=[a,c,d]
miss=[b] → a=3 b=1 c=2 d=3
session 3: due=[b] ok=[] miss=[]
skipped=[b] ignored=[c,a] → a=3 b=1 c=2
d=3
session 4: due=[a,b,c,d] ok=[a,b,c,d]
miss=[] → a=M b=2 c=3 d=M
mastered: a,d | box3: c | box2: b |
box1: -
next due (session 5): []
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Line 1: `cards: a b c …` (all start in box 1). Then `session <n>: a=ok b=miss …`.
// Box 1 is due every session, box 2 every 2nd, box 3 every 4th; box 4 is mastered (
const box = new Map(lines[0].replace(/^cards:\s*/, "").trim().split(/\s+/).map((c) =
const INTERVAL = { 1: 1, 2: 2, 3: 4 };
const dueIn = (session) => [...box].filter(([, b]) => b < 4 && session % INTERVAL[b]
const boxes = () => [...box].map(([c, b]) => `${c}=${b === 4 ? "M" : b}`).join(" ");
let last = 0;
for (const line of lines.slice(1)) {
  const [, n, rest] = line.match(/^session (\d+):\s*(.*)$/);
  const session = Number(n);
  last = session;
  const due = dueIn(session);
  const results = Object.fromEntries((rest.trim() ? rest.trim().split(/\s+/) : []).m
  const ok = [], miss = [], ignored = [], skipped = [];
  for (const card of due) {
    if (results[card] === "ok") { box.set(card, box.get(card) + 1); ok.push(card); }
    else if (results[card] === "miss") { box.set(card, 1); miss.push(card); }
    else skipped.push(card);                                                    // du
  }
  for (const card of Object.keys(results)) if (!due.includes(card)) ignored.push(car
  console.log(`session ${session}: due=[${due.join(",")}] ok=[${ok.join(",")}] miss=
}
const byBox = (b) => [...box].filter(([, v]) => v === b).map(([c]) => c).join(",") |
console.log(`mastered: ${byBox(4)} | box3: ${byBox(3)} | box2: ${byBox(2)} | box1: $
console.log(`next due (session ${last + 1}): [${dueIn(last + 1).join(",")}]`);
```

EXERCISE 2

## An answer grader

Line 1 is `keywords: a, b, c` (the terms a strong answer mentions), line 2 `misconceptions: x, y` (phrases that mark a wrong model); the remaining lines are `<candidate>: <answer>`. Normalise the answer (lower-case, punctuation to spaces, collapsed whitespace, padded with spaces) and match each keyword and misconception as a whole phrase. Score = coverage % minus 25 per flagged misconception; grade `strong` (≥ 80% coverage, nothing flagged), `partial` (≥ 50%), else `weak`, with `(misconception)` appended to partial/weak when something was flagged. Print, sorted by score then name, `<name>: coverage=<hit>/<total> (<pct>%) missing=[…] flagged=[…] words=<n>` (plus `(long)` past 60 words) → `<grade>`, then `best: <name>`.

Example (garbage collection) →

```
ada: coverage=5/6 (83%) missing=[reachable] flagged=[] words=29 → strong
bob: coverage=1/6 (17%) missing=[reachable, roots, scavenge, promoted, mark-sweep] f
cid: coverage=0/6 (0%) missing=[…] flagged=[] words=10 → weak
best: ada
```

"unreachable" does not count as "reachable" — the padded-phrase match is deliberate.

INPUT 1

```
keywords: reachable, roots,
generational, scavenge, promoted, mark-
sweep
misconceptions: reference count, frees
immediately
ada: Objects unreachable from the roots
are garbage. V8 is generational: young
objects are freed by a cheap scavenge,
survivors get promoted to old space,
which is collected by mark-sweep.
bob: The engine keeps a reference count
and frees immediately when it hits zero;
the heap is generational.
cid: It cleans up memory automatically
when you stop using variables.
```

OUTPUT 1

```
ada: coverage=5/6 (83%) missing=
[reachable] flagged=[] words=29 →
strong
bob: coverage=1/6 (17%) missing=
[reachable, roots, scavenge, promoted,
mark-sweep] flagged=[reference count,
frees immediately] words=17 → weak
(misconception)
cid: coverage=0/6 (0%) missing=
[reachable, roots, generational,
scavenge, promoted, mark-sweep] flagged=
[] words=10 → weak
best: ada
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Line 1: `keywords: a, b, c`; line 2: `misconceptions: x, y`; then `<candidate>: <
const list = (line) => line.split(":").slice(1).join(":").split(",").map((s) => s.tr
const keywords = list(lines[0]), misconceptions = list(lines[1]);
const normalise = (s) => ` ${s.toLowerCase().replace(/[^\p{L}\p{N}\s-]/gu, " ").repl
const graded = lines.slice(2).map((line) => {
  const [name, ...rest] = line.split(":");
  const answer = rest.join(":").trim(), text = normalise(answer);
  const hit = keywords.filter((k) => text.includes(` ${k} `)), missing = keywords.fi
  const flagged = misconceptions.filter((m) => text.includes(` ${m} `));
  const words = answer.split(/\s+/).filter(Boolean).length;
  const coverage = Math.round((hit.length / keywords.length) * 100);
  const score = Math.max(0, coverage - 25 * flagged.length);
  const grade = flagged.length ? (coverage >= 50 ? "partial (misconception)" : "weak
  return { name: name.trim(), hit, missing, flagged, words, coverage, score, grade }
}).sort((a, b) => b.score - a.score || a.name.localeCompare(b.name));
for (const g of graded) console.log(`${g.name}: coverage=${g.hit.length}/${keywords.
console.log(`best: ${graded[0]?.name ?? "-"}`);
```

QUIZ

**1.** The two-sentence answer to "what is a closure?" is…

A. "It's like a function inside a function"

B. A function keeps access to the variables of the scope where it was created, even after that scope has returned; this implements private state, memoisation and module patterns

C. A way to copy variables

D. A class

**2.** Microtasks versus macrotasks:

A. Both run one per loop turn

B. Microtasks (promise reactions, `queueMicrotask`) drain completely before the loop continues; one macrotask (timer, I/O) runs per turn

C. Timers are microtasks

D. There is no difference in Node

**3.** CORS protects…

A. The server from attackers

B. Users — the browser blocks cross-origin responses unless the server opts in; it is not authentication

C. The database

D. Cookies from JavaScript

**4.** "`const` makes an object immutable."

A. True

B. False — `const` fixes the binding, not the value; `Object.freeze` is shallow immutability, and deep immutability needs copy-on-write discipline

C. True for arrays only

D. True in strict mode

**5.** When you do not know a theory answer you should…

A. Guess confidently

B. Say so cleanly, reduce it to something you do know if possible, and move on

C. Change the subject

D. Answer a different question

ANSWER KEY · THEORY DRILL — THE QUESTIONS, THE TWO-SENTENCE ANSWERS, AND WHERE EACH ONE GOES DEEPER

**1. B** Definition first, example only if asked; then stop and take the follow-up.

**2. B** Node runs `process.nextTick` callbacks before the microtasks.

**3. B** `HttpOnly` cookies and CSP answer the other two threats.

**4. B** One of the classic confusions the drill exists to remove.

**5. B** Guessing confidently and wrong is the worst outcome; a clean "I don't know that one" is not.

CHECKPOINT · 26 MIN

# Checkpoint — Interview idioms

This checkpoint closes the track. It covers the interview template and its JavaScript-specific decisions; the twelve algorithm patterns and their complexities; the idiom sheet and the trap behind each idiom; the reimplementation questions and the spec details they hinge on; and the theory repertoire in its two-sentence form.

**How it works.** Twelve questions and three programs; 70% on the questions and every program accepted clears the module — and with it the JavaScript study plan.

**Before you start**, make sure you can answer:

- The six steps of the template and the choices to make before being asked (`Map` / `Set`, comparators, index-pointer queues, code points, `BigInt`, `Array.from` grids).
- The signs that identify each pattern, its template shape, and the complexity of the whole solution including any sort.
- The trap behind each idiom on the sheet: `sort()`, `Math.max(...big)`, `Array(n).map`, `reduce` without a seed, `s.length`, `parseInt` in `map`, JSON round trips, `Object.freeze`.
- The detail that decides each reimplementation: `bind` and `new`; `map` and holes; `reduce` and the empty array; `Promise.all` and order; `then` and microtasks/adoption; `JSON.stringify` and `undefined` / `NaN` / `toJSON` /cycles.
- The two-sentence answers to the theory repertoire and where each goes deeper.

The programs are the three reimplementations senior rounds ask for most: a `Promise` class with chaining, adoption, `catch` / `finally` and the four static combinators, driven by a script and printing the real settlement order; a `JSON.stringify` that matches the native one on every listed case, including `toJSON`, replacers, indentation and cycle detection; and a deep-equality and deep-clone toolkit that handles `NaN`, `Date`, `Map`, `Set`, cycles and shared references, reported as a TAP log.

---

EXERCISE 1

## A Promise from scratch

Implement `MyPromise`: one-shot settlement; `then` / `catch` / `finally` returning new promises; handlers always run asynchronously (`queueMicrotask`) even on a settled promise; missing handlers pass the value or reason through; a handler's return value resolves the next promise and a throw

rejects it; thenables are **adopted** both when returned from a handler and when passed to `resolve` (another `MyPromise`, a plain `{ then }` object, or a native `Promise`); an executor that throws rejects. Statics: `resolve`, `reject`, `all` (input order, fail-fast, `[]` resolves at once), `allSettled`, `race` and `any` (`AggregateError` when every input rejects, including an empty input). The starter's scenarios — `chain adopt reject order all allSettled race any`, chosen by stdin — drive it and print `[<scenario>] <message>` lines.

Example (`order`) →

```
[order] sync 1
[order] sync 2 (then already registered)
[order] then on a settled promise
[order] queueMicrotask registered after
```

The `then` on an already-fulfilled promise still ran after the synchronous code — and before a microtask queued later.

INPUT 1

```
chain
adopt
reject
order
```

OUTPUT 1

```
[chain] step 2
[chain] step 20
[chain] fulfilled undefined
[chain] fulfilled "x"
[adopt] fulfilled 201
[adopt] fulfilled "adopted in executor"
[adopt] rejected Error: adopted
rejection
[reject] caught boom
[reject] finally ran
[reject] fulfilled "after recovered"
[reject] fulfilled "caught RangeError"
[reject] rejected Error: executor threw
[reject] rejected "plain value"
[order] sync 1
[order] sync 2 (then already registered)
[order] then on a settled promise
[order] queueMicrotask registered after
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Implement MyPromise: one-shot settlement, then/catch/finally returning new promis
// adoption of thenables (returned from handlers and passed to resolve), and the sta
// Input: scenario names — chain adopt reject order all allSettled race any
class MyPromise {
  #state = "pending"; #value; #handlers = [];
  constructor(executor) {
    const settle = (state, value) => { if (this.#state !== "pending") return; this.#
    const doResolve = (value) => {
      if (value === this) return settle("rejected", new TypeError("Chaining cycle de
      const thenable = value !== null && (typeof value === "object" || typeof value
      if (!thenable) return settle("fulfilled", value);
      let once = false;                                                    // a
      try { value.then((v) => { if (!once) { once = true; doResolve(v); } }, (r) =>
      catch (e) { if (!once) { once = true; settle("rejected", e); } }
    };
    let called = false;
    try { executor((v) => { if (!called) { called = true; doResolve(v); } }, (r) =>
    catch (e) { if (!called) { called = true; settle("rejected", e); } }
  }
  then(onFulfilled, onRejected) {
    return new MyPromise((resolve, reject) => {
      const run = () => queueMicrotask(() => {                             // as
        const handler = this.#state === "fulfilled" ? onFulfilled : onRejected;
        if (typeof handler !== "function") return this.#state === "fulfilled" ? resc
        try { resolve(handler(this.#value)); } catch (e) { reject(e); }      // re
      });
      this.#state === "pending" ? this.#handlers.push(run) : run();
    });
  }
  catch(onRejected) { return this.then(undefined, onRejected); }
  finally(onFinally) { return this.then((v) => MyPromise.resolve(onFinally()).then((
  static resolve(v) { return v instanceof MyPromise ? v : new MyPromise((res) => res
  static reject(r) { return new MyPromise((_, rej) => rej(r)); }
  static all(items) { return new MyPromise((resolve, reject) => { const arr = [...it
  static allSettled(items) { return MyPromise.all([...items].map((item) => MyPromise
  static race(items) { return new MyPromise((resolve, reject) => { for (const item o
  static any(items) { return new MyPromise((resolve, reject) => { const arr = [...it
}
const log = (scenario, msg) => console.log(`[${scenario}] ${msg}`);
const toNative = (p) => new Promise((res) => p.then((v) => res(`fulfilled ${JSON.str
const order = [];
const delay = (ms, value, fail = false) => new MyPromise((res, rej) => setTimeout(()
const scenarios = {
  async chain() {
    log("chain", await toNative(MyPromise.resolve(1).then((v) => v + 1).then((v) =>
    log("chain", await toNative(MyPromise.resolve("x").then(undefined, undefined).th
  },
```

```
  async adopt() {
    log("adopt", await toNative(MyPromise.resolve(1).then((v) => delay(10, v + 1)).t
    log("adopt", await toNative(new MyPromise((res) => res(MyPromise.resolve("adopte
    log("adopt", await toNative(new MyPromise((res) => res(MyPromise.reject(new Erro
  },
  async reject() {
    log("reject", await toNative(new MyPromise((_, rej) => rej(new Error("boom"))).t
    log("reject", await toNative(MyPromise.resolve().then(() => { throw new RangeErr
    log("reject", await toNative(new MyPromise(() => { throw new Error("executor thr
    log("reject", await toNative(MyPromise.reject("plain value").finally(() => {})))
  },
  async order() {
    log("order", "sync 1");
    const p = MyPromise.resolve("x");
    p.then(() => log("order", "then on a settled promise"));
    queueMicrotask(() => log("order", "queueMicrotask registered after"));
    log("order", "sync 2 (then already registered)");
    await toNative(p);
  },
  async all() {
    order.length = 0;
    log("all", `${await toNative(MyPromise.all([delay(30, "a"), delay(10, "b"), dela
    log("all", await toNative(MyPromise.all([1, MyPromise.resolve(2), Promise.resolv
    log("all", await toNative(MyPromise.all([])));
    order.length = 0;
    const out = await toNative(MyPromise.all([delay(60, "slow"), delay(10, "fast-fai
    log("all", `${out} settledSoFar=${order.join(",")} (rejected before the slow tas
  },
  async allSettled() {
    log("allSettled", await toNative(MyPromise.allSettled([delay(20, "a"), delay(10,
    log("allSettled", await toNative(MyPromise.allSettled([])));
  },
  async race() {
    log("race", await toNative(MyPromise.race([delay(30, "slow"), delay(10, "fast",
    log("race", await toNative(MyPromise.race([delay(30, "slow"), "immediate"])));
    MyPromise.race([]).then(() => log("race", "never"));
    log("race", "race([]) stays pending forever");
  },
  async any() {
    log("any", await toNative(MyPromise.any([delay(20, "a", true), delay(10, "b", tr
    log("any", await toNative(MyPromise.any([delay(10, "x", true), delay(20, "y", tr
    log("any", await toNative(MyPromise.any([]).catch((e) => `${e.name} with ${e.err
  },
};
(async () => {
  for (const name of lines.map((l) => l.trim())) {
    if (!scenarios[name]) { console.log(`?? unknown scenario ${name}`); continue; }
    await scenarios[name]();
```

```
  }
})();
```

---

EXERCISE 2

## JSON.stringify from scratch

Write `stringify(value, replacer, indent)` matching the native one. Each input line is `<expression> | <replacer expression or →> | <indent expression or →>` (the starter evaluates them); print `mine=<result> match=<bool>` — or `throws <ErrorName> match=<bool>` when yours throws — comparing against `JSON.stringify`. The details: `toJSON` is applied first (Dates), then a replacer **function** (called with `this` = the holder); an **array** replacer is an allow-list applied in *its* order at every level; `undefined`, functions and symbols are omitted from objects and become `null` in arrays; `NaN` / `±Infinity` become `null`; boxed primitives unwrap; BigInt and cycles throw `TypeError`; strings are escaped (`"`, `\`, control characters as `\b \f \n \r \t` or `\u00XX`); a numeric indent is capped at 10 spaces and a string indent at 10 characters, with `": "` after keys only when indenting; empty containers print `{}` / `[]`.

Example: `{a: 1, b: 2, c: {a: 3, b: 4}} | ["c", "a"] | -` → `mine={"c":{"a":3},"a":1} match=true`; a self-referencing object → `throws TypeError match=true`.

INPUT 1

```
{a: undefined, b: [undefined, () ⇒ 1, Symbol("s")], c: NaN, d: -Infinity, e: "q\"uote\n\ttab", f: null} | - | -
{date: new Date(0), nested: {n: 1, arr: [1, [2, [3]]], empty: {}, none: []}} | - | 2
{a: 1, b: 2, c: {a: 3, b: 4}} | ["c", "a"] | -
{a: 1, b: "x", c: [2, 3]} | (k, v) ⇒ typeof v === "number" ? v * 10 : v | -
(() ⇒ { const o = { name: "loop" }; o.self = o; return o; })() | - | -
{ big: 10n } | - | -
[new Number(3), new String("s"), new Boolean(false), null] | - | -
{ toJSON() { return { replaced: true }; }, ignored: 1 } | - | -
"plain string" | - | -
undefined | - | -
{ b: 1, 2: "two", a: 3, 1: "one", m: new Map([[1, 2]]), s: new Set([1]) } | - | -
{a: [], b: {c: [1]}} | - | "\t"
```

OUTPUT 1

```
mine={"b":[null,null,null],"c":null,"d":null,"e":"q\"uote\n\ttab","f":null}
match=true
mine={
  "date": "1970-01-01T00:00:00.000Z",
  "nested": {
    "n": 1,
    "arr": [
      1,
      [
        2,
        [
          3
        ]
      ]
    ],
    "empty": {},
    "none": []
  }
} match=true
mine={"c":{"a":3},"a":1} match=true
mine={"a":10,"b":"x","c":[20,30]} match=true
throws TypeError match=true
throws TypeError match=true
mine=[3,"s",false,null] match=true
mine={"replaced":true} match=true
mine="plain string" match=true
mine=undefined match=true
mine={"1":"one","2":"two","b":1,"a":3,"m":{},"s":{}} match=true
mine={
    "a": [],
    "b": {
        "c": [
            1
        ]
    }
} match=true
```

## REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const evaluate = (src) => new Function(`return (${src});`)();   // the inputs are tr
// Each line: `<expression> | <replacer expression or -> | <indent expression or ->`
function stringify(value, replacer, indent) {
  const gap = typeof indent === "number" ? " ".repeat(Math.max(0, Math.min(10, inder
  const allow = Array.isArray(replacer) ? [...new Set(replacer.map(String))] : null;
  const fn = typeof replacer === "function" ? replacer : null;
  const ESC = { '"': '\\"', "\\": "\\\\", "\b": "\\b", "\f": "\\f", "\n": "\\n", "\r
  const quote = (s) => '"' + s.replace(/[\"\u0000-\u001f]/g, (c) => ESC[c] ?? "\\u'
  const stack = [];
  const ser = (holder, key, value, depth) => {
    if (value !== null && typeof value === "object" && typeof value.toJSON === "func
    if (fn) value = fn.call(holder, key, value);                                  // t
    if (value === null) return "null";
    if (value instanceof Number || value instanceof String || value instanceof Boole
    switch (typeof value) {
      case "string": return quote(value);
      case "number": return Number.isFinite(value) ? String(value) : "null";
      case "boolean": return String(value);
      case "bigint": throw new TypeError("Do not know how to serialize a BigInt");
      case "undefined": case "function": case "symbol": return undefined;         //
    }
    if (stack.includes(value)) throw new TypeError("Converting circular structure to
    stack.push(value);
    const inner = gap.repeat(depth + 1), outer = gap.repeat(depth);
    let out;
    if (Array.isArray(value)) {
      const parts = value.map((v, i) => ser(value, String(i), v, depth + 1) ?? "null
      out = parts.length === 0 ? "[]" : gap ? `[\n${inner}${parts.join(",\n" + inner
    } else {
      const parts = [];
      for (const k of allow ?? Object.keys(value)) { const s = ser(value, k, value[k
      out = parts.length === 0 ? "{}" : gap ? `{\n${inner}${parts.join(",\n" + inner
    }
    stack.pop();
    return out;
  };
  return ser({ "": value }, "", value, 0);
}
const attempt = (f) => { try { return { out: f() }; } catch (e) { return { err: e.na
for (const line of lines) {
  const [src, replacerSrc = "-", indentSrc = "-"] = line.split("|").map((p) => p.tri
  const value = evaluate(src), replacer = replacerSrc === "-" ? undefined : evaluate
  const mine = attempt(() => stringify(value, replacer, indent)), native = attempt(()
  const same = "err" in mine ? mine.err === native.err : mine.out === native.out;
  console.log(`${"err" in mine ? `throws ${mine.err}` : `mine=${mine.out}`} match=${
}
```

EXERCISE 3

## A deep-equality and deep-clone toolkit

Implement `deepEqual(a, b)` (`Object.is` for primitives so `NaN` equals `NaN` and `0` differs from `-0`; same prototype; `Date` by time; `RegExp` by source and flags; `Map` / `Set` by entries; arrays by elements; objects by own keys; cycles via a seen-map), `deepClone(value)` (primitives, `Date`, `RegExp`, `Map`, `Set`, arrays and objects keeping their prototype; a `WeakMap` of clones so **cycles and shared references** survive), and `flatten` / `unflatten` with dotted keys where arrays and empty objects are leaves. Commands: `equal <expected> ||| <a> ||| <b>`, `clone <expr>` (checks equality, distinctness, a `self` cycle, `x` / `y` shared references and per-key prototypes), `flatten <expr>` (prints the flat JSON and checks the round trip). Output is TAP: `ok <n> - …` / `not ok <n> - …`, then `1..<n>` and `# pass <p> fail <f>`.

Example: `clone (() ⇒ { const shared = { v: 1 }; const o = { x: shared, y: shared, when: new Date(0), tags: new Set([1]) }; o.self = o; return o; })()` → `ok 7 - clone: equal=true distinct=true cycle=kept shared=kept typesKept=true`; `flatten {a: {b: {c: 1}, d: [1, 2]}, e: "x", f: {}}` → `ok 9 - flatten {"a.b.c":1,"a.d":[1,2],"e":"x","f":{}} roundtrip=true`.

INPUT 1

```
equal true ||| {a: [1, {b: NaN}], d: new
Date(0)} ||| {a: [1, {b: NaN}], d: new
Date(0)}
equal false ||| {a: 1} ||| {a: 1, b:
undefined}
equal false ||| [1, 2] ||| [2, 1]
equal true ||| new Map([["k", [1]]]) |||
new Map([["k", [1]]])
equal false ||| {n: 1} |||
Object.assign(Object.create(null), {n:
1})
equal false ||| 0 ||| -0
clone (() ⇒ { const shared = { v: 1 };
const o = { x: shared, y: shared, when:
new Date(0), tags: new Set([1]), m: new
Map([[1, {z: 2}]]), re: /a/g }; o.self =
o; return o; })()
clone [1, [2, [3]]]
flatten {a: {b: {c: 1}, d: [1, 2]}, e:
"x", f: {}}
```

OUTPUT 1

```
ok 1 - equal {a: [1, {b: NaN}], d: new
Date(0)} vs {a: [1, {b: NaN}], d: new
Date(0)} → true
ok 2 - equal {a: 1} vs {a: 1, b:
undefined} → false
ok 3 - equal [1, 2] vs [2, 1] → false
ok 4 - equal new Map([["k", [1]]]) vs
new Map([["k", [1]]]) → true
ok 5 - equal {n: 1} vs
Object.assign(Object.create(null), {n:
1}) → false
ok 6 - equal 0 vs -0 → false
ok 7 - clone: equal=true distinct=true
cycle=kept shared=kept typesKept=true
ok 8 - clone: equal=true distinct=true
cycle=n/a shared=n/a typesKept=true
ok 9 - flatten {"a.b.c":1,"a.d":
[1,2],"e":"x","f":{}} roundtrip=true
1..9
# pass 9 fail 0
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const evaluate = (src) => new Function(`return (${src});`)();   // the inputs are tr
// Commands (TAP output): equal <true|false> ||| <a> ||| <b>   |   clone <expr>   |
function deepEqual(a, b, seen = new Map()) {
  if (Object.is(a, b)) return true;                                          // No
  if (typeof a !== "object" || typeof b !== "object" || a === null || b === null) re
  if (Object.getPrototypeOf(a) !== Object.getPrototypeOf(b)) return false;
  if (seen.get(a) === b) return true;                                        // a
  seen.set(a, b);
  if (a instanceof Date) return a.getTime() === b.getTime();
  if (a instanceof RegExp) return a.source === b.source && a.flags === b.flags;
  if (a instanceof Map) { if (a.size !== b.size) return false; for (const [k, v] of
  if (a instanceof Set) { if (a.size !== b.size) return false; for (const v of a) if
  if (Array.isArray(a)) { if (a.length !== b.length) return false; for (let i = 0; i
  const ka = Object.keys(a), kb = Object.keys(b);
  return ka.length === kb.length && ka.every((k) => Object.prototype.hasOwnProperty.
}
function deepClone(value, seen = new WeakMap()) {
  if (typeof value !== "object" || value === null) return value;
  if (seen.has(value)) return seen.get(value);                               // cy
  if (value instanceof Date) return new Date(value.getTime());
  if (value instanceof RegExp) return new RegExp(value.source, value.flags);
  if (value instanceof Map) { const out = new Map(); seen.set(value, out); for (cons
  if (value instanceof Set) { const out = new Set(); seen.set(value, out); for (cons
  if (Array.isArray(value)) { const out = []; seen.set(value, out); value.forEach((v
  const out = Object.create(Object.getPrototypeOf(value));                  // ke
  seen.set(value, out);
  for (const k of Object.keys(value)) out[k] = deepClone(value[k], seen);
  return out;
}
function flatten(obj, prefix = "", out = {}) {
  for (const [k, v] of Object.entries(obj)) {
    const key = prefix ? `${prefix}.${k}` : k;
    if (v !== null && typeof v === "object" && !Array.isArray(v) && Object.keys(v).l
  }
  return out;
}
function unflatten(flat) {
  const out = {};
  for (const [path, v] of Object.entries(flat)) { const parts = path.split("."); let
  return out;
}
let n = 0, pass = 0;
const tap = (ok, text) => { n++; if (ok) pass++; console.log(`${ok ? "ok" : "not ok"
for (const line of lines) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  const body = rest.join(" ");
  if (cmd === "equal") {
```

```
    const [expected, a, b] = body.split("|||").map((p) => p.trim());
    const got = deepEqual(evaluate(a), evaluate(b));
    tap(String(got) === expected, `equal ${a} vs ${b} → ${got}`);
  } else if (cmd === "clone") {
    const original = evaluate(body), copy = deepClone(original);
    const distinct = copy !== original, equal = deepEqual(copy, original);
    const cycle = original && original.self === original ? (copy.self === copy ? "ke
    const shared = original && original.x !== undefined && original.x === original.y
    const types = Object.keys(original ?? {}).every((k) => Object.getPrototypeOf(cop
    tap(distinct && equal && cycle !== "BROKEN" && shared !== "BROKEN" && types, `cl
  } else if (cmd === "flatten") {
    const value = evaluate(body), flat = flatten(value);
    tap(deepEqual(unflatten(flat), value), `flatten ${JSON.stringify(flat)} roundtri
  }
}
console.log(`1..${n}`);
console.log(`# pass ${pass} fail ${n - pass}`);
```

---

### CHECKPOINT QUESTIONS

**1.** Stating the brute force before the optimal solution…

A. Wastes time

B. Shows you separate understanding from optimisation and gives you a working fallback if the clever idea stalls

C. Is only for junior candidates

D. Lowers your score

**2.** "Longest substring with at most k distinct characters" is…

A. Binary search

B. A sliding window with a count `Map`, shrinking from the left while `map.size > k`

C. Backtracking

D. A heap problem

**3.** Using `arr.shift()` as a BFS queue on $10^6$ nodes is…

A. Fine

B. O(n²) overall — each `shift` moves every remaining element; use an index pointer over the array

C. O(n log n)

D. Faster than a pointer

**4.** `new Set([NaN, NaN, 0, -0]).size` is…

A. 4

B. 2 — `Set` uses SameValueZero: one `NaN`, and `0` equals `-0`

C. 3

D. 1

**5.** `"b" + "a" + +"a" + "a"` …

A. Is `"baaa"`

B. Is `"baNaNa"` — the unary plus turns `"a"` into `NaN`, which concatenates as text

C. Throws

D. Is `"ba"`

**6.** A hand-written `new` must…

A. Always return the created object

B. Create an object with `Ctor.prototype`, call `Ctor` with it as `this`, and return the constructor's result instead only when that result is an object

C. Call `Object.freeze`

D. Copy the prototype

**7.** `Array.prototype.reduce` on a sparse `[, , 5]` with no seed…

A. Starts from `undefined`

B. Starts from `5`, the first *present* element — holes are skipped; an all-holes array throws `TypeError`

C. Throws

D. Starts from `0`

**8.** In a correct `then`, a handler that returns a thenable…

A. Resolves the next promise with the thenable object itself

B. Is adopted — the next promise follows the thenable's settlement

C. Throws

D. Is ignored

**9.** `JSON.stringify(obj, ["c", "a"])`…

A. Keeps all keys

B. Keeps only `c` and `a`, in that order, at every level of the object

C. Throws

D. Sorts the keys

**10.** `deepClone` without a seen-map on an object with `o.self = o`…

A. Works

B. Recurses forever (stack overflow) — register each clone before descending into its children

C. Returns `null`

D. Copies the reference

**11.** Two heap snapshots compared, and the retainer path of a grown class, answer the question…

A. "Why is my code slow?"

B. "How do you find a memory leak?"

C. "What is the event loop?"

D. "How does `this` work?"

**12.** The best response to a theory question you cannot answer is…

A. A confident guess

B. "I don't know that one" — cleanly, optionally reducing it to something you do know

C. Silence

D. Asking to skip the interview

ANSWER KEY · CHECKPOINT — INTERVIEW IDIOMS

**1. B** Name its complexity, then say what repeated work the better approach removes.

**2. B** Delete keys at count 0 so `size` means distinct.

**3. B** The same trap appears in the idiom sheet and the performance module.

**4. B** `includes` uses the same rule; `indexOf` and `===` do not find `NaN`.

**5. B** Coercion questions are traps on the idiom sheet; know the rule, not the meme.

**6. B** Same rule the real `new` follows — a returned primitive is ignored.

**7. B** Arity, not `undefined`-ness, decides whether a seed was given.

**8. B** Adoption is what makes chains flatten instead of nesting.

**9. B** An array replacer is an allow-list; a function replacer is applied after `toJSON`.

**10. B** The same map makes shared references stay shared in the copy.

**11. B** Two sentences, then stop; the follow-up is where the depth goes.

**12. B** Interviewers help candidates who expose their reasoning; they cannot help a bluff.

---

CodeKairo · JavaScript study plan · Module 16: Interview idioms. Solve the exercises online to track your progress and earn XP.