Module 15 of 16

Performance and memory

How V8 runs code — tiers, hidden classes, inline caches, elements kinds; benchmarks that lie and profiles that do not; reachability, the generational collector and the leak catalogue; the habits that survive measurement; server latency at the percentile, round trips, caches, pools and event-loop lag.

Performance work in JavaScript is mostly about not fooling yourself: the engine optimises on assumptions you can keep or break without noticing, a naive benchmark measures the wrong thing, the garbage collector frees only what you stopped referencing, and a server is slow for reasons a CPU profile never shows. This module gives you the model behind each of those — enough to predict, measure and fix rather than guess.

The first lesson follows a function through V8's tiers and explains hidden classes, inline caches and elements kinds — what keeps hot code fast and what deoptimises it. The second is about measuring: monotonic clocks, the six ways a benchmark lies, reading distributions with percentiles, profiling with flame charts, and checking complexity before constants. The third covers memory: reachability, the young and old generations, scavenges and promotion, the eight leak patterns and the heap-snapshot workflow that finds them. The fourth is the short list of techniques that survive measurement — and the folklore that does not. The fifth moves to the server: latency budgets at p95/p99, the round-trip arithmetic of sequential awaits, caching layers and single-flight, pools and backpressure, event-loop lag, load testing to the knee.

Timings differ on every machine, so the exercises model the rules instead: a shape tracker and an elements-kinds tracer, a benchmark statistics tool and a complexity fitter, a reachability collector and a leak detective, a comparison counter and a typed-array lab, a percentile/SLO report and an LRU cache simulator — then a hidden-class and inline-cache simulator, a generational collector with promotion and a remembered set, and a request-trace analyser with per-route percentiles, SLO burn and N+1 suspects.

CONTENTS

1	How V8 runs your code — parsing, tiers, hidden classes and inline caches	13 min
2	Measure first — benchmarks that lie, profiles that do not, and reading the numbers	13 min

3	Memory and garbage collection — generations, reachability, and the leaks that survive it	13 min
4	Writing fast JavaScript — the habits that survive measurement	13 min
5	Server and network performance — latency budgets, percentiles, caching and the event loop under load	13 min
✓	Checkpoint — Performance and memory	24 min

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

LESSON 1 · 13 MIN

How V8 runs your code — parsing, tiers, hidden classes and inline caches

JavaScript is fast because the engine bets on your code being *predictable* — that a property access at a given line will see objects of the same shape every time, that a function will be called with the same argument types, that an array holds only small integers — and compiles specialised machine code for those bets. When the bets hold, JavaScript runs within a small factor of C; when they break, the engine throws the specialised code away and falls back to slow paths. Understanding the model — parse, interpret, profile, optimise, deoptimise — and the two ideas underneath it, **hidden classes** and **inline caches**, is what turns "JavaScript performance" from folklore into a handful of habits. This lesson explains the pipeline as V8 (Chrome, Node, Deno, Edge) implements it; other engines differ in names, not in principles.

The pipeline

Parse — source text → AST. Lazily for functions not yet called (a pre-parser checks syntax only), so startup cost scales with what runs, not with what ships. Code caches (V8's compile cache, Node's `v8.compileCache`) skip this on later runs.

Ignition — the bytecode interpreter. Every function starts here; bytecode is compact and quick to produce.

Profiling — while interpreting, V8 records *feedback*: which shapes reached each property access, which types reached each operator, how often each function runs.

Sparkplug — a fast baseline compiler that turns bytecode into unoptimised machine code without feedback (cheap speed-up, no assumptions).

Maglev — a mid-tier optimising compiler (V8 ≥ 11.7; not in Node 16) using feedback for moderately hot code.

TurboFan — the optimising compiler for hot functions: uses the feedback to generate specialised machine code — inlined property loads, unboxed numbers, inlined small functions, eliminated checks.

Deoptimisation — when an assumption fails at run time (a new shape appears, a string arrives where numbers were expected), the optimised code is abandoned and execution returns to the interpreter with fresh feedback. Repeated deopts for the same function eventually make V8 stop trying.

The practical shape: code that runs once (startup) is interpreted — micro-optimising it is pointless; hot code is compiled with assumptions — keeping those assumptions stable is where performance lives.

Hidden classes (shapes)

Objects in JavaScript are dictionaries in principle, but V8 does not store them that way. Each object points to a **hidden class** (also called a *map* or *shape*) describing its property names and where each value sits in memory; objects built the same way share one hidden class. Adding properties creates a **transition chain**:

```
const a = {};          // shape S0 (empty)
a.x = 1;               // S0 --x--> S1
a.y = 2;               // S1 --y--> S2
const b = { x: 5, y: 6 };   // same S2 — same properties in the same order
const c = { y: 6, x: 5 };   // a DIFFERENT shape: order matters
delete a.x;            // a drops to dictionary mode — slow, generic hash look
```

Consequences: initialise all properties in the constructor (or literal), in the same order, every time; never add properties later or conditionally; never `delete` (set to `undefined` / `null` instead, or use a `Map`); avoid objects whose property *sets* vary by data (use a `Map` for dictionaries). Class instances built by one constructor naturally share a shape — one of the real performance reasons to use classes for many small objects.

Inline caches (ICs)

Every property access site (`obj.x` at a specific line) keeps a small cache of the shapes it has seen and, for each, the memory offset of `x`. States:

Monomorphic — one shape: the load is a shape check plus a fixed-offset read, the fastest possible.

Polymorphic — two to four shapes: a short chain of checks, still fast.

Megamorphic — more than four: the site gives up on caching and uses a slow generic lookup (a global hash table). This is where "the same function got 10× slower" comes from.

A function that handles objects of many shapes (a generic `render(item)` over heterogeneous records, a serialiser) is megamorphic by nature — fine, as long as it is not your hot loop. Hot loops want monomorphic sites: uniform objects, consistent types.

Elements kinds

Arrays also have kinds tracked separately from properties: **PACKED_SMI** (small integers only), **PACKED_DOUBLE** (any numbers), **PACKED_ELEMENTS** (anything), and **HOLEY** variants when the array has gaps. Transitions go one way — SMI → DOUBLE → ELEMENTS, PACKED → HOLEY — and never back. `arr[100] = 1` on a short array creates holes; `new Array(n)` starts holey; `[1, 2, 3.5]` is DOUBLE; `[1, "a"]` is ELEMENTS. Operations on PACKED_SMI arrays are the fastest (no hole checks, no boxing); for numeric work of any size, **typed arrays** (`Float64Array`, `Int32Array`) fix the kind entirely. Build arrays with `push` from empty or with `Array.from`, keep them homogeneous, and never write past the end.

Numbers: Smis and heap numbers

V8 represents integers in a 31-bit range (on 64-bit, "Smi") directly inside the pointer — no allocation; other numbers (fractions, large integers) are boxed "heap numbers". An array or object field that once held a double keeps a double representation. Mixing `1` and `1.5` in the same field forces boxing or a wider representation; `|0` and `>>> 0` coerce to integers in hot numeric code (asm.js style), and typed arrays make the representation explicit.

What TurboFan inlines and eliminates

Small functions called from hot code are inlined — the call disappears — so the "function call overhead" folklore mostly does not apply to small, monomorphic helpers; `arr.map(f)` with a tiny `f` compiles to a loop. Bounds checks, type checks and shape checks are hoisted or removed when feedback proves them redundant. Escape analysis removes allocations of short-lived objects that never leave the function (a `{x, y}` returned and immediately destructured). All of this depends on stable feedback: the same code with megamorphic sites gets none of it.

Things that used to matter and mostly do not

`try` / `catch` in hot functions (optimisable since 2017), `arguments` (fine unless leaked), `for…of` versus indexed `for` (both optimise), closures per se, getters/setters on classes, `const` / `let` in loops. Measure before believing any performance folklore — the engine changes every six weeks.

Things that still matter

Shape stability and monomorphic hot sites.

Elements kinds: packed, homogeneous, no holes; typed arrays for numeric work.

Avoiding `delete`, dictionary-mode objects and megamorphic hot loops.

Allocation rate in hot paths (short-lived objects are cheap but not free — module lesson 3).

Algorithmic complexity — a hash lookup versus a linear scan dwarfs every micro-effect (lesson 4).

Common mistakes

Adding properties conditionally or in varying order; `delete` on hot objects.

Arrays with holes or mixed types in numeric code; `new Array(n)` without filling.

One generic function serving as the hot loop for every record type.

Optimising startup code that runs once; believing folklore instead of profiling.

Reading V8 internals as portable truth — other engines differ in tiers and thresholds.

Interview angle

"How does V8 execute JavaScript?" Parse → Ignition bytecode with feedback → Sparkplug baseline → TurboFan optimised code on assumptions → deoptimise when they fail.

"What is a hidden class?" An object's shape descriptor (property names and offsets); objects built the same way share one; adding properties or `delete` changes it.

"Monomorphic versus megamorphic?" A property site seeing one shape (fast fixed-offset load) versus more than four (slow generic lookup).

"Why is `delete` slow?" It drops the object into dictionary mode, disabling fast fixed-offset property access.

"Why prefer typed arrays for numeric work?" Fixed element kind and representation — no boxing, no hole checks, predictable memory.

Key takeaways

Interpret first, optimise hot code on feedback, deoptimise on surprise; startup code is never optimised.

Shapes: same properties in the same order share a hidden class; no conditional adds, no `delete`; classes/constructors give uniform shapes.

Inline caches: keep hot property sites monomorphic; generic code is fine outside hot loops.

Elements kinds transition one way (SMI → DOUBLE → ELEMENTS, PACKED → HOLEY); typed arrays for numbers.

Most old folklore is obsolete; shape stability, algorithmic complexity and allocation rate are what remain.

EXERCISE 1

A hidden-class tracker

Model V8's transition tree. Each input line is either `<name>: <prop> <prop> …` — an object built by adding those properties in that order — or `site <name> <obj> <obj> …` — a property-access site and the objects it has seen. Every object starts at shape `S0`; adding property `p` to shape `S` follows the transition `S --p⟶` to an existing shape if that transition was taken before, or creates the next `S<n>`. A token `—<prop>` deletes the property: the object drops to its own dictionary shape, printed `DICT(<name>)`. Print each object as `<name>: <shape> (<props comma-joined>)` (`<dictionary>` for a dictionary object), then `distinct shapes=<n> transitions=<n>`, then each site as `site <name>: <monomorphic|polymorphic|megamorphic> (<n> shape[s])` — one shape, two to four, more than four.

Example →

```
p1: S2 (x,y)
p2: S2 (x,y)
p3: S4 (y,x)
…
distinct shapes=5 transitions=5
site readX: monomorphic (1 shape)
site readXY: polymorphic (2 shapes)
site mixed: megamorphic (5 shapes)
```

`p3` added the same two properties in the other order and got a different shape — order is part of the hidden class.

INPUT 1

```
p1: x y
p2: x y
p3: y x
p4: x y z
p5: x
p6: x y -x
site readX p1 p2
site readXY p1 p3
site mixed p1 p3 p4 p5 p6
site dict p6
```

OUTPUT 1

```
p1: S2 (x,y)
p2: S2 (x,y)
p3: S4 (y,x)
p4: S5 (x,y,z)
p5: S1 (x)
p6: DICT(p6) (<dictionary>)
distinct shapes=5 transitions=5
site readX: monomorphic (1 shape)
site readXY: polymorphic (2 shapes)
site mixed: megamorphic (5 shapes)
site dict: monomorphic (1 shape)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Each line: `<name>: <property> <property> ...` — the order properties are added;
// A `site <name> <obj> <obj> ...` line is a property-access site that has seen thos
const transitions = new Map();     // "S0|x" → shape id
let nextShape = 1;
const shapeProps = new Map([["S0", []]]);
const shapes = new Map();          // object name → shape id
const sites = [];
for (const line of lines) {
  if (line.startsWith("site ")) { const [, name, ...objs] = line.trim().split(/\s+/)
  const [name, rest] = line.split(":");
  let shape = "S0";
  for (const step of (rest ?? "").trim().split(/\s+/).filter(Boolean)) {
    if (step.startsWith("-")) { shape = `DICT(${name.trim()})`; shapeProps.set(shape
    const key = `${shape}|${step}`;
    if (!transitions.has(key)) { transitions.set(key, `S${nextShape++}`); shapeProps
    shape = transitions.get(key);                          // the same property adde
  }
  shapes.set(name.trim(), shape);
}
for (const [name, shape] of shapes) console.log(`${name}: ${shape} (${shapeProps.get
console.log(`distinct shapes=${new Set(shapes.values()).size} transitions=${transiti
for (const site of sites) {
  const seen = new Set(site.objs.map((o) => shapes.get(o)));
  const state = seen.size === 1 ? "monomorphic" : seen.size <= 4 ? "polymorphic" : '
  console.log(`site ${site.name}: ${state} (${seen.size} shape${seen.size === 1 ? "'
}
```

EXERCISE 2

Elements-kinds tracer

Each line is the sequence of values pushed into a fresh array: integers, decimals, `_` for a hole (a write past the end), anything else a non-number. Track the array's elements kind starting from `PACKED_SMI`: the type part moves `SMI → DOUBLE → ELEMENTS` and never back; a hole flips `PACKED` to `HOLEY` permanently. Print `[<tokens>] ⇒ <kind> → <kind> …`, listing a kind only when it changes.

Example: `2 _ 3.5 x` → `[2 _ 3.5 x] ⇒ PACKED_SMI → HOLEY_SMI → HOLEY_DOUBLE → HOLEY_ELEMENTS`; `1 2 3` → `[1 2 3] ⇒ PACKED_SMI`.

INPUT 1

```
1 2 3
1 2.5 3
1 _ 3
1 a
2 _ 3.5 x
1.5 1
```

OUTPUT 1

```
[1 2 3] ⇒ PACKED_SMI
[1 2.5 3] ⇒ PACKED_SMI → PACKED_DOUBLE
[1 _ 3] ⇒ PACKED_SMI → HOLEY_SMI
[1 a] ⇒ PACKED_SMI → PACKED_ELEMENTS
[2 _ 3.5 x] ⇒ PACKED_SMI → HOLEY_SMI
→ HOLEY_DOUBLE → HOLEY_ELEMENTS
[1.5 1] ⇒ PACKED_SMI → PACKED_DOUBLE
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Tokens: integers, decimals, `_` for a hole, anything else is a non-number element
const RANK = { SMI: 0, DOUBLE: 1, ELEMENTS: 2 };
function kindAfter(kind, token) {
  let [packing, type] = kind.split("_");
  if (token === "_") packing = "HOLEY";                                    // a h
  else {
    const t = /^-?\d+$/.test(token) ? "SMI" : /^-?\d+\.\d+$/.test(token) ? "DOUBLE"
    if (RANK[t] > RANK[type]) type = t;                                     // SM
  }
  return `${packing}_${type}`;
}
for (const line of lines) {
  const tokens = line.trim().split(/\s+/);
  let kind = "PACKED_SMI";
  const path = [kind];
  for (const t of tokens) { const next = kindAfter(kind, t); if (next !== kind) path
  console.log(`[${tokens.join(" ")}] ⇒ ${path.join(" → ")}`);
}
```

QUIZ

1. Where does every function start executing in V8?

A. In TurboFan, the optimising compiler

B. In Ignition, the bytecode interpreter — feedback gathered there decides what gets optimised later

C. As native code from the parser

D. In a Web Worker

2. `const a = { x: 1, y: 2 }` and `const b = { y: 2, x: 1 }`…

A. Share a hidden class

B. Have different hidden classes — property order is part of the shape

C. Are both dictionary-mode objects

D. Cannot both be optimised

3. A property-access site that has seen six different shapes is…

A. Monomorphic

B. Polymorphic

C. Megamorphic — it gives up caching and uses a slow generic lookup

D. Deoptimised permanently

4. `const arr = [1, 2, 3]; arr[10] = 4;` makes the array…

A. PACKED_SMI still

B. HOLEY_SMI — and it never goes back to PACKED

C. PACKED_DOUBLE

D. A typed array

5. Deoptimisation happens when…

A. A function is called too often

B. An assumption baked into optimised code fails — a new shape or type appears — and execution falls back to the interpreter

C. The garbage collector runs

D. A `try` / `catch` is entered

ANSWER KEY · HOW V8 RUNS YOUR CODE — PARSING, TIERS, HIDDEN CLASSES AND INLINE CACHES

1. B Interpret first, optimise hot code on feedback, deoptimise on surprise.

2. B Same properties in the same order share a shape; `delete` drops to dictionary mode.

3. C One shape is monomorphic, two to four polymorphic, more is megamorphic.

4. B Elements kinds transition one way: SMI → DOUBLE → ELEMENTS, PACKED → HOLEY.

5. B Repeated deopts for the same function eventually make V8 stop optimising it.

LESSON 2 · 13 MIN

Measure first — benchmarks that lie, profiles that do not, and reading the numbers

Almost every performance change made without measuring makes the code worse in some way — less readable, more fragile, or actually slower — because intuition about JIT-compiled code is unreliable and the bottleneck is rarely where it feels. The discipline is simple to state: **profile to find the hot spot, benchmark to compare alternatives, measure in production to confirm**. Each step has traps: benchmarks that time the wrong thing (JIT warm-up, dead-code elimination, GC pauses), profiles read at the wrong level, and summary numbers (averages) that hide the users who suffer. This lesson covers the tools — `performance.now`, `console.time`, `--cpu-prof`, DevTools, flame charts — and the statistics you need to trust a result: warm-up, repetitions, medians and percentiles, variance, and what "significant" means.

Timing correctly

```
const t0 = performance.now();              // monotonic, sub-millisecond; globa
work();
const ms = performance.now() - t0;
process.hrtime.bigint();                   // nanoseconds, for very short spans
console.time("parse"); parse(); console.timeEnd("parse");   // quick and dirt
```

`Date.now()` is wall-clock and coarse; never use it for durations. `performance.mark` / `measure` label spans that DevTools and `PerformanceObserver` can display. In browsers, `performance.now()` is coarsened (to ~100 µs, or 5 µs with cross-origin isolation) for security — fine for anything you can see, useless for nanosecond claims.

Why naive benchmarks lie

```
const t0 = performance.now();
for (let i = 0; i < 1e6; i++) add(i, 1);       // what did this measure?
```

JIT warm-up: the first thousands of iterations run interpreted, then get optimised mid-loop — the total mixes tiers. Warm the code before timing, then time a separate run.

Dead-code elimination: `add(i, 1)`'s result is unused; the optimiser may remove the call entirely and you time an empty loop. Consume results (accumulate and print them).

Constant folding and inlining: a benchmark with literal inputs lets the compiler precompute answers a real workload would not. Feed data from outside the function (an array built at runtime).

GC pauses: an allocation-heavy loop pays for collections at unpredictable moments — a single run's time is noise. Repeat and look at the distribution.

Micro-benchmarks measure micro-effects: a 2× difference on a 5 ns operation is irrelevant next to one database call. Benchmark the operation at the granularity that matters.

Machine noise: turbo boost, other processes, thermal throttling. Run several times, interleave the alternatives, and compare distributions, not single numbers.

Tools that handle this for you: **Benchmark.js**/**tinybench**/**mitata** (warm-up, many samples, statistics), Node's `--allow-natives-syntax` with `%OptimizeFunctionOnNextCall` for controlled experiments, and `deno bench` / `bun bench`. Write the harness yourself once to see what they do; then use one.

Reading a distribution

Report the **median** (p50) and a high percentile (p95/p99), not the mean — a few slow runs (GC, page faults) drag the mean up and hide the typical case; a few fast ones hide the tail that users notice. Report the **spread** (standard deviation, or min/max, or an interquartile range). Two alternatives are distinguishable only when their distributions barely overlap: a 3% difference with 10% run-to-run variance is noise. Discard the first samples (warm-up), use enough repetitions that the median stabilises, and be suspicious of any result you cannot reproduce twice.

Profiling: find the hot spot

Benchmarks compare alternatives *you already suspect*. A **profiler** tells you where time actually goes:

Node: `node --cpu-prof app.js` writes a `.cpuprofile` to open in Chrome DevTools (Performance panel → load) or `speedscope`; `--prof` + `--prof-process` gives a text tick summary; `--inspect` attaches DevTools live. `clinic flame` builds flame graphs from a running process.

Browser: DevTools → Performance → record → the flame chart (time on the x-axis, call depth on the y-axis: a wide bar is where time goes; a tall narrow stack is recursion) plus the bottom-up table (self time per function). Long tasks over 50 ms are the input-blocking culprits.

Sampling (what these do) runs the program at full speed and samples the stack every ~1 ms — cheap and honest; **instrumentation** (timers around every function) is exact but distorts.

Read a profile top-down for structure and bottom-up for self time. The usual finding: one or two functions account for most of the time, and they are not the ones you guessed. Optimise those; leave the rest alone.

Complexity before constants

Before any micro-work, check the **algorithm**: a linear scan inside a loop (`includes` per item — O(n²)), a repeated `Object.keys` in a hot loop, sorting inside a comparator, re-rendering everything for one change, an N+1 query. Big-O differences swamp every JIT effect: replacing an O(n²) with O(n) on 10 000 items is a 10 000× change; the fastest possible O(n²) is still O(n²). Estimate complexity by measuring operation counts at two or three input sizes — if doubling n quadruples the work, no amount of shape stability will save you.

Measuring in production

Laboratory numbers do not survive contact with real hardware, networks and data. **Real-user monitoring** (RUM) — `PerformanceObserver` for Core Web Vitals in browsers, request timings and event-loop lag in servers, exported to your metrics system — shows the distribution users actually experience. Track **percentiles per endpoint/route**, watch for regressions per deploy, and alert on the tail (p99), which is where timeouts and retries live. This repository's telemetry writes route timings for exactly this reason.

The loop

Set a target (a budget: "p95 under 200 ms", "60 fps while scrolling").

Profile to find the hot spot.

Fix the algorithm first, then the constants.

Benchmark the change against the original, warmed and repeated.

Ship, measure in production, keep or revert.

Optimisations without a measured before/after are guesses, and guesses that complicate code should be reverted.

Common mistakes

Timing with `Date.now()`; single runs; means instead of medians.

Benchmarks whose results are unused (eliminated), with literal inputs (folded), or without warm-up (mixed tiers).

Micro-optimising a function that takes 0.1% of the profile.

Comparing alternatives on different machines/runs without interleaving.

Optimising for lab hardware and never measuring in production.

Keeping a "faster" version that measurably is not.

Interview angle

"How would you find why a Node service is slow?" Profile it (`--cpu-prof` / `--inspect`), read the flame chart for self time, check event-loop lag and per-route percentiles, fix the algorithmic hot spot, re-measure.

"Why do micro-benchmarks mislead?" JIT warm-up, dead-code elimination, constant folding, GC noise, and measuring effects too small to matter.

"Mean or median?" Median plus a high percentile; means hide both the typical case and the tail.

"What is a flame chart?" Sampled call stacks over time: width is time spent, depth is call nesting; wide bars are hot spots.

"What do you optimise first?" The algorithm (Big-O), then allocation and shape stability in the measured hot spot.

Key takeaways

`performance.now()` / `hrtime` for durations; `Date.now()` never.

Benchmarks need warm-up, many samples, consumed results, runtime inputs and interleaving; use a harness.

Report p50 and p95/p99 with spread; overlapping distributions mean no difference.

Profile (`--cpu-prof` , DevTools flame charts) to find the hot spot; fix complexity before constants.

Measure in production per route/percentile; every optimisation has a before and an after or it is a guess.

EXERCISE 1

Benchmark statistics

Line 1 is `warmup=<k>` ; each following line is `<name>: <sample> <sample> …` (milliseconds). Drop the first `k` samples of each series (the JIT warm-up), then print `<name>: n=<> min=<> median=<> mean=<> p95=<> stddev=<> outliers=<>` with two decimals: p95 by nearest rank, the sample standard deviation, and outliers counted as samples more than `3 × 1.4826 × MAD` from the median. If there are two series, compare the second to the first: `<b> vs <a>: median <±x.xx>% → <verdict>` where the verdict is `not distinguishable (IQRs overlap)` when the interquartile ranges (p25–p75) overlap, else `faster` or `slower` .

Example: `warmup=2` , `baseline: 40 25 12.1 12.4 12.0 12.2 12.9 12.3 12.1 30.0 12.2` , `candidate: 38 24 11.0 11.2 10.9 11.1 11.3 11.0 11.2 11.1 11.0` →

```
baseline: n=9 min=12.00 median=12.20 mean=14.24 p95=30.00 stddev=5.91 outliers=2
candidate: n=9 min=10.90 median=11.10 mean=11.09 p95=11.30 stddev=0.13 outliers=0
candidate vs baseline: median -9.02% → faster
```

The baseline's mean is dragged to 14.24 by one GC pause; its median tells the truth.

INPUT 1

```
warmup=2
baseline: 40 25 12.1 12.4 12.0 12.2 12.9
12.3 12.1 30.0 12.2
candidate: 38 24 11.0 11.2 10.9 11.1
11.3 11.0 11.2 11.1 11.0
```

OUTPUT 1

```
baseline: n=9 min=12.00 median=12.20
mean=14.24 p95=30.00 stddev=5.91
outliers=2
candidate: n=9 min=10.90 median=11.10
mean=11.09 p95=11.30 stddev=0.13
outliers=0
candidate vs baseline: median -9.02% →
faster
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sorted = (xs) => [...xs].sort((a, b) => a - b);
const percentile = (xs, p) => { const s = sorted(xs); return s[Math.min(s.length - 1
const median = (xs) => { const s = sorted(xs); const m = s.length >> 1; return s.len
const mean = (xs) => xs.reduce((a, b) => a + b, 0) / xs.length;
const warmup = Number(lines[0].replace(/^warmup=/, ""));
const stddev = (xs) => { const m = mean(xs); return Math.sqrt(xs.reduce((s, x) => s
const mad = (xs) => { const m = median(xs); return median(xs.map((x) => Math.abs(x -
const series = lines.slice(1).map((l) => { const [name, rest] = l.split(":"); return
const f = (x) => x.toFixed(2);
for (const s of series) {
  s.samples = s.raw.slice(warmup);                                    // the first ru
  const m = median(s.samples), d = mad(s.samples);
  const outliers = s.samples.filter((x) => Math.abs(x - m) > 3 * 1.4826 * d).length;
  console.log(`${s.name}: n=${s.samples.length} min=${f(Math.min(...s.samples))} med
}
if (series.length >= 2) {
  const [a, b] = series;
  const q = (xs, p) => percentile(xs, p);
  const overlap = !(q(a.samples, 75) < q(b.samples, 25) || q(b.samples, 75) < q(a.sa
  const diff = ((median(b.samples) - median(a.samples)) / median(a.samples)) * 100;
  console.log(`${b.name} vs ${a.name}: median ${diff >= 0 ? "+" : ""}${f(diff)}% →
}
```

EXERCISE 2

Fit the complexity

Each line is `<n> <operations>` — an operation count measured at input size n. For each model in the starter's table (`O(1)`, `O(log n)`, `O(n)`, `O(n log n)`, `O(n^2)`) compute `operations / f(n)` per point and the coefficient of variation of those ratios (population standard deviation ÷ mean): a perfect fit gives a constant ratio, so the smallest CV wins. Print every model as `<name padded to 10> cv=<x.xxx>` sorted ascending, then `best fit: <name>`, then `n x<ratio> → ops x<ratio>` from the first to the last point (one decimal).

Example: `10 100`, `20 400`, `40 1600`, `80 6400` →

```
O(n^2)     cv=0.000
O(n log n) cv=0.527
O(n)       cv=0.715
O(log n)   cv=1.088
O(1)       cv=1.191
best fit: O(n^2)
n x8.0 → ops x64.0
```

INPUT 1

```
100 700
200 1600
400 3600
800 8000
1600 17600
```

OUTPUT 1

```
O(n log n) cv=0.007
O(n)       cv=0.157
O(n^2)     cv=0.750
O(log n)   cv=0.876
O(1)       cv=0.982
best fit: O(n log n)
n x16.0 → ops x25.1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const points = lines.map((l) => l.trim().split(/\s+/).map(Number));   // [n, operati
const MODELS = {
  "O(1)": () => 1, "O(log n)": (n) => Math.log2(n), "O(n)": (n) => n, "O(n log n)":
};
const cv = (xs) => { const m = xs.reduce((a, b) => a + b, 0) / xs.length; const sd =
const scores = Object.entries(MODELS).map(([name, f]) => [name, cv(points.map(([n, c
for (const [name, c] of scores) console.log(`${name.padEnd(10)} cv=${c.toFixed(3)}`)
console.log(`best fit: ${scores[0][0]}`);
const [n1, o1] = points[0], [n2, o2] = points.at(-1);
console.log(`n x${(n2 / n1).toFixed(1)} → ops x${(o2 / o1).toFixed(1)}`);
```

QUIZ

1. Which clock should time a duration?

A. `Date.now()`

B. `performance.now()` (or `process.hrtime`) — monotonic and sub-millisecond; `Date.now()` is wall-clock and coarse

C. `new Date().getTime()`

D. `setTimeout` callbacks

2. A loop that calls `add(i, 1)` a million times and discards the result measures…

A. The cost of `add`

B. Possibly an empty loop — dead-code elimination can remove the unused call; consume the results

C. Garbage collection

D. Nothing at all, it throws

3. Why report the median rather than the mean of benchmark samples?

A. The mean is harder to compute

B. A few slow runs (GC pauses, page faults) drag the mean up and hide the typical case

C. The median is always smaller

D. Benchmark tools cannot compute means

4. In a flame chart, a wide bar means…

A. A deep call stack

B. A function where a lot of time is spent — width is time, depth is the call stack

C. A memory leak

D. An error

5. Doubling the input quadruples the operation count. Before micro-optimising you should…

A. Cache `arr.length`

B. Fix the algorithm — that is O(n²), and no shape stability will rescue it

C. Switch `forEach` to `for`

D. Add a Web Worker

ANSWER KEY · MEASURE FIRST — BENCHMARKS THAT LIE, PROFILES THAT DO NOT, AND READING THE NUMBERS

1. B Wall clocks jump with NTP and DST; monotonic clocks only move forward.

2. B Also warm up first, use runtime inputs, and repeat for a distribution.

3. B Pair it with p95/p99 and a spread; overlapping distributions mean no difference.

4. B Read top-down for structure and bottom-up for self time.

5. B Complexity before constants; Big-O swamps every JIT effect.

LESSON 3 · 13 MIN

Memory and garbage collection — generations, reachability, and the leaks that survive it

JavaScript frees memory for you, which is why memory problems in JavaScript are not "forgot to free" but "kept a reference": the collector reclaims only what is **unreachable**, and one surviving reference — a listener, a cache entry, a closure, a timer — keeps an entire object graph alive. Understanding how V8's collector works (a young generation for short-lived objects, an old generation for survivors), what reachability means precisely, and the handful of patterns that leak in practice turns "the process grows until it dies" from a mystery into a checklist. This lesson covers the heap and its generations, allocation cost, the leak catalogue, the diagnostic tools (heap snapshots, `process.memoryUsage`, comparisons), and the limits you should know.

Reachability

An object is alive if it can be reached from a **root** — the global object, the current call stack (locals and arguments of every active frame), and internal engine references — by following properties, closure scopes, array elements, Map/Set entries and prototype links. Everything else is garbage and will be collected *eventually*; when is not specified and not observable (module 6). Two consequences: a reference *anywhere* on a reachable path keeps the target alive, however "forgotten"; and freeing is never manual — you release memory by dropping references (`x = null`, `map.delete`, `removeEventListener`, letting a scope end).

The heap: young and old

V8 splits the heap by object age, betting on the **generational hypothesis** — most objects die young:

New space (young generation), a few MB, split into two semi-spaces. Allocation is a pointer bump — nearly free. It fills quickly; a **scavenge** (minor GC) copies the live objects to the other semi-space and abandons the dead ones in one sweep. Cheap because most are dead. Objects that survive two scavenges are **promoted** to old space.

Old space (old generation), large: survivors, big objects, and code. Collected by **mark–sweep–compact** (major GC): mark everything reachable from the roots, sweep the rest,

occasionally compact to defragment. Expensive and proportional to the live heap — but V8 runs it **incrementally and concurrently** (Orinoco) so pauses stay short, and only when old space grows.

The performance angle: a high **allocation rate** of short-lived objects costs scavenges (cheap, but not free — and each promotes a few objects that then cost major GCs); a growing set of **long-lived** objects costs major GCs proportional to the heap. Allocation-heavy hot loops (a temporary object per iteration, spread copies in `reduce`) show up in profiles as GC time; the fix is fewer allocations in the hot path — local mutable accumulators, reusing buffers, typed arrays — not object pooling everywhere (V8's allocator is faster than most pools).

The leak catalogue

A leak is a reference you did not intend to keep. In order of frequency:

Unbounded caches — a module-level `Map`/object keyed by request data that only grows (module 6's `WeakMap` and module 11's LRU are the fixes).

Event listeners never removed — a component adds `window.addEventListener` and is destroyed without removing it; the listener's closure keeps the component and everything it references. Same for `EventEmitter.on` on long-lived emitters (Node warns at 11 listeners for a reason).

Timers — `setInterval` never cleared; a `setTimeout` chain that restarts itself. The callback keeps its closure alive.

Closures over large data — a small callback captures a variable from a scope that also holds a large buffer; the whole scope object stays.

Detached DOM nodes — removed from the document but still referenced from a JavaScript array or Map; each keeps its subtree.

Global accumulation — arrays that are pushed to and never trimmed (logs, history, "seen" lists); `globalThis` properties.

Promises never settled — pending promises hold their reaction closures; a queue of `new Promise` that never resolves grows forever.

Bound methods and per-render closures — a new bound function per render added as a listener each time.

Each one is "a reference you forgot", which is why the diagnostic is always the same: find what is holding the objects.

Diagnosing

```
process.memoryUsage();          // { rss, heapTotal, heapUsed, external, arra
```

`heapUsed` climbing across requests (after GC — trigger it with `--expose-gc` and `global.gc()` in a test, never in production) is the signature. Then a **heap snapshot**: Chrome DevTools → Memory → Heap snapshot (browser or Node via `--inspect`), or `v8.writeHeapSnapshot()` / `--heapsnapshot-signal` in Node. Take two snapshots with the suspected activity between them and use **Comparison** view: objects whose count grew are the leak; the **Retainers** panel shows the path from a root to a leaked object — the reference you forgot. Sort by *retained size* (what would be freed) not *shallow size*. The **Allocation timeline** records who allocated what between two points. `--trace-gc` logs every collection with heap sizes — a quick way to see promotion pressure and whether the old generation is growing.

Limits and tuning

Node's old space is capped (roughly 2–4 GB depending on version and platform; `--max-old-space-size=4096` raises it). Hitting it produces `FATAL ERROR: Reached heap limit — JavaScript heap out of memory`, usually from a leak or from loading something huge into memory (a whole file, a whole result set) instead of streaming (module 9). Raising the limit buys time; it does not fix a leak. Buffers and typed arrays live outside the JavaScript heap (`external` / `arrayBuffers`) and have their own accounting. Browser tabs have their own limits and kill the page.

Weak references, again

`WeakMap` / `WeakSet` for metadata keyed by objects, `WeakRef` for droppable caches, `FinalizationRegistry` as a backstop — module 6 covered when. They are the tools for "I want to associate data with this object without keeping it alive", and the wrong tool for "I want to know when memory is freed" (unobservable by design).

Habits

Every `addEventListener` / `on` / `setInterval` /subscription has a matching removal in the component's teardown, or uses an `AbortSignal`.

Caches are bounded (LRU/TTL) or weak.

Long-lived collections are trimmed; "history" has a maximum length.

Large data is streamed, not accumulated; results are paginated.

Hot loops avoid per-iteration allocations; benchmarks include GC time (many samples).

Memory is measured in staging under load before and after a change, like latency.

Common mistakes

Treating growth as "the GC is lazy" — the collector cannot free what is reachable.

Module-level `Map` caches keyed by ids in a server; listeners without teardown; `setInterval` without `clearInterval`.

Loading files/query results whole; raising `--max-old-space-size` instead of fixing the leak.

Object pooling by reflex (slower than the allocator for small objects).

Reading shallow size instead of retained size in a snapshot; taking one snapshot instead of comparing two.

Interview angle

"How does JavaScript garbage collection work?" Reachability from roots; V8 uses a generational collector — cheap scavenges of the young generation, incremental mark-sweep-compact of the old — and frees only unreachable objects.

"What causes memory leaks in JavaScript?" Kept references: unbounded caches, listeners and timers never removed, closures over large scopes, detached DOM nodes, ever-growing arrays.

"How do you find a leak?" Watch `heapUsed` grow across GCs, take two heap snapshots around the activity, compare, and follow the retainer path from a root.

"Why is a high allocation rate a cost if young-gen GC is cheap?" Scavenges are frequent and each promotes a few objects that later cost major GCs; fewer allocations in hot paths means less GC time.

"When would you use a `WeakMap` to prevent a leak?" Data keyed by objects you do not own, so entries vanish with the objects.

Key takeaways

- Reachable = alive; leaks are references you kept — the collector is never the culprit.
- Young generation: cheap scavenges, survivors promoted; old generation: incremental mark-sweep-compact proportional to live heap.
- Leak catalogue: unbounded caches, listeners, timers, closures over big scopes, detached DOM, growing globals, pending promises.
- Diagnose with `process.memoryUsage`, two heap snapshots compared, retainer paths and retained size; `--trace-gc` for promotion pressure.
- Bound caches, remove listeners, stream large data, limit allocations in hot loops; `--max-old-space-size` buys time, not a fix.

EXERCISE 1

Reachability collector

Model the heap as a graph. Commands: `alloc <id> <bytes>`; `ref <from|root> <to>` adds an edge (from the root set when `root`); `unref <from|root> <to>` removes it; `gc` marks from the roots and sweeps everything unreachable, printing `gc: freed <ids or nothing> (<bytes> bytes) live=<ids or -> heap=<bytes> bytes`; `retainers <id>` prints the shortest path from a root — `retainers <id>: root -> a -> b -> <id>` — or `unreachable (garbage)` / `not allocated`. Ids are printed in allocation order.

Example: allocate a→b→c→d with d→b, root→a, then `retainers d`, `unref a b`, `gc` →

```
retainers d: root -> a -> b -> c -> d
gc: freed b,c,d (85 bytes) live=a heap=100 bytes
```

The cycle b→c→d→b kept nothing alive once `a` let go — reachability, not reference counting.

INPUT 1

```
alloc a 100
alloc b 50
alloc c 25
alloc d 10
ref root a
ref a b
ref b c
ref c d
ref d b
retainers d
unref a b
gc
retainers d
ref root d
alloc e 5
gc
```

OUTPUT 1

```
retainers d: root → a → b → c → d
gc: freed b,c,d (85 bytes) live=a
heap=100 bytes
retainers d: not allocated
gc: freed e (5 bytes) live=a heap=100
bytes
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const heap = new Map();    // id → { size, refs: Set }
const roots = new Set();
function mark() {
  const live = new Set();
  const stack = [...roots].filter((r) => heap.has(r));
  while (stack.length) {
    const id = stack.pop();
    if (live.has(id)) continue;
    live.add(id);
    for (const to of heap.get(id).refs) if (heap.has(to)) stack.push(to);
  }
  return live;
}
function retainers(target) {
  const prev = new Map();
  const queue = [...roots].filter((r) => heap.has(r));
  for (const r of queue) prev.set(r, null);
  while (queue.length) {
    const id = queue.shift();
    if (id === target) { const path = []; for (let n = id; n !== null; n = prev.get(
    for (const to of heap.get(id).refs) if (heap.has(to) && !prev.has(to)) { prev.se
  }
  return null;
}
for (const line of lines) {
  const [cmd, a, b] = line.trim().split(/\s+/);
  if (cmd === "alloc") heap.set(a, { size: Number(b ?? 1), refs: new Set() });
  else if (cmd === "ref") (a === "root" ? roots.add(b) : heap.get(a).refs.add(b));
  else if (cmd === "unref") (a === "root" ? roots.delete(b) : heap.get(a).refs.delet
  else if (cmd === "gc") {
    const live = mark();
    const dead = [...heap.keys()].filter((id) => !live.has(id));
    const freed = dead.reduce((s, id) => s + heap.get(id).size, 0);
    for (const id of dead) heap.delete(id);
    console.log(`gc: freed ${dead.join(",") || "nothing"} (${freed} bytes) live=${[
  } else if (cmd === "retainers") {
    const path = heap.has(a) ? retainers(a) : null;
    console.log(`retainers ${a}: ${path ? `root → ${path.join(" → ")}` : heap.has(
  }
}
```

EXERCISE 2

Leak detective

Each line is a heap-snapshot summary: `snapshot <n>: Class=count Class=count …`. A class whose count grows **strictly** at every snapshot (three or more snapshots) is a suspect; a class whose count never changes is stable; anything else is normal churn. Print suspects sorted by total growth as `suspect <Class>: <c1> → <c2> → … (+<per snapshot, one decimal>/snapshot, +<total> total)`, then `stable: <classes or →`, `fluctuating (normal churn): <classes or →`, and a verdict: `verdict: likely leak in <top suspect> — find who retains them in a heap snapshot comparison` or `verdict: no monotonic growth across snapshots`.

Example: four snapshots with Listener 100→200→300→400, Buffer bouncing, Session flat, Timer 5→8 →

```
suspect Listener: 100 → 200 → 300 → 400 (+100.0/snapshot, +300 total)
suspect Timer: 5 → 6 → 7 → 8 (+1.0/snapshot, +3 total)
stable: Session
fluctuating (normal churn): Buffer
verdict: likely leak in Listener — find who retains them in a heap snapshot comparis
```

INPUT 1

```
snapshot 1: Listener=100 Buffer=40
Session=10 Timer=5
snapshot 2: Listener=200 Buffer=35
Session=10 Timer=6
snapshot 3: Listener=300 Buffer=42
Session=10 Timer=7
snapshot 4: Listener=400 Buffer=38
Session=10 Timer=8
```

OUTPUT 1

```
suspect Listener: 100 → 200 → 300 →
400 (+100.0/snapshot, +300 total)
suspect Timer: 5 → 6 → 7 → 8
(+1.0/snapshot, +3 total)
stable: Session
fluctuating (normal churn): Buffer
verdict: likely leak in Listener — find
who retains them in a heap snapshot
comparison
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// Each line: `snapshot <n>: Class=count Class=count ...`
const snapshots = lines.map((l) => Object.fromEntries(l.split(":")[1].trim().split(/
const classes = [...new Set(snapshots.flatMap((s) => Object.keys(s)))].sort();
const suspects = [], stable = [], noisy = [];
for (const c of classes) {
  const counts = snapshots.map((s) => s[c] ?? 0);
  const growing = counts.every((v, i) => i === 0 || v > counts[i - 1]);           //
  const total = counts.at(-1) - counts[0];
  if (growing && snapshots.length >= 3) suspects.push({ c, counts, perSnapshot: tota
  else if (counts.every((v) => v === counts[0])) stable.push(c);
  else noisy.push(c);
}
suspects.sort((a, b) => b.total - a.total);
for (const s of suspects) console.log(`suspect ${s.c}: ${s.counts.join(" → ")} (+${
console.log(`stable: ${stable.join(", ") || "-"}`);
console.log(`fluctuating (normal churn): ${noisy.join(", ") || "-"}`);
console.log(suspects.length ? `verdict: likely leak in ${suspects[0].c} — find who r
```

QUIZ

1. An object is eligible for collection when…

A. Its reference count reaches zero

B. It is unreachable from every root — the global object, the call stack, engine internals — by any path

C. It has not been used for a while

D. `delete` is called on it

2. Objects that survive two scavenges…

A. Are freed

B. Are promoted to old space, which is collected by incremental mark–sweep–compact

C. Stay in new space forever

D. Become weak references

3. Which is the most common JavaScript memory leak?

A. Forgetting to call `free()`

B. A module-level cache (`Map`/object) keyed by request data that only grows

C. Using `const`

D. Deep recursion

4. To find what is leaking you…

A. Read `heapTotal` once

B. Take two heap snapshots around the suspected activity, compare them, and follow the retainer path of an object whose count grew

C. Raise `--max-old-space-size`

D. Call `global.gc()` in production

5. Why is a high allocation rate a cost when young-generation GC is cheap?

A. It is not a cost

B. Scavenges are frequent and each promotes a few objects that later cost major GCs proportional to the live heap

C. Allocation is synchronous I/O

D. It fragments the stack

ANSWER KEY · MEMORY AND GARBAGE COLLECTION — GENERATIONS, REACHABILITY, AND THE LEAKS THAT SURVIVE IT

1. B Cycles are no problem; a single kept reference anywhere on a reachable path is.

2. B The generational hypothesis: most objects die young, so scavenging the young generation is cheap.

3. B Bound it (LRU/TTL) or key it weakly; listeners and timers without teardown come next.

4. B Sort by retained size, not shallow size; `--trace-gc` shows promotion pressure.

5. B Fewer allocations in hot paths means less GC time — not object pools, which V8's allocator outruns.

LESSON 4 · 13 MIN

Writing fast JavaScript — the habits that survive measurement

After profiling has pointed at a hot spot, what do you actually change? The techniques that reliably help are few and mostly boring: fix the algorithm, use the right data structure, avoid allocating in the loop, keep shapes and types uniform, batch work that crosses a boundary, and move CPU-heavy work off the main thread. The techniques that do not help — or hurt — are the folklore: `for` versus `forEach`, string concatenation tricks, object pools, avoiding closures. This lesson is the first list with reasons, the second list with why it is obsolete, and the meta-rule: every item here is a *response to a measurement*, never a default style.

---

1. Fix the complexity

The largest wins, always. Patterns to recognise in a profile:

Slow shape	Fast shape
`arr.includes(x)` / `find` inside a loop over another array — O(n·m)	a `Set` / `Map` built once — O(n + m)
`Object.keys(obj).length` or `JSON.stringify` inside a loop	compute once outside
`arr.shift()` as a queue — O(n) per dequeue	an index pointer or a linked deque
`[...acc, x]` / `{ ...acc }` in `reduce` — O(n²)	push into a local accumulator
`str += piece` for megabytes	parts array + one `join`, or a stream
sort inside a comparator; `localeCompare` in a hot sort	precompute keys; `Intl.Collator` reused
repeated `querySelector` in a loop	query once, or delegate
N+1 database queries	one query with `IN`, or a join

Estimate: how many times does the inner operation run for n items? If the answer grows faster than n, that is the fix.

---

2. Choose the data structure

`Set` for membership, `Map` for lookups by key (any type) and frequent add/delete, arrays for ordered sequences and iteration, typed arrays for large numeric data, `Object.create(null)` / `Map` for dictionaries keyed by data. An array of objects with 8 fields is far more memory- and cache-efficient than an object of arrays only when iterated whole; **struct-of-arrays** (`Float64Array` per field) wins for numeric batch computation (physics, charts, image data). Strings are immutable: repeated `slice` / `concat` on huge strings creates rope structures that are fine to build and slow to index.

---

3. Allocate less in hot paths

Every `{}` , `[]` , closure, spread, `map` result and template literal in a hot loop is an allocation the young-gen collector must process. Techniques: reuse a scratch array or object across iterations (reset it, do not reallocate); return primitives or reuse a result object when the caller consumes it immediately; prefer `for…of` / `for` over chains that create intermediate arrays **when profiling shows it**; use typed arrays as fixed buffers; avoid creating closures inside the loop body (hoist them). Do not pool small objects — V8 allocates faster than a pool checks in and out, and pools leak.

---

4. Keep shapes and types stable (lesson 1)

Uniform objects from one constructor or literal shape; no conditional property adds; no `delete` ; homogeneous arrays without holes; numbers that stay integers or stay doubles; one type per variable in hot functions (a variable that is sometimes a string and sometimes a number makes every operator on it polymorphic). Small, single-purpose functions inline well; a giant function with a dozen roles does not.

---

5. Do less work

Lazy evaluation: compute on first use (`once`, lazy getters — module 11); render what is visible (virtualised lists); load what is needed (code splitting, `IntersectionObserver`).

Memoize pure, repeated, expensive computations with bounded caches (module 11).

Debounce/**throttle** high-frequency input (module 8): one search per pause, one layout per frame.

Early exit: `some` / `find` / `break` instead of processing everything and filtering at the end.

Cache derived values when the source changes rarely; invalidate by reference (module 11's selectors).

---

6. Batch across boundaries

Crossing a boundary — DOM, network, disk, process, worker — costs far more than the work inside. Batch DOM writes (fragments, `requestAnimationFrame` — module 12), coalesce network requests (one request for ten ids, not ten requests), write files in chunks or through a stream, send one message with an array to a worker rather than a thousand messages. The event loop is a boundary too: a 200 ms synchronous chunk blocks everything; split long CPU work across tasks (`setImmediate`, `scheduler.yield()` in browsers) so input and I/O interleave — or move it out entirely.

---

7. Move CPU work off the thread

`worker_threads`/Web Workers for parsing, compression, image processing, crypto, large sorts; pass `ArrayBuffer`s by **transfer** (zero-copy) or use `SharedArrayBuffer` for shared numeric state; use a worker pool sized to cores. WebAssembly for existing native libraries or hot numeric kernels (module 14). Neither helps I/O-bound code — that is already asynchronous.

---

8. Stream instead of accumulate

Whole-file reads, whole-result-set queries and whole-response buffers cost memory proportional to the data and delay the first output until the last byte arrives. Streams (module 9) process in constant memory and start emitting immediately; paginate queries; use `for await` over line-oriented input.

Folklore, retired

`for` versus `forEach` / `map`: the engine inlines small callbacks; pick by readability, switch only if the profiler names the loop.

String building with arrays "for speed": `+=` is fine up to large sizes (ropes); use `join` for clarity or streams for huge output.

Avoiding closures/ `try` / `arguments` / `const` for speed: no longer meaningful.

Object pooling for small objects: slower and leak-prone.

Caching `arr.length` in a `for` condition, bit tricks (`x >> 1`), `switch` over `if` chains: micro-noise the compiler already handles.

`==` versus `===`: `===` is correct and not slower.

The meta-rule

Readable code that runs in the interpreter for one millisecond a day needs nothing from this list. Apply a technique when a profile shows the hot spot and a benchmark shows the change helps; leave a comment saying what was measured (this codebase's comments carry measured latencies for exactly that reason); and remove the optimisation if a later measurement shows it stopped mattering. Fast code that nobody can read is a maintenance bill; unreadable fast code with no measurement behind it is the worst of both.

Common mistakes

Micro-optimising before checking Big-O.

Rewriting readable chains into loops everywhere "for performance" without a profile.

`arr.shift()` queues; `includes` in loops; spreading accumulators.

Per-iteration allocations and closures in hot loops; polymorphic variables.

Batching nothing: a DOM write or a request per item.

CPU-heavy work on the main thread; whole-file/whole-result loading.

Interview angle

"How would you speed up a slow function?" Profile; fix complexity and data structures; reduce allocations and keep shapes stable in the hot path; batch boundary crossings; measure before and after.

"`for` loop or `map`?" Whichever reads better — the engine inlines small callbacks; change only when a profile names the loop.

"When are typed arrays worth it?" Large numeric data: fixed representation, no boxing or holes, transferable to workers.

"How do you keep the UI responsive during heavy computation?" Split across frames/tasks or move to a Web Worker; batch DOM writes.

"Should you pool objects in JavaScript?" Rarely — the allocator is fast and pools leak; reuse a scratch buffer only where a profile shows allocation cost.

Key takeaways

Complexity and data structures first: Sets/Maps for lookups, no O(n²) reduces, no `shift` queues, no N+1.

Then allocation and shape stability in the measured hot path; typed arrays for numbers.

Do less (lazy, memoized, debounced, early exit) and batch every boundary crossing; split or offload long CPU work; stream large data.

Retire the folklore: loops versus methods, string tricks, closures, pools — pick readability.

Every optimisation answers a measurement and carries a comment saying so.

EXERCISE 1

Count the comparisons

Line 1 lists ids, line 2 the queries. Count the work each approach does — no timing. Membership: scan the ids for each query the way `includes` does (one comparison per element until a match) versus one hash probe per query with a `Set`; print `includes: comparisons=<> found=<> | Set: probes=<> found=<> | ratio=<x.x>x`. Queue: drain a copy of the ids with `shift()` counting the element moves each shift makes (remaining length − 1) versus stepping an index pointer; print `queue of <n>: shift moves=<> (O(n^2)) | index pointer steps=<> (O(n))`. Finish with `worst case for <n> ids: scan=<n> comparisons per miss, Set=1 probe`.

Example: ids `a b c d e f g h i j`, queries `j a zzz e j` →

```
includes: comparisons=36 found=4 | Set: probes=5 found=4 | ratio=7.2x
queue of 10: shift moves=45 (O(n^2)) | index pointer steps=10 (O(n))
worst case for 10 ids: scan=10 comparisons per miss, Set=1 probe
```

INPUT 1

```
a b c d e f g h i j
j a zzz e j
```

OUTPUT 1

```
includes: comparisons=36 found=4 | Set:
probes=5 found=4 | ratio=7.2x
queue of 10: shift moves=45 (O(n^2)) |
index pointer steps=10 (O(n))
worst case for 10 ids: scan=10
comparisons per miss, Set=1 probe
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const ids = lines[0].trim().split(/\s+/);
const queries = lines[1].trim().split(/\s+/);
let comparisons = 0, found = 0;
for (const q of queries) {
  let hit = false;
  for (const id of ids) { comparisons++; if (id === q) { hit = true; break; } }
  if (hit) found++;
}
const set = new Set(ids);
const setOps = queries.length;
const setFound = queries.filter((q) => set.has(q)).length;
console.log(`includes: comparisons=${comparisons} found=${found} | Set: probes=${set
let moves = 0;
const queue = [...ids];
while (queue.length) { moves += queue.length - 1; queue.shift(); }
let index = 0, pointerOps = 0;
while (index < ids.length) { index++; pointerOps++; }
console.log(`queue of ${ids.length}: shift moves=${moves} (O(n^2)) | index pointer s
console.log(`worst case for ${ids.length} ids: scan=${ids.length} comparisons per mi
```

EXERCISE 2

Typed-array lab

Line 1 is a list of numbers. Build `Int8Array`, `Uint8ClampedArray`, `Float32Array` and `Float64Array` views from it and print what each representation did to the values: `Int8Array=<JSON> (wraps modulo 256)`, `Uint8ClampedArray=<JSON> (clamps to 0..255, rounds)`, `Float32Array=<JSON> exact=<every element equals the input>`, `bytes: i8=<> f32=<> f64=<> elements=<>`, `sum`

`plain=<> typed=<> equal=<>`. Then show sharing: write `-1` through `subarray(0, 2)` and `-2` through `slice(0, 2)` and print `after subarray write: f64[0]=<> sharedBuffer=<> | after slice write: f64[1]=<> copyBuffer=<>`. Then a `DataView` on a 4-byte buffer with `setUint16(0, 258)`: `DataView: bytes=<b0,b1> readBE=<> readLE=<>`. Finally write to index 7 of a `Float64Array(3)` and print `out-of-range write on a typed array: ignored (no holes, no growth); length after: 3`.

Example: `1 127 128 300 -1 0.1` →

```
Int8Array=[1,127,-128,44,-1,0] (wraps modulo 256)
Uint8ClampedArray=[1,127,128,255,0,0] (clamps to 0..255, rounds)
Float32Array=[1,127,128,300,-1,0.10000000149011612] exact=false
bytes: i8=6 f32=24 f64=48 elements=6
sum plain=555.1 typed=555.1 equal=true
after subarray write: f64[0]=-1 sharedBuffer=true | after slice write: f64[1]=127 co
DataView: bytes=1,2 readBE=258 readLE=513
out-of-range write on a typed array: ignored (no holes, no growth); length after: 3
```

INPUT 1

```
1 127 128 300 -1 0.1
```

OUTPUT 1

```
Int8Array=[1,127,-128,44,-1,0] (wraps
modulo 256)
Uint8ClampedArray=[1,127,128,255,0,0]
(clamps to 0..255, rounds)
Float32Array=
[1,127,128,300,-1,0.10000000149011612]
exact=false
bytes: i8=6 f32=24 f64=48 elements=6
sum plain=555.1 typed=555.1 equal=true
after subarray write: f64[0]=-1
sharedBuffer=true | after slice write:
f64[1]=127 copyBuffer=false
DataView: bytes=1,2 readBE=258
readLE=513
out-of-range write on a typed array:
ignored (no holes, no growth); length
after: 3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const values = lines[0].trim().split(/\s+/).map(Number);
const i8 = Int8Array.from(values), u8c = Uint8ClampedArray.from(values), f32 = Float
console.log(`Int8Array=${JSON.stringify([...i8])} (wraps modulo 256)`);
console.log(`Uint8ClampedArray=${JSON.stringify([...u8c])} (clamps to 0..255, rounds
console.log(`Float32Array=${JSON.stringify([...f32])} exact=${[...f32].every((v, i)
console.log(`bytes: i8=${i8.byteLength} f32=${f32.byteLength} f64=${f64.byteLength}
const plainSum = values.reduce((a, b) => a + b, 0), typedSum = f64.reduce((a, b) =>
console.log(`sum plain=${plainSum} typed=${typedSum} equal=${plainSum === typedSum}`
const view = f64.subarray(0, 2);
view[0] = -1;                                                    // subarray shar
const copy = f64.slice(0, 2);
copy[1] = -2;                                                    // slice copies:
console.log(`after subarray write: f64[0]=${f64[0]} sharedBuffer=${view.buffer === f
const dv = new DataView(new ArrayBuffer(4));
dv.setUint16(0, 258);                                            // default big-e
console.log(`DataView: bytes=${[...new Uint8Array(dv.buffer, 0, 2)].join(",")} readE
let holey;
try { new Float64Array(3)[7] = 1; holey = "ignored (no holes, no growth)"; } catch (
console.log(`out-of-range write on a typed array: ${holey}; length after: ${new Floa
```

QUIZ

1. `for (const x of a) if (b.includes(x)) …` over two arrays of 10 000 is…

A. O(n) and fine

B. O(n·m) — 100 million comparisons; build a `Set` from `b` once and it becomes O(n + m)

C. A memory leak

D. Faster than a `Set`

2. `[...acc, x]` as the accumulator step in `reduce`…

A. Is idiomatic and free

B. Copies the whole accumulator every step — O(n²) for n items; push into a local array instead

C. Is O(log n)

D. Deoptimises the callback

3. `for` versus `forEach` / `map` for speed…

A. Always use `for`

B. Pick by readability — the engine inlines small callbacks; switch only when a profile names that loop

C. Always use `map`

D. Use `while`

4. Crossing a boundary (DOM, network, worker) per item should be…

A. Left alone

B. Batched — one fragment, one request for ten ids, one message with an array — because the crossing costs more than the work

C. Made synchronous

D. Moved into a `reduce`

5. Object pooling for small objects in JavaScript…

A. Is a reliable speed-up

B. Is usually slower than V8's bump allocator and leaks — reuse a scratch buffer only where a profile shows allocation cost

C. Is required for typed arrays

D. Prevents deoptimisation

ANSWER KEY · WRITING FAST JAVASCRIPT — THE HABITS THAT SURVIVE MEASUREMENT

1. B Complexity and data structures first; every other technique is a rounding error next to this.

2. B Same for `{ ...acc }` and for `arr.shift()` used as a queue.

3. B Most loop, closure and string-building folklore is obsolete; measure before believing it.

4. B The event loop is a boundary too: split long CPU work across tasks or move it to a worker.

5. B Every optimisation here answers a measurement and carries a comment saying so.

LESSON 5 · 13 MIN

Server and network performance — latency budgets, percentiles, caching and the event loop under load

A Node service is slow for reasons that rarely appear in a CPU profile: it waits — on a database 500 ms away, on ten sequential awaits that could be one, on a connection pool that is full, on a cold cache. The unit of performance here is **latency at the percentile users feel**, not CPU time, and the levers are round trips, caching, concurrency, and keeping the event loop free. This lesson covers how to think in latency budgets, why p99 matters more than the average, the round-trip arithmetic behind "count your awaits", caching layers and their invalidation, pools and backpressure, event-loop lag as the health metric of a Node process, and how to load-test before users do.

Latency budgets and percentiles

Decide what "fast" means as a number with a percentile: *p95 of* `/api/dashboard` *under 300 ms*. **Percentiles**: p50 is the typical request, p95 the slow-but-common, p99 the tail. Averages are useless — a mean of 120 ms can hide 5% of requests at 3 s, which is 5% of users refreshing. Tails compound: a page making 10 parallel calls sees the *max* of 10 draws, so its p50 is roughly the calls' p93. Budget the tail: if the target is 300 ms and the database round trip is 50 ms, sequential calls are capped at five before there is nothing left for the rest.

Round trips: count the awaits

```
const user = await db.user(id);              // 50 ms to the database
const team = await db.team(user.teamId);     // another 50 — dependent, unavoi
const [posts, badges] = await Promise.all([db.posts(id), db.badges(id)]);   //
```

In a topology where the database is far away (this repository's API is in Singapore, its MySQL in Phoenix — about 500 ms per round trip), the number of **sequential** awaits *is* the response time. Levers: run independent queries concurrently (`Promise.all`); replace N queries with one (`WHERE id IN (...)` , a join, a batch endpoint — the **N+1** problem is a loop of awaits); fetch only needed columns; cache composed payloads; move the compute next to the data (co-

location is the biggest lever of all). The same arithmetic applies to a browser calling an API: waterfalls of dependent fetches are the classic slow page.

Caching layers

A cache turns a slow, repeated read into a fast one at the price of **staleness** and **invalidation**:

In-process (L1): a bounded `Map`/LRU with a TTL — microseconds, per instance, lost on restart, inconsistent across instances.

Shared (L2): Redis/Memcached — a network hop (often 1–300 ms depending on where it is), shared by all instances.

HTTP: `Cache-Control`, `ETag`/`If-None-Match`, CDNs — the fastest cache is the one in the browser or at the edge that never reaches you.

Patterns: **cache-aside** (check, miss → load → store), **stale-while-revalidate** (serve stale immediately, refresh in the background — module 12's checkpoint), **single-flight** (one loader per key at a time so a cache miss under load does not stampede the database — cache the promise, module 8), **write-through**/**invalidate-on-write** (delete or update the key when the data changes; broadcast the invalidation to other instances). Cache **composed payloads** (the dashboard object) rather than individual rows: one hit saves ten queries, and invalidation has one key. Namespace keys by environment/database so a local process pointed at a shared Redis cannot feed production its data — this repository learned that one the hard way.

Pools and backpressure

A database allows N connections; a **pool** hands them out and queues callers beyond that. A slow query holds a connection; under load the queue grows and every request waits — latency climbs before CPU does. Size pools to the database's limit divided by your instance count, keep queries short, and add timeouts so a stuck connection is reclaimed.

Backpressure is the same idea for streams (module 9) and for producers of any kind: when the consumer cannot keep up, slow the producer (bounded queues, `write()` returning false, `Promise` pools with a limit — module 8) rather than buffering without bound until memory runs out. Rate limiting (this repository's `middleware/rate-limit.ts`) is backpressure at the edge.

The event loop under load

Node handles thousands of concurrent connections with one thread *only while that thread is free*. Any synchronous work — JSON parsing a 5 MB body, a regex with catastrophic backtracking (module 10), synchronous `fs`, a tight loop — blocks every other request for its duration. **Event-loop lag** (how late timers fire: `perf_hooks.monitorEventLoopDelay()`, or a `setInterval` that measures its own drift) is the single best health metric of a Node process: p99 lag over ~50–100 ms means requests are queuing behind CPU work. Fixes: move CPU work to workers, stream large bodies, cap body sizes (`express.json({ limit })`), and never `readFileSync` in a request handler. `cluster`/multiple processes use the other cores; each is still one loop.

Payloads and the wire

Compress responses (`compression` middleware, Brotli/gzip — text shrinks 5–10×); send only needed fields; paginate lists (cursor-based for large tables); stream large responses; keep connections alive (HTTP keep-alive / HTTP/2 multiplexing); set cache headers; put static assets on a CDN with immutable hashed names (module 14). On the browser side, the order of requests (critical first), preloading, and avoiding request waterfalls decide the perceived speed more than any server optimisation.

Load testing

Before users do: `autocannon` / `k6` / `wrk` against a staging environment with production-like data, ramping concurrency while watching p50/p95/p99, error rate, event-loop lag, pool queue length and memory. Find the **knee** — the concurrency where latency starts climbing — and the failure mode past it (timeouts? memory? pool exhaustion?). Then fix the first bottleneck and repeat; the knee moves. A load test that only reports requests-per-second at a fixed concurrency tells you little.

Observability

Record per-request duration, route, status and the time spent in each dependency (database, cache, external API); export histograms; alert on p99 and error rate per route; correlate deploys with regressions. Structured logs with a request id let you follow one slow

request across services. This is the production half of "measure first" — laboratory profiles find CPU hot spots; only production telemetry finds the slow database, the cold cache and the queueing pool.

Common mistakes

- Sequential awaits for independent work; N+1 queries; fetching whole rows and whole tables.
- Unbounded or un-namespaced caches; cache stampedes on miss; caching rows instead of payloads.
- Synchronous CPU work or `fs` in request handlers; huge bodies parsed in memory.
- Pool sizes larger than the database allows (times the instance count); no query timeouts.
- Reporting averages; load testing without percentiles or lag; optimising CPU when the profile is 95% waiting.

Interview angle

- *"Why p99 and not the average?"* Tails are where users suffer and where fan-out compounds; averages hide both the typical case and the tail.
- *"What is the N+1 problem?"* A query per item in a loop — n round trips instead of one batch query; fix with `IN` /joins or a batch endpoint.
- *"How do you keep a Node server responsive?"* Keep the loop free: workers for CPU, streams for large data, body limits, no sync I/O; watch event-loop lag.
- *"Cache-aside versus stale-while-revalidate?"* Miss → load → store versus serve stale now and refresh in the background; single-flight prevents stampedes.
- *"What is backpressure on a server?"* Bounded queues and pools that slow producers when consumers cannot keep up, instead of buffering until memory runs out.

Key takeaways

- Set latency budgets at percentiles; tails compound under fan-out; averages hide everything.
- Response time ≈ sequential round trips: `Promise.all` independent work, batch N+1, cache composed payloads, co-locate.

Caches: L1 bounded + L2 shared + HTTP; cache-aside, SWR, single-flight; invalidate on write; namespace keys.

Pools and bounded queues are backpressure; event-loop lag is the health metric; CPU work leaves the loop.

Compress, paginate, stream, keep-alive; load-test to the knee with percentiles; observe per route in production.

EXERCISE 1

Percentiles and SLO burn

Line 1 is `slo=<ms> target=<percent>`; the following lines are `<name>: <ms> <ms> …` request durations. For each series print `<name>: n=<> p50=<> p95=<> p99=<> mean=<one decimal> withinSLO=<xx.xx>% budgetBurn=<n>% <OK|BREACH>` — percentiles by nearest rank, budget burn = requests over the SLO ÷ the allowed failures (`n × (1 - target/100)`) as a percentage. When both `before` and `after` exist, add `p95 change: <±x.x>% → <regression|improvement|no significant change>` (more than ±10%) followed by `; mean change <x.x>% (means hide the tail)`.

Example: `slo=300 target=99`, a `before` series with one 950 ms request among twenty, an `after` series without it →

```
before: n=20 p50=127 p95=140 p99=950 mean=168.4 withinSLO=95.00% budgetBurn=500% BRE
after: n=20 p50=125 p95=136 p99=138 mean=125.5 withinSLO=100.00% budgetBurn=0% OK
p95 change: -2.9% → no significant change; mean change -25.5% (means hide the tail)
```

One tail request burned five times the error budget; the mean moved 25% while p95 barely moved.

INPUT 1

```
slo=300 target=99
before: 120 130 125 140 118 122 135 128
950 126 124 129 131 127 123 121 133 119
138 130
after: 118 128 122 138 116 120 132 126
130 124 122 127 129 125 121 119 131 117
136 128
```

OUTPUT 1

```
before: n=20 p50=127 p95=140 p99=950
mean=168.4 withinSLO=95.00%
budgetBurn=500% BREACH
after: n=20 p50=125 p95=136 p99=138
mean=125.5 withinSLO=100.00%
budgetBurn=0% OK
p95 change: -2.9% → no significant
change; mean change -25.5% (means hide
the tail)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sorted = (xs) => [...xs].sort((a, b) => a - b);
const percentile = (xs, p) => { const s = sorted(xs); return s[Math.min(s.length - 1
const median = (xs) => { const s = sorted(xs); const m = s.length >> 1; return s.len
const mean = (xs) => xs.reduce((a, b) => a + b, 0) / xs.length;
const cfg = Object.fromEntries(lines[0].trim().split(/\s+/).map((kv) => kv.split("='
const slo = Number(cfg.slo), target = Number(cfg.target);
const series = lines.slice(1).map((l) => { const [name, rest] = l.split(":"); return
const report = {};
for (const s of series) {
  const within = s.xs.filter((x) => x <= slo).length;
  const pct = (within / s.xs.length) * 100;
  const allowedFailures = s.xs.length * (1 - target / 100);
  const burn = allowedFailures === 0 ? Infinity : (s.xs.length - within) / allowedFa
  report[s.name] = { p50: percentile(s.xs, 50), p95: percentile(s.xs, 95), p99: perc
  console.log(`${s.name}: n=${s.xs.length} p50=${report[s.name].p50} p95=${report[s.
}
if (report.before && report.after) {
  const d = ((report.after.p95 - report.before.p95) / report.before.p95) * 100;
  console.log(`p95 change: ${d >= 0 ? "+" : ""}${d.toFixed(1)}% → ${d > 10 ? "regre
}
```

EXERCISE 2

Cache-hit simulator

Line 1 lists cache capacities; line 2 is a request trace of keys, `|` separating batches — keys inside a batch arrive **concurrently**, so a miss's load only lands after the batch. For each capacity run an LRU cache (a `Map` re-inserted on hit, oldest evicted) over the trace: a hit costs 2 ms, a miss 60 ms; a miss whose key is already loading in the same batch is **coalesced** (single-flight: it waits but starts no load). Print `capacity=<c>: hits=<> misses=<> (<n> coalesced) hitRatio=<xx.x>% backendLoads=<> avgLatency=<x.x>ms` per capacity, then `no cache: avgLatency=60ms backendLoads=<total>`.

Example: capacities `1 2 3`, trace `a a a | b | a | c | a | b | d | a | e | a | b` →

```
capacity=1: hits=0 misses=13 (2 coalesced) hitRatio=0.0% backendLoads=11 avgLatency=
capacity=2: hits=3 misses=10 (2 coalesced) hitRatio=23.1% backendLoads=8 avgLatency=
capacity=3: hits=5 misses=8 (2 coalesced) hitRatio=38.5% backendLoads=6 avgLatency=3
no cache: avgLatency=60ms backendLoads=13
```

Three concurrent requests for `a` produced one backend load; the capacity decides everything after that.

INPUT 1

```
1 2 3
a a a | b | a | c | a | b | d | a | e |
a | b
```

OUTPUT 1

```
capacity=1: hits=0 misses=13 (2
coalesced) hitRatio=0.0% backendLoads=11
avgLatency=60.0ms
capacity=2: hits=3 misses=10 (2
coalesced) hitRatio=23.1% backendLoads=8
avgLatency=46.6ms
capacity=3: hits=5 misses=8 (2
coalesced) hitRatio=38.5% backendLoads=6
avgLatency=37.7ms
no cache: avgLatency=60ms
backendLoads=13
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const capacities = lines[0].trim().split(/\s+/).map(Number);
const batches = lines[1].trim().split("|").map((b) => b.trim().split(/\s+/).filter(B
const HIT = 2, MISS = 60;   // ms
const total = batches.reduce((n, b) => n + b.length, 0);
for (const capacity of capacities) {
  const lru = new Map();
  let hits = 0, misses = 0, loads = 0, coalesced = 0, latency = 0;
  for (const batch of batches) {
    const inFlight = new Set();                              // single-flight: one l
    for (const key of batch) {
      if (lru.has(key)) { hits++; latency += HIT; const v = lru.get(key); lru.delete
      misses++; latency += MISS;                             // a request that has t
      if (inFlight.has(key)) coalesced++; else { inFlight.add(key); loads++; }   //
    }
    for (const key of inFlight) { lru.set(key, true); if (lru.size > capacity) lru.d
  }
  console.log(`capacity=${capacity}: hits=${hits} misses=${misses} (${coalesced} coa
}
console.log(`no cache: avgLatency=${MISS}ms backendLoads=${total}`);
```

QUIZ

1. Why report p99 rather than the average response time?

A. p99 is easier to compute

B. Tails are where users suffer and where fan-out compounds — a page making ten parallel calls sees the max of ten draws; averages hide both

C. Averages are always wrong

D. p99 is what load balancers use

2. Three independent database reads awaited one after another when each round trip is 500 ms…

A. Take 500 ms

B. Take 1.5 s — `Promise.all` them for one round trip; sequential awaits *are* the response time

C. Are cached automatically

D. Are batched by the driver

3. Single-flight in a cache means…

A. Only one key is cached

B. One loader runs per key at a time; concurrent misses for the same key share it, so a miss under load does not stampede the database

C. The cache has one entry

D. The cache is per request

4. Event-loop lag climbing to 200 ms at p99 indicates…

A. The database is slow

B. Synchronous CPU work is blocking the loop — requests queue behind it; move it to workers, stream large bodies, cap body sizes

C. Redis is down

D. Too many connections in the pool

5. A connection pool sized larger than the database's limit…

A. Improves throughput

B. Produces connection errors under load — size it to the limit divided by the instance count, keep queries short, add timeouts

C. Is required for transactions

D. Is harmless

ANSWER KEY · SERVER AND NETWORK PERFORMANCE — LATENCY BUDGETS, PERCENTILES, CACHING AND THE EVENT LOOP UNDER LOAD

1. B Set latency budgets at percentiles per route.

2. B N+1 — a query per item in a loop — is the same mistake at scale; batch with `IN` or a join.

3. B Cache the promise. Pair it with stale-while-revalidate and cache composed payloads.

4. B Lag is the single best health metric of a Node process.

5. B Pools and bounded queues are backpressure; unbounded buffering ends in out-of-memory.

CHECKPOINT · 24 MIN

Checkpoint — Performance and memory

This checkpoint covers V8's pipeline (interpreter, feedback, optimising compiler, deoptimisation), hidden classes and inline caches, elements kinds; benchmarking pitfalls and statistics, profiling and flame charts, complexity before constants; reachability, the generational collector, the leak catalogue and heap-snapshot diagnosis; the habits that survive measurement; and server-side latency — percentiles, round trips, caching layers, pools, event-loop lag, load testing.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module. The programs model the engine's and the collector's rules rather than timing anything — timings would differ on every machine, and the rules are what you need to reason with.

Before you start, make sure you can answer:

- What a hidden class is and what breaks sharing; monomorphic versus megamorphic; how elements kinds transition.
- Why a naive benchmark lies (warm-up, dead code, folding, GC); median versus mean; what a flame chart's width means.
- What keeps an object alive; young versus old generation; the seven leak patterns; how to read a snapshot comparison.
- The complexity table (Set for membership, no `shift` queues, no spread in `reduce`); what to batch; when to use a worker.
- Why p99 matters; the round-trip arithmetic of sequential awaits; cache-aside versus SWR; what event-loop lag indicates.

The programs are a hidden-class and inline-cache simulator that reports shapes and IC states per call site, a generational garbage-collector simulator with promotion and mark-sweep, and a request-trace analyser that computes per-route percentiles, SLO burn and N+1 suspects.

EXERCISE 1

Hidden classes and inline caches, simulated

Commands: `class <Name>: <prop> <prop> …` defines a constructor that assigns those properties in order; `new <Name> <obj>` instantiates; `set <obj> <prop>` adds a property later (a transition off the constructor's shape); `delete <obj> <prop>` drops the object to dictionary mode; `site <name> <obj> …` is a property-access site and the objects it has seen. Maintain a transition tree from `S0` where the same property added to the same shape reaches the same child. Print `transition tree:` then the tree indented two spaces per level as `S<n> {props}` (`{}` for the root); `objects: <obj>=<shape|dictionary> …`; each site as `site <name>: <state> [<shapes seen>]` where the state is `monomorphic`, `polymorphic` (2–4), `megamorphic` (more), or `megamorphic (dictionary-mode object)` when any object seen is in dictionary mode; and `shapes created=<n> uniformClasses=<classes whose live non-dictionary instances all share one shape, or →`.

Example: `Point` and `Vec` both declaring `x y`, `p2` given `z` later, `p3` with `x` deleted →

```
transition tree:
  S0 {}
    S1 {x}
      S2 {x,y}
        S3 {x,y,z}
objects: p1=S2 p2=S3 v1=S2 p3=dictionary
site readX: polymorphic [S2,S3]
site same: monomorphic [S2]
site broken: megamorphic (dictionary-mode object) [S2,dictionary]
site one: monomorphic [S2]
shapes created=3 uniformClasses=Vec
```

Two classes with the same property order share one shape; one late `set` forked `Point`'s instances apart.

INPUT 1

```
class Point: x y
class Vec: x y
new Point p1
new Point p2
new Vec v1
set p2 z
new Point p3
delete p3 x
site readX p1 p2 v1
site same p1 v1
site broken p1 p3
site one p1
```

OUTPUT 1

```
transition tree:
  S0 {}
    S1 {x}
      S2 {x,y}
        S3 {x,y,z}
objects: p1=S2 p2=S3 v1=S2 p3=dictionary
site readX: polymorphic [S2,S3]
site same: monomorphic [S2]
site broken: megamorphic (dictionary-
mode object) [S2,dictionary]
site one: monomorphic [S2]
shapes created=3 uniformClasses=Vec
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const root = { id: "S0", props: [], children: new Map() };
let counter = 1;
const all = [root];
function transition(shape, prop) {
  if (!shape.children.has(prop)) { const next = { id: `S${counter++}`, props: [...sh
  return shape.children.get(prop);
}
const classes = new Map(), objects = new Map(), sites = [];
for (const line of lines) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  if (cmd === "class") classes.set(rest[0].replace(/:$/, ""), rest.slice(1));
  else if (cmd === "new") { let s = root; for (const p of classes.get(rest[0])) s =
  else if (cmd === "set") { const o = objects.get(rest[0]); if (!o.dictionary) o.sha
  else if (cmd === "delete") { const o = objects.get(rest[0]); o.dictionary = true;
  else if (cmd === "site") sites.push({ name: rest[0], objs: rest.slice(1) });
}
function printTree(shape, depth) {
  console.log(`${"  ".repeat(depth)}${shape.id}${shape.props.length ? ` {${shape.prc
  for (const child of shape.children.values()) printTree(child, depth + 1);
}
console.log("transition tree:");
printTree(root, 1);
const shapeOf = (o) => (o.dictionary ? "dictionary" : o.shape.id);
console.log(`objects: ${[...objects].map(([n, o]) => `${n}=${shapeOf(o)}`).join(" ")
for (const site of sites) {
  const seen = [...new Set(site.objs.map((n) => shapeOf(objects.get(n))))];
  const state = seen.includes("dictionary") ? "megamorphic (dictionary-mode object)"
  console.log(`site ${site.name}: ${state} [${seen.join(",")}]`);
}
console.log(`shapes created=${all.length - 1} uniformClasses=${[...classes.keys()].f
```

EXERCISE 2

A generational garbage collector

Simulate V8's two generations. Commands: `alloc <id> <bytes>` (young, age 0); `ref <from|root> <to>`; `unref <from|root> <to>`; `scavenge` — a young-generation collection: mark from the roots **and from every old-space object** (the remembered set — a scavenge never looks inside old space, so a dead old object still keeps its young children alive), free unreachable young objects, age the survivors and promote those reaching age 2; print `scavenge #<n>: freed=<ids or -> promoted=<ids or -> young=<bytes>B old=<bytes>B`. `major` — mark from the real roots over both spaces and sweep: `major #<n>: freed=<ids or -> young=<>B old=<>B`. `stats` → `stats: objects=<> young=<>B old=<>B scavenges=<> majors=<>`.

Example: `a` (rooted) and `b` survive two scavenges and are promoted; `c` is then allocated under `b`, `a` drops `b`, and a scavenge runs →

```
scavenge #4: freed=- promoted=c young=0B old=170B
…
major #1: freed=b,c young=0B old=100B
```

`c` was unreachable from the roots yet survived and was promoted, because dead old `b` counts as a root during a scavenge. Only the major collection frees them both.

INPUT 1

```
alloc a 100
alloc b 50
alloc tmp1 10
ref root a
ref a b
scavenge
alloc tmp2 10
scavenge
alloc c 20
ref b c
scavenge
unref a b
scavenge
stats
major
stats
```

OUTPUT 1

```
scavenge #1: freed=tmp1 promoted=-
young=150B old=0B
scavenge #2: freed=tmp2 promoted=a,b
young=0B old=150B
scavenge #3: freed=- promoted=-
young=20B old=150B
scavenge #4: freed=- promoted=c young=0B
old=170B
stats: objects=3 young=0B old=170B
scavenges=4 majors=0
major #1: freed=b,c young=0B old=100B
stats: objects=1 young=0B old=100B
scavenges=4 majors=1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
// alloc <id> <size> | ref <from|root> <to> | unref <from|root> <to> | scavenge | ma
// young objects survive two scavenges then get promoted to old space; a scavenge tr
// (the remembered set — it never looks inside old space); a major collection marks
const objects = new Map();   // id → { size, refs: Set, gen: "young"|"old", age }
const roots = new Set();
let scavenges = 0, majors = 0;
function live(extraRoots = []) {
  const seen = new Set(), stack = [...roots, ...extraRoots].filter((r) => objects.ha
  while (stack.length) { const id = stack.pop(); if (seen.has(id)) continue; seen.ad
  return seen;
}
const bytes = (gen) => [...objects.values()].filter((o) => o.gen === gen).reduce((s,
for (const line of lines) {
  const [cmd, a, b] = line.trim().split(/\s+/);
  if (cmd === "alloc") objects.set(a, { size: Number(b), refs: new Set(), gen: "your
  else if (cmd === "ref") (a === "root" ? roots.add(b) : objects.get(a).refs.add(b))
  else if (cmd === "unref") (a === "root" ? roots.delete(b) : objects.get(a).refs.de
  else if (cmd === "scavenge") {
    scavenges++;
    const old = [...objects].filter(([, o]) => o.gen === "old").map(([id]) => id);
    const reachable = live(old);                                  // a dead old object
    const freed = [], promoted = [];
    for (const [id, o] of objects) {
      if (o.gen !== "young") continue;
      if (!reachable.has(id)) { freed.push(id); objects.delete(id); continue; }
      if (++o.age >= 2) { o.gen = "old"; promoted.push(id); }   // survived twice: p
    }
    console.log(`scavenge #${scavenges}: freed=${freed.join(",") || "-"} promoted=${
  } else if (cmd === "major") {
    majors++;
    const reachable = live();
    const freed = [...objects.keys()].filter((id) => !reachable.has(id));
    for (const id of freed) objects.delete(id);
    console.log(`major #${majors}: freed=${freed.join(",") || "-"} young=${bytes("yo
  } else if (cmd === "stats") console.log(`stats: objects=${objects.size} young=${by
}
```

EXERCISE 3

Request-trace analyser

Line 1 is `slo=<ms> target=<percent> queries=<limit>`; each following line is one request: `<route> <ms> <status> <queries>`. Group by route and print, sorted by p95 descending, `<route>: n=<> p50=<> p95=<> p99=<> errors=<status ≥ 500> n+1suspects=<requests over the query limit>`

`maxQueries=<>` (nearest-rank percentiles). Then `slowest by p95: <route>`; `SLO <ms>ms @ <target>%: <xx.x>% good → <met|missed>; error budget used=<n>%` where a request is good when within the SLO and not a 5xx, and budget used = bad requests ÷ allowed bad (`n × (1 - target/100)`); and the single worst offender: `N+1 candidate: <route> ran <q> queries in one request (limit <limit>) - batch them` or `N+1: none over the limit`.

Example: ten requests over `/home`, `/orders` (one 900 ms request with 45 queries, one 500) and `/search` at `slo=300 target=95 queries=10` →

```
/orders: n=4 p50=260 p95=900 p99=900 errors=1 n+1suspects=1 maxQueries=45
/home: n=3 p50=120 p95=140 p99=140 errors=0 n+1suspects=0 maxQueries=3
/search: n=3 p50=80 p95=95 p99=95 errors=0 n+1suspects=0 maxQueries=1
slowest by p95: /orders
SLO 300ms @ 95%: 80.0% good → missed; error budget used=400%
N+1 candidate: /orders ran 45 queries in one request (limit 10) - batch them
```

INPUT 1

```
slo=300 target=95 queries=10
/home 120 200 3
/home 140 200 3
/home 110 200 3
/orders 250 200 4
/orders 900 200 45
/orders 260 200 4
/orders 280 500 4
/search 80 200 1
/search 95 200 1
/search 70 200 1
```

OUTPUT 1

```
/orders: n=4 p50=260 p95=900 p99=900
errors=1 n+1suspects=1 maxQueries=45
/home: n=3 p50=120 p95=140 p99=140
errors=0 n+1suspects=0 maxQueries=3
/search: n=3 p50=80 p95=95 p99=95
errors=0 n+1suspects=0 maxQueries=1
slowest by p95: /orders
SLO 300ms @ 95%: 80.0% good → missed;
error budget used=400%
N+1 candidate: /orders ran 45 queries in
one request (limit 10) - batch them
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const sorted = (xs) => [...xs].sort((a, b) => a - b);
const percentile = (xs, p) => { const s = sorted(xs); return s[Math.min(s.length - 1
const median = (xs) => { const s = sorted(xs); const m = s.length >> 1; return s.len
const mean = (xs) => xs.reduce((a, b) => a + b, 0) / xs.length;
const cfg = Object.fromEntries(lines[0].trim().split(/\s+/).map((kv) => kv.split("="
const requests = lines.slice(1).map((l) => { const [route, ms, status, queries] = l.
const slo = Number(cfg.slo), target = Number(cfg.target), queryLimit = Number(cfg.qu
const byRoute = new Map();
for (const r of requests) (byRoute.get(r.route) ?? byRoute.set(r.route, []).get(r.ro
const rows = [...byRoute].map(([route, rs]) => {
  const ms = rs.map((r) => r.ms);
  return { route, n: rs.length, p50: percentile(ms, 50), p95: percentile(ms, 95), p9
}).sort((a, b) => b.p95 - a.p95);
for (const r of rows) console.log(`${r.route}: n=${r.n} p50=${r.p50} p95=${r.p95} p9
const within = requests.filter((r) => r.ms <= slo && r.status < 500).length;
const pct = (within / requests.length) * 100;
console.log(`slowest by p95: ${rows[0].route}`);
console.log(`SLO ${slo}ms @ ${target}%: ${pct.toFixed(1)}% good → ${pct >= target ?
const worst = requests.filter((r) => r.queries > queryLimit).sort((a, b) => b.queries
console.log(worst ? `N+1 candidate: ${worst.route} ran ${worst.queries} queries in
```

CHECKPOINT QUESTIONS

1. Code that runs once at startup…

A. Should be micro-optimised first

B. Runs in the interpreter and is never optimised — micro-optimising it is pointless; hot code is where performance lives

C. Is compiled by TurboFan

D. Cannot allocate

2. Assigning `this.z` in a method that only some callers reach…

A. Is fine

B. Forks the instances into two shapes — initialise every property in the constructor, in the same order, every time

C. Deletes `z`

D. Makes the array holey

3. `[1, 2, 3.5]` has elements kind…

A. PACKED_SMI

B. PACKED_DOUBLE — one fraction moves the whole array to doubles, permanently

C. HOLEY_ELEMENTS

D. Typed

4. A benchmark with literal inputs like `add(2, 3)` inside the loop…

A. Is the most accurate

B. Lets the compiler constant-fold the answer — feed data built at runtime instead

C. Measures GC

D. Cannot warm up

5. Two alternatives whose latency distributions overlap almost entirely are…

A. Distinguishable if the means differ

B. Not distinguishable — a 3% difference under 10% run-to-run variance is noise

C. Both wrong

D. Both optimised

6. A sampling profiler…

A. Wraps every function in timers

B. Records the stack every ~1 ms at full speed — cheap and honest; instrumentation is exact but distorts

C. Only works in browsers

D. Requires `--expose-gc`

7. A `setInterval` whose callback closes over a component that has since been destroyed…

A. Is collected with the component

B. Keeps the component and everything it references alive until `clearInterval` — a classic leak

C. Throws

D. Runs once more and stops

8. In a heap snapshot you sort by…

A. Shallow size

B. Retained size — what would actually be freed if the object went away

C. Name

D. Allocation time

9. During a scavenge, a young object referenced only by an unreachable old-space object…

A. Is freed

B. Survives — the scavenger treats old space as roots (the remembered set) and does not know the old object is dead; a major GC frees both

C. Is promoted immediately

D. Causes a deoptimisation

10. `--max-old-space-size=8192` when a server grows until it dies…

A. Fixes the leak

B. Buys time — the leak is a kept reference and the heap will reach the new limit too

C. Disables the collector

D. Enables incremental marking

11. The N+1 problem is…

A. Off-by-one indexing

B. One query per item in a loop — n round trips where one `IN` /join/batch query would do

C. A pool that is one connection short

D. A cache with one extra key

12. Load testing to find the knee means…

A. Measuring requests per second at one concurrency

B. Ramping concurrency while watching p95/p99, errors, event-loop lag and pool queues until latency starts climbing — then fixing that bottleneck and repeating

C. Testing once before launch

D. Testing only the fastest route

ANSWER KEY · CHECKPOINT — PERFORMANCE AND MEMORY

1. B Ignition first; TurboFan only for hot functions with stable feedback.

2. B Conditional adds and `delete` are the two shape killers.

3. B Typed arrays fix the representation when the data is numeric.

4. B Warm-up, dead code, folding, GC, granularity and machine noise: six ways a benchmark lies.

5. B Compare medians and spread, interleave runs, and reproduce twice.

6. B `node --cpu-prof` writes a profile DevTools can open as a flame chart.

7. B Every listener, timer and subscription needs a matching teardown or an `AbortSignal`.

8. B Compare two snapshots and follow the retainer path from a root.

9. B The generational trade: cheap frequent scavenges, occasional full marks.

10. B Find the retainer; bound the cache; remove the listener.

11. B Count the sequential awaits; response time is round trips × latency.

12. B The knee moves after each fix; production telemetry per route is the other half of measuring.

CodeKairo · JavaScript study plan · Module 15: Performance and memory. Solve the exercises online to track your progress and earn XP.