Module 1 of 16

JavaScript and its runtime

What the language is and who defines it, engines and the JIT, Node as a host, reading stdin and writing exact output, statements versus expressions and ASI, strict mode, and a first look at the event loop.

JavaScript is the language of the browser, the server and most of the tooling in between, and it looks the way it does because of five early decisions: dynamic types, first-class functions, prototypes, a single thread with an event loop, and forgiving semantics. This module makes those decisions visible before any of them can surprise you.

It covers what JavaScript, ECMAScript and V8 each are; how an engine parses, interprets and JIT-compiles code and why predictable object shapes are fast; what Node adds around the engine and how a program starts and stops; the three ways to read standard input and the habits that make output exact; statements versus expressions with the automatic-semicolon-insertion traps spelled out; strict mode; and the event loop — one stack, a task queue, a microtask queue that drains first — far enough to predict the order of logs in code with timers and promises.

Every exercise is a Node program reading stdin and printing stdout, judged on Node 16. The template from lesson 3 is the one you will reuse throughout the plan.

CONTENTS

1	What JavaScript is, and why it looks the way it does	12 min
2	Engines, runtimes and running code with Node	13 min
3	Reading input and writing output in Node	13 min
4	Statements, expressions and the semicolon rules	12 min
5	The event loop — a first look	14 min
✓	Checkpoint — JavaScript and its runtime	24 min

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

LESSON 1 · 12 MIN

What JavaScript is, and why it looks the way it does

JavaScript was written in ten days in 1995 to make web pages move, and it now runs the browser, the server (Node), the phone (React Native), the desktop (Electron) and the build tools of every other language. It is the only language every web developer must know and, by most counts, the most used programming language in the world. Its shape — C-like syntax, first-class functions from Scheme, prototype-based objects from Self, dynamic types, one thread with an event loop — explains most of what will surprise you in this plan. This lesson is the map: what the language is, who defines it, where it runs, and what a JavaScript program is at its simplest.

Three names, one language

JavaScript is the language you write. **ECMAScript** is its specification, maintained by the TC39 committee and published yearly (ES2015 — also called ES6 — was the big modernisation; ES2016 onwards adds a few features each year). **ECMA-262** is the document's number. The names "ES6", "ES2020", "ESNext" all refer to editions of the spec; engines implement the spec, and features move through **four TC39 stages** (proposal → draft → candidate → finished) before landing. Nothing about the language is owned by one company, which is why it evolves slowly and never breaks old code — a page written in 1998 still runs.

The design in five decisions

Dynamic typing. Variables have no type; values do. `let x = 1; x = "one";` is legal. The trade-off is flexibility against the runtime errors a compiler would have caught — which is why TypeScript exists and why this plan spends a whole module on values and coercion.

First-class functions and closures. Functions are values you can pass, return and store; an inner function remembers the variables around it. This is the heart of the language — callbacks, event handlers, promises, React components are all closures.

Prototypes, not classes. Objects inherit directly from other objects through a prototype chain; the `class` keyword (2015) is a cleaner syntax over the same mechanism.

Single-threaded with an event loop. One thread runs your code; I/O happens elsewhere and calls back when done. No data races, no locks — and a program that blocks that one thread freezes everything.

Forgiving semantics. Missing semicolons are inserted, `"5" * 2` is `10`, undefined properties are `undefined` rather than errors. Convenient for a ten-line script, treacherous in a ten-thousand-line application; the rules for staying safe are a recurring theme.

Where it runs

Browsers — the original host. The language plus the DOM, `fetch`, `localStorage` and the rest of the Web APIs.

Node.js (2009) — Chrome's V8 engine with file, network and process APIs instead of a DOM. Servers, CLIs, build tools. The runtime this plan judges on: **Node 16**.

Deno and **Bun** — newer server runtimes with the same language and different standard libraries.

Embedded — in databases, editors, game engines, and as the scripting layer of countless applications.

The *language* is the same everywhere; the *host objects* differ. `console.log` exists in all of them; `document` exists only in a browser; `require("fs")` only in Node.

Your first program

```
// hello.js
"use strict";
const name = "world";
console.log(`Hello, ${name}!`);
```

Run it with `node hello.js`. There is no class, no `main`, no compilation step you see — the file is the program, executed top to bottom. `"use strict"` at the top opts into **strict mode**, which turns several silent mistakes into errors (assigning to an undeclared variable, duplicate parameter names, writing to read-only properties); modules and classes are strict automatically, and every program in this plan should start with it.

Reading input, the judge's way

Exercises here read standard input and print to standard output, exactly like any other judged language:

```
const input = require("fs").readFileSync(0, "utf8");     // everything on std
const tokens = input.trim().split(/\s+/);                 // whitespace-separ
const n = Number(tokens[0]);
console.log(n * 2);
```

Lesson 3 covers the alternatives; this one line is enough for now.

Versions you will hear about

Edition	Year	Brought
ES5	2009	strict mode, JSON, `Array.prototype.map/filter/reduce`
ES2015 (ES6)	2015	`let` / `const`, arrows, classes, template literals, destructuring, promises, modules, `Map` / `Set`
ES2017	2017	`async` / `await`
ES2019–2020		`flat`, optional catch binding, optional chaining `?.`, nullish `??`, `BigInt`
ES2021–2022		`replaceAll`, logical assignment, class fields, `at()`, top-level `await`

Node 16 supports everything through ES2022. Newer features (`toSorted`, `structuredClone`, the `using` declaration) are noted where they matter but not exercised.

Interview angle

"Is JavaScript compiled or interpreted?" Both — engines parse to bytecode, interpret, and JIT-compile hot code to machine code (next lesson).

"What is ECMAScript?" The specification JavaScript implements; editions are yearly.

"Why is JavaScript single-threaded?" By design: one call stack, an event loop for I/O; simple concurrency model, no races.

"What does `"use strict"` do?" Turns silent errors into thrown errors and removes some unsafe behaviours; on by default in modules and classes.

"Node versus browser?" Same language and engine family; different host APIs (files and processes versus DOM and `fetch`).

Key takeaways

JavaScript = the language; ECMAScript = the yearly spec from TC39; engines implement it and never break old code.

Dynamic types, first-class functions and closures, prototypes, one thread plus an event loop, forgiving coercion.

Same language everywhere; host APIs differ. This plan runs on Node 16.

A file is a program; `"use strict"` at the top; `readFileSync(0)` reads all of stdin.

ES2015 is the modern baseline; know what each later edition added.

EXERCISE 1

Hello, strictly

Write a strict-mode program that reads an integer `n` and `n` names, prints `Hello, <name>!` for each using a template literal, then `Greeted <n> person` or `Greeted <n> people`. Use `const` for anything that does not change and `let` only for the loop counter.

Example: `2` then `Ada Linus` →

```
Hello, Ada!
Hello, Linus!
Greeted 2 people
```

INPUT 1

```
2
Ada Linus
```

OUTPUT 1

```
Hello, Ada!
Hello, Linus!
Greeted 2 people
```

INPUT 2

```
1
Grace
```

OUTPUT 2

```
Hello, Grace!
Greeted 1 person
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
for (let i = 1; i ⩽ n; i++) {
  console.log(`Hello, ${tokens[i]}!`);
}
console.log(`Greeted ${n} ${n === 1 ? "person" : "people"}`);
```

EXERCISE 2

Which edition?

Build a `Map` from feature name to the ECMAScript year that shipped it: `let-const` 2015, `arrow-functions` 2015, `promises` 2015, `async-await` 2017, `flat` 2019, `optional-chaining` 2020, `nullish-coalescing` 2020, `bigint` 2020, `replaceAll` 2021, `class-fields` 2022, `at` 2022, `top-level-await` 2022. Read `n` feature names and print `<feature>: ES<year>` or `<feature>: unknown`, then `es6=<how many of the queried features are from 2015>`.

Example: `3` then `arrow-functions at jsx` →

```
arrow-functions: ES2015
at: ES2022
jsx: unknown
es6=1
```

INPUT 1

```
3
arrow-functions at jsx
```

OUTPUT 1

```
arrow-functions: ES2015
at: ES2022
jsx: unknown
es6=1
```

INPUT 2

```
2
promises let-const
```

OUTPUT 2

```
promises: ES2015
let-const: ES2015
es6=2
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const editions = new Map([
  ["let-const", 2015], ["arrow-functions", 2015], ["promises", 2015], ["async-await'
  ["flat", 2019], ["optional-chaining", 2020], ["nullish-coalescing", 2020], ["bigir
  ["replaceAll", 2021], ["class-fields", 2022], ["at", 2022], ["top-level-await", 20
]);
const n = Number(tokens[0]);
let es6 = 0;
for (let i = 1; i ⩽ n; i++) {
  const feature = tokens[i];
  const year = editions.get(feature);
  console.log(year === undefined ? `${feature}: unknown` : `${feature}: ES${year}`);
  if (year === 2015) es6++;
}
console.log(`es6=${es6}`);
```

QUIZ

1. JavaScript, ECMAScript and V8 are, respectively...

A. Three names for the same thing

B. The language, its specification, and an engine that implements it

C. A browser, a server, a compiler

D. Old, new and future versions

2. Which design decision explains why JavaScript needs no locks for shared state?

A. Dynamic typing

B. One thread with an event loop

C. Prototypes

D. Automatic semicolon insertion

3. `"use strict"` at the top of a file…

A. Makes the code faster only

B. Turns silent mistakes — such as assigning to an undeclared name — into thrown errors

C. Disables `var`

D. Is required by Node

4. Which host object exists in Node but not in a browser?

A. `console`

B. `Promise`

C. `process`

D. `Math`

5. ES2015 (ES6) introduced…

A. `async` / `await`

B. `let` / `const`, arrows, classes, promises, destructuring and modules

C. Optional chaining

D. Class fields

ANSWER KEY · WHAT JAVASCRIPT IS, AND WHY IT LOOKS THE WAY IT DOES

1. B TC39 publishes ECMAScript yearly; V8 (Chrome, Node), SpiderMonkey and JavaScriptCore implement it.

2. B A task runs to completion before any other callback runs, so two callbacks never interleave.

3. B Modules and class bodies are strict automatically; scripts must opt in.

4. C `console` and the language built-ins are everywhere; `process`, `require` and `Buffer` are Node's; `document` and `window` are the browser's.

5. B `async` / `await` came in 2017, `?.` and `??` in 2020, class fields in 2022.

LESSON 2 · 13 MIN

Engines, runtimes and running code with Node

"JavaScript is slow" was true in 2005 and false by 2010, and the reason is the **engine**: the program inside every browser and inside Node that turns your source into machine code while it runs. Understanding roughly how V8 does that — and what Node adds around it — explains why some code is fast, why `node --version` matters, what `process` and `console` really are, and how a script starts and stops. This lesson is the runtime tour: engine, host, and the command line.

The engine

An engine takes source text and executes it. The three that matter: **V8** (Chrome, Edge, Node, Deno), **SpiderMonkey** (Firefox) and **JavaScriptCore** (Safari, Bun). All three follow the same pipeline:

Parse the source into an abstract syntax tree; report syntax errors here, before anything runs.

Compile to bytecode (V8's *Ignition*) and start **interpreting** it immediately — fast start-up.

Profile while running: which functions are hot, what types flow through them.

JIT-compile hot functions to optimised machine code (V8's *TurboFan*), speculating on the observed types — an `add(a, b)` that has only ever seen numbers gets a machine-code integer add.

Deoptimise if a speculation fails (a string arrives), falling back to bytecode and recompiling later.

Two consequences you can see in your own code: **monomorphic** code — functions that always receive the same shapes of object — is much faster than polymorphic code, and objects created with the same properties in the same order share a *hidden class* and stay fast. "Write predictable, boring code" is performance advice as well as style advice.

Memory is managed by a **generational garbage collector** (young objects in a nursery collected often, survivors promoted), the same idea as every other managed runtime; you never free memory by hand, and a leak is an object still reachable from something long-lived — a global, a closure, an event listener.

The runtime: Node around V8

V8 knows nothing about files, sockets or timers — the *host* provides them. Node wraps V8 with **libuv** (the event loop and asynchronous I/O), and a standard library of modules: `fs`, `path`, `http`, `crypto`, `os`, `child_process`, `readline`, `events`, `stream`. It also provides the globals you meet immediately:

- `console` — `log`, `error` (to stderr), `table`, `time` / `timeEnd`, `dir`.
- `process` — `argv` (command-line arguments), `env` (environment variables), `exit(code)`, `stdin` / `stdout` / `stderr`, `version`, `platform`, `cwd()`, `hrtime`.
- `setTimeout`, `setInterval`, `setImmediate`, `queueMicrotask` — the timer and scheduling functions (lesson 5).
- `require` / `module` / `__dirname` in CommonJS files; `import` / `export` in ES modules.
- `Buffer`, `URL`, `TextEncoder`, `fetch` (18+), `structuredClone` (17+) — the web-flavoured extras.

In the browser the same slot is filled by `window` / `document` / `fetch` / `localStorage`. Code that touches none of the host objects — pure logic — runs unchanged in both.

Running code

```
node hello.js              # run a file
node -e "console.log(1+1)" # run a one-liner
node                       # start the REPL: type expressions, see results, .e
node --version             # v16.17.1 on this plan's judge
node --check hello.js      # parse only: catch syntax errors without running
```

Node reads the file, wraps it (in CommonJS) in a function that provides `require`, `module`, `exports`, `__filename` and `__dirname`, runs it top to bottom, then keeps the process alive **as long as anything is pending** — a timer, an open socket, an unfinished read. A script with no pending work exits when the last statement finishes; `process.exit(1)` ends it immediately with a non-zero exit code (the convention for "failed"), and `process.exitCode = 1` sets the code while letting pending work finish.

Versions and releases

Node ships a new major every six months; even-numbered majors become **LTS** (long-term support) in October and are maintained for 30 months — 16, 18, 20, 22. Projects pin a version with `.nvmrc` or `engines` in `package.json`, and switch with **nvm** or **fnm**. The version decides which ECMAScript features and which built-ins exist: Node 16 has class fields and `at()`, not `structuredClone` or `Array.prototype.toSorted`. When an exercise here says "not on this runtime", that is why.

Errors you will meet on day one

`SyntaxError: Unexpected token` at start-up — the parse step failed; the line number is exact, the cause is usually a bracket or a stray comma.

`ReferenceError: x is not defined` — you used a name that was never declared (a typo, or a browser global in Node).

`TypeError: undefined is not a function` / `Cannot read properties of undefined (reading 'x')` — you called or dereferenced something that is not there; the most common runtime error in the language.

`Error: Cannot find module './x'` — a `require` path or a missing `npm install`.

Every uncaught error prints a **stack trace**: the message, then one line per frame from the point of failure outward, each with `file:line:column`. Read the top frame first; the first frame in *your* code is where to look.

The ecosystem, in one paragraph

npm (bundled with Node) is the package manager and registry — the largest in any language. `npm init` creates `package.json`, `npm install lodash` adds a dependency to `node_modules/` and records it, `npx` runs a package's binary without installing it globally. You will meet **TypeScript** (types on top of JavaScript), **ESLint** and **Prettier** (linting and formatting), **Jest**/**Vitest** (tests), and bundlers (**Vite**, **esbuild**) in any real project; none is needed for this plan, where every exercise is one file on plain Node.

Interview angle

"How does V8 execute JavaScript?" Parse → bytecode interpreted by Ignition → profiling → TurboFan JIT for hot code, with deoptimisation on failed type speculation.

"What is Node.js?" V8 plus libuv (event loop, async I/O) plus a standard library, running JavaScript outside the browser.

"What is `process`?" Node's global for the running process: arguments, environment, stdio, exit.

"Why do objects with the same property order run faster?" They share a hidden class; the JIT emits monomorphic, inline-cached code.

"When does a Node program exit?" When the event loop has nothing pending — or on `process.exit`.

Key takeaways

Engines parse, interpret, profile and JIT-compile; predictable shapes and types are fast.

Node = V8 + libuv + standard modules; `console`, `process`, timers and `require` are host globals.

`node file.js`, `-e`, the REPL, `--check`; the process lives while work is pending; exit codes signal failure.

Even majors are LTS; the version fixes the available features — this judge is Node 16.

Read a stack trace from the top; the four day-one errors each have one usual cause.

EXERCISE 1

Language, Node or browser?

Read `n` global names and classify each: `language` for built-ins the specification defines (`Array`, `Math`, `JSON`, `Promise`, `Map`, `Object`, `Number`, `String`), `node` for Node's host globals (`process`, `require`, `Buffer`, `__dirname`, `module`), `browser` for the browser's (`document`, `window`, `localStorage`, `navigator`, `alert`), `both` for globals every host provides (`console`, `setTimeout`, `setInterval`, `queueMicrotask`), and `unknown` otherwise. Print `<name>: <class>`.

Example: `4` then `process document Math console` →

```
process: node
document: browser
Math: language
console: both
```

INPUT 1

```
4
process document Math console
```

OUTPUT 1

```
process: node
document: browser
Math: language
console: both
```

INPUT 2

```
2
Buffer fetch
```

OUTPUT 2

```
Buffer: node
fetch: unknown
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const groups = {
  language: ["Array", "Math", "JSON", "Promise", "Map", "Object", "Number", "String'
  node: ["process", "require", "Buffer", "__dirname", "module"],
  browser: ["document", "window", "localStorage", "navigator", "alert"],
  both: ["console", "setTimeout", "setInterval", "queueMicrotask"],
};
const n = Number(tokens[0]);
for (let i = 1; i ⩽ n; i++) {
  const name = tokens[i];
  const found = Object.entries(groups).find(([, names]) ⇒ names.includes(name));
  console.log(`${name}: ${found ? found[0] : "unknown"}`);
}
```

EXERCISE 2

Read the stack trace

Read a Node stack trace from stdin (until end of input). The first line is `<ErrorName>: <message>`; each following line is a frame of the form `at <function> (<file>:<line>:<col>)` or `at <file>:<line>:<col>`. Print `error=<ErrorName>`, `message=<message>`, `frames=<count>`, and `user=<file>:<line>` for the **first** frame whose file does not start with `node:` (or `user=none`).

Example input

```
TypeError: Cannot read properties of undefined (reading 'x')
    at total (/workspace/main.js:12:18)
    at Module._compile (node:internal/modules/cjs/loader:1126:14)
```

→

```
error=TypeError
message=Cannot read properties of undefined (reading 'x')
frames=2
user=/workspace/main.js:12
```

INPUT 1

```
TypeError: Cannot read properties of
undefined (reading 'x')
    at total (/workspace/main.js:12:18)
    at Module._compile
(node:internal/modules/cjs/loader:1126:14)
```

OUTPUT 1

```
error=TypeError
message=Cannot read properties of
undefined (reading 'x')
frames=2
user=/workspace/main.js:12
```

INPUT 2

```
ReferenceError: x is not defined
    at
node:internal/main/run_main_module:17:47
```

OUTPUT 2

```
error=ReferenceError
message=x is not defined
frames=1
user=none
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim
const [head, ...frames] = lines;
const colon = head.indexOf(": ");
console.log(`error=${head.slice(0, colon)}`);
console.log(`message=${head.slice(colon + 2)}`);
console.log(`frames=${frames.length}`);
let user = "none";
for (const frame of frames) {
  const m = frame.match(/\(?([^\s()]+):(\d+):\d+\)?\s*$/);
  if (m && !m[1].startsWith("node:")) {
    user = `${m[1]}:${m[2]}`;
    break;
  }
}
console.log(`user=${user}`);
```

QUIZ

1. V8 makes hot code fast by…

A. Interpreting it twice

B. Profiling types at run time and JIT-compiling to machine code, deoptimising if a speculation fails

C. Caching the source file

D. Running it on a second thread

2. Node.js is…

A. A different language from browser JavaScript

B. The V8 engine plus libuv (event loop and async I/O) plus a standard library, outside the browser

C. A browser without a window

D. A JavaScript compiler

3. A Node process exits when…

A. The last line of the file runs

B. The event loop has nothing pending — no timers, sockets or reads — or `process.exit` is called

C. `return` is executed at top level

D. Memory runs out

4. Objects created with the same properties in the same order are faster because…

A. They use less memory only

B. They share a hidden class, so the JIT can emit monomorphic, inline-cached property access

C. V8 sorts them

D. They are frozen

5. `TypeError: Cannot read properties of undefined (reading 'x')` means…

A. `x` was never declared

B. You accessed `.x` on a value that is `undefined`

C. `x` is a reserved word

D. A syntax error

ANSWER KEY · ENGINES, RUNTIMES AND RUNNING CODE WITH NODE

1. B Ignition interprets bytecode; TurboFan compiles hot, type-stable functions.

2. B Same language and engine; different host APIs.

3. B A server stays alive because its listening socket is pending work.

4. B Predictable shapes are the single most effective performance habit in engine-level terms.

5. B A `ReferenceError` is the undeclared-name case; this one is the most common runtime error in JavaScript.

LESSON 3 · 13 MIN

Reading input and writing output in Node

Every exercise in this plan is a program that reads standard input and prints to standard output — the same contract as any coding judge. Node has three ways to read stdin, two ways to write stdout, and a handful of formatting habits that keep the output exactly right. This lesson gives you the template you will use ninety times and explains the alternatives so you recognise them in other people's code.

The template: read everything at once

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");   // file descriptor 0 is
const lines = input.split("\n");                        // by line…
const tokens = input.trim().split(/\s+/);               // …or by whitespace

let pos = 0;
const next = () => tokens[pos++];                       // a cursor over the
const nextInt = () => Number(next());

const n = nextInt();
let sum = 0;
for (let i = 0; i < n; i++) sum += nextInt();
console.log(sum);
```

`readFileSync(0, "utf8")` blocks until stdin is closed and returns the whole text — fine for anything under a few tens of megabytes, which is every judge. `trim()` removes the trailing newline so the last token is not an empty string; `split(/\s+/)` splits on any run of whitespace, so the input's line structure does not matter. Watch the one edge case: `"".split(/\s+/)` gives `[""]`, not `[]`, so check for empty input before parsing.

`Number(token)` converts a numeric string; `parseInt(token, 10)` stops at the first non-digit (useful for `"12px"`, dangerous for `"1e3"`); `+token` is the terse form of `Number`. For numbers beyond 2^{53} use `BigInt(token)` — module 2 explains why.

Line by line: `readline`

When input arrives over time (an interactive tool, a pipe that never closes), the event-based reader is the tool:

```
const rl = require("readline").createInterface({ input: process.stdin });
const lines = [];
rl.on("line", (line) => lines.push(line));
rl.on("close", () => {
  // all input has arrived; solve here
  console.log(lines.length);
});
```

Nothing after `createInterface` waits — the callbacks run later, when lines arrive, and `close` fires at end of input. This is the asynchronous style the whole runtime is built on (lesson 5 and module 8), and the reason the *solve* code lives inside the `close` handler: outside it, `lines` is still empty. For judged exercises, the synchronous template is simpler and never wrong.

Streams, for completeness

```
let data = "";
process.stdin.setEncoding("utf8");
process.stdin.on("data", (chunk) => (data += chunk));
process.stdin.on("end", () => solve(data));
```

Chunks arrive in arbitrary sizes; only at `end` do you have the whole input. Same shape as `readline`, lower level. You will see it in older competitive code.

Writing output

`console.log(a, b, c)` prints its arguments separated by spaces with a newline, formatting objects readably (`{ x: 1 }`, `[ 1, 2 ]`) — good for humans, wrong for judges that want exact text. For exact output, build strings yourself:

```
console.log(`${name} scored ${score}`);          // template literal
console.log([1, 2, 3].join(" "));                 // "1 2 3", not "[ 1, 2, 3
console.log(String(x));                           // explicit conversion
process.stdout.write("no newline");               // raw write, you add "\n"
```

Performance: `console.log` in a loop of 10^5 lines is slow — each call is a synchronous write. Collect lines in an array and print once: `console.log(out.join("\n"))`. That single change is the difference between "time limit exceeded" and passing on large outputs.

`console.error` writes to **stderr**, which a judge ignores — use it for debugging without corrupting the answer.

Formatting numbers and text

Need	Write
two decimals	`x.toFixed(2)` → `"3.14"` (a **string**, rounded half away from zero… mostly — see module 2)
padding	`String(n).padStart(3, "0")` → `"007"`; `name.padEnd(10)`
thousands separators	`n.toLocaleString("en-US")` → `"1,234,567"`
integer division	`Math.floor(a / b)` or `Math.trunc(a / b)` — `/` is always floating point
booleans	`String(true)` → `"true"`
join with separator	`arr.join(", ")`
repeat	`"-".repeat(20)`

`toFixed` and `toLocaleString` return strings; arithmetic on them concatenates. Convert back with `Number(...)` if you must compute further — or, better, keep numbers as numbers until the final print.

Arguments and environment

`process.argv` is `[nodePath, scriptPath, ...userArgs]`; `process.argv.slice(2)` is what the user typed after the script name. `process.env.HOME` and friends are environment variables (strings or `undefined`). Judges pass nothing through either, so exercises here use stdin only — but a CLI you write yourself will use both.

Exit codes

A script that runs to completion exits `0`. An uncaught exception exits `1` and prints the stack trace to stderr. `process.exit(2)` ends immediately with code 2 — and skips anything still buffered for stdout in some cases, so prefer `process.exitCode = 2` and let the program end naturally.

A checklist for judged output

- Read once with `readFileSync(0, "utf8")`; `trim()` then split.
- Handle empty input if the statement allows it.
- Convert tokens with `Number`; use `BigInt` past 2^{53}.
- Build exact strings; join arrays yourself; never `console.log` an object or array directly.
- Collect many lines and print once.
- Debug to `console.error`.

Interview angle

- *"How do you read stdin in Node?"* `fs.readFileSync(0, "utf8")` for everything at once; `readline` or the `process.stdin` stream for incremental input.
- *"What does `console.log([1, 2])` print?"* `[ 1, 2 ]` — a human format; join for exact output.
- *"Why is `console.log` in a loop slow?"* A synchronous write per call; batch into one string.
- *"`Number("12px")` versus `parseInt("12px")`?"* `NaN` versus `12` — `Number` needs the whole string to be numeric.
- *"How does a Node process signal failure?"* A non-zero exit code: uncaught error → 1, or `process.exitCode`.

Key takeaways

- Template: `readFileSync(0, "utf8")`, `trim`, `split(/\s+/)`, a token cursor, `Number`.
- `readline`/streams deliver input in callbacks — solve inside the `close`/`end` handler.

Exact output means strings you built; `join`, `toFixed`, `padStart`; one big `console.log` for large outputs.

`console.error` for debugging; exit codes via `process.exitCode`.

Empty input and `"".split(...)` are the edge cases.

EXERCISE 1

A cursor over ragged input

Read an integer `n` followed by `n` integers that may be spread over any number of lines with any spacing. Use the token-cursor pattern (`readFileSync(0)`, `trim`, `split(/\s+/)`, a `next()` helper). Print `sum=<s> min=<lo> max=<hi>`, or `empty` when `n` is 0.

Example (three lines): `5`, `3 9`, `-2 7 1` →

```
sum=18 min=-2 max=9
```

INPUT 1

```
5
3    9
-2 7 1
```

OUTPUT 1

```
sum=18 min=-2 max=9
```

INPUT 2

```
0
```

OUTPUT 2

```
empty
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
let pos = 0;
const next = () => tokens[pos++];
const nextInt = () => Number(next());

const n = nextInt();
if (n === 0) {
  console.log("empty");
} else {
  let sum = 0, min = Infinity, max = -Infinity;
  for (let i = 0; i < n; i++) {
    const x = nextInt();
    sum += x;
    min = Math.min(min, x);
    max = Math.max(max, x);
  }
  console.log(`sum=${sum} min=${min} max=${max}`);
}
```

EXERCISE 2

Exact columns

Read an integer `n` and `n` lines `<name> <score>` (scores are decimals). Print a table: each row is the name left-aligned in 10 characters (`padEnd`) followed by the score with two decimals right-aligned in 8 (`toFixed(2).padStart(8)`); then a rule of 18 dashes; then `TOTAL` formatted the same way with the sum. Collect every line in an array and print once with `join("\n")` .

Example: `2` then `ada 91.5` , `bob 78` →

```
ada          91.50
bob          78.00
------------------
TOTAL       169.50
```

INPUT 1

```
2
ada 91.5
bob 78
```

OUTPUT 1

```
ada          91.50
bob          78.00
------------------
TOTAL       169.50
```

INPUT 2

```
1
zed 0
```

OUTPUT 2

```
zed           0.00
------------------
TOTAL         0.00
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
let pos = 0;
const next = () => tokens[pos++];
const n = Number(next());
const out = [];
let total = 0;
for (let i = 0; i < n; i++) {
  const name = next();
  const score = Number(next());
  total += score;
  out.push(name.padEnd(10) + score.toFixed(2).padStart(8));
}
out.push("-".repeat(18));
out.push("TOTAL".padEnd(10) + total.toFixed(2).padStart(8));
console.log(out.join("\n"));
```

QUIZ

1. `require("fs").readFileSync(0, "utf8")` …

A. Reads the first line only

B. Reads all of standard input as one string, blocking until it is closed

C. Reads a file named `0`

D. Returns a Buffer of bytes

2. `"".trim().split(/\s+/)` evaluates to…

A. `[]`

B. `[""]` — one empty string, so empty input needs a check

C. `undefined`

D. Throws

3. `console.log([1, 2, 3])` prints…

A. `1 2 3`

B. `[ 1, 2, 3 ]` — a human-readable format; use `join` for exact output

C. `1,2,3`

D. `[1,2,3]`

4. Printing 100 000 lines fastest is done by…

A. `console.log` in a loop

B. Pushing into an array and one `console.log(out.join("\n"))`

C. `process.stdout.write` per line

D. `alert`

5. `Number("12px")` and `parseInt("12px", 10)` are…

A. `12` and `12`

B. `NaN` and `12`

C. `12` and `NaN`

D. Both throw

ANSWER KEY · READING INPUT AND WRITING OUTPUT IN NODE

1. B File descriptor 0 is stdin; the encoding argument makes it a string rather than a Buffer.

2. B `split` always yields at least one element.

3. B Node's inspector formatting is for people, not judges.

4. B Each `console.log` is a synchronous write; one big write is orders of magnitude faster.

5. B `Number` requires the whole string to be numeric; `parseInt` reads a leading integer and stops.

LESSON 4 · 12 MIN

Statements, expressions and the semicolon rules

A JavaScript program is a sequence of statements, and most statements are built from expressions — but the two are not the same thing, and the language's most notorious quirk, **automatic semicolon insertion**, lives exactly on their boundary. This lesson covers the anatomy of a program: what counts as a statement, what counts as an expression, how blocks and comments work, the rules by which the parser inserts semicolons you did not write, the three places that rule bites, and the conventions (strict mode, naming, formatting) that keep code readable across a team.

Expressions produce values; statements do things

An **expression** evaluates to a value: `2 + 3`, `x`, `f(y)`, `a ? b : c`, `[1, 2]`, `() ⇒ 1`, `x = 5` (assignment is an expression too — its value is the assigned value). A **statement** performs an action and produces nothing: `let x = 5;`, `if (...) {...}`, `for (...)`, `return x;`, `throw e;`. An expression can stand alone as an *expression statement* — `f(y);` — but a statement cannot appear where a value is needed: `const a = if (x) 1 else 2;` is a syntax error, which is what the conditional operator `x ? 1 : 2` is for.

This distinction decides where arrow bodies need braces (`x ⇒ x * 2` is an expression body, `x ⇒ { return x * 2; }` a block body) and why `console.log(let y = 1)` fails.

Blocks, scope and declarations

Braces group statements into a **block**; `let` and `const` declared inside are visible only there (module 2). A lone block is legal and occasionally useful for scoping a temporary name. Declarations come in five kinds — `let`, `const`, `var`, `function`, `class` — and `const` is the default: a binding that cannot be reassigned catches a class of bugs for free. Reach for `let` when the variable must change, and never for `var`.

Comments and strict mode

`// to end of line` and `/* block */`. There is no nesting of block comments. A JSDoc comment `/** … */` above a function is what editors read for hover documentation and is worth writing for anything public.

`"use strict";` as the first statement of a file (or function) enables strict mode: assigning to an undeclared name throws `ReferenceError` instead of creating a global; assigning to a read-only property throws; `this` in a plain function call is `undefined` rather than the global object; duplicate parameter names and octal literals are errors; `delete` of an undeletable property throws. ES modules and class bodies are strict automatically. There is no reason to write sloppy-mode code in this decade.

Automatic semicolon insertion (ASI)

Semicolons end statements, and the parser inserts one when it *must* to make sense of a line break. The rule, roughly: if the next token cannot continue the current statement and there is a newline before it, a semicolon is inserted; also before a closing `}` and at end of input. Three consequences:

1. `return` on its own line returns `undefined`.

```
function f() {
  return          // ASI inserts a semicolon here
    { ok: true }; // this object is an unreachable block
}
f();              // undefined
```

The same applies to `break`, `continue`, `throw` (`throw` alone is a syntax error) and `yield`. Keep the value on the same line as `return`.

2. A line starting with `(`, `[`, `` ` ``, `+`, `-` or `/` continues the previous line.

```
const a = b
(c || d).run()     // parsed as: const a = b(c || d).run()  — calling b!
const x = 1
[1, 2].forEach(f)  // parsed as: 1[1, 2].forEach(f)
```

If you omit semicolons, start such lines with a defensive `;` — `;(c || d).run()` — which is the convention in semicolon-free codebases. If you write semicolons, this never happens.

3. `++` / `--` bind to the token on their own line.

```
a
++b   // parsed as: a; ++b;  — not a++
```

The pragmatic rule: **pick one style and let Prettier enforce it.** With semicolons, the only trap left is the `return` one; without, the leading-bracket rule. Either is fine; mixing is not.

Identifiers and naming

Names are case-sensitive, may contain letters, digits, `_` and `$`, and may not start with a digit; Unicode letters are allowed but rarely wise. Conventions: `camelCase` for variables and functions, `PascalCase` for classes and constructors, `SCREAMING_SNAKE` for true constants (`MAX_RETRIES`), a leading `_` to *hint* privacy (real privacy is `#field`, module 5). Reserved words (`class`, `delete`, `new`, `typeof`, `await` in modules…) cannot be identifiers but can be property names: `obj.class` is legal, `const class = 1` is not.

Operators worth naming now

`typeof x` (a string naming the type), `x ?? y` (nullish coalescing — `y` only when `x` is `null` or `undefined`), `a?.b` (optional chaining — `undefined` instead of a throw when `a` is nullish), `...spread`, the comma operator (avoid), `void expr` (evaluates and gives `undefined` — mostly for arrow functions that must not return a value). All get their own treatment later; recognise them when they appear.

Formatting that a team can live with

Two-space indentation, one statement per line, braces on every `if` / `for` body even when it is one line (the dangling-else and "I added a second statement" bugs are both eliminated), `===` everywhere, `const` first. Run **Prettier** so no one argues, and **ESLint** to catch `no-undef`, `no-unused-vars`, `eqeqeq` and the ASI hazards mechanically. In an interview, the visible habits — `const`, `===`, braces, meaningful names — are what an interviewer reads as competence before the algorithm even starts.

Interview angle

"Expression versus statement?" Expressions evaluate to a value and can be nested; statements perform actions and cannot appear where a value is expected.

"What is ASI and when does it bite?" The parser's semicolon insertion at line breaks; `return` followed by a newline, and lines starting with `(` or `[`.

"What does strict mode change?" Undeclared assignment throws, `this` is `undefined` in plain calls, several silent failures become errors; on by default in modules.

"`let` versus `const`?" Both block-scoped; `const` forbids reassignment (not mutation) — default to `const`.

"Can a reserved word be a property name?" Yes (`obj.class`); it just cannot be a variable name.

Key takeaways

Expressions have values; statements do not. Arrow expression bodies, the conditional operator and assignment-as-expression follow from this.

`const` by default, `let` when needed, `var` never; `"use strict"` always (automatic in modules).

ASI: keep `return` values on the same line; with no semicolons, prefix lines starting with `(` / `[` / `` ` `` with `;`.

camelCase / PascalCase / SCREAMING_SNAKE; reserved words are fine as property names.

Prettier + ESLint, braces on every body, `===` — the habits interviewers notice first.

EXERCISE 1

Grades, as expressions

Read `n` integer scores. Write `const grade = (s) ⇒ …` with an **expression body** — a chain of conditional operators, no `if`, no braces: 90 and above `A`, 75–89 `B`, 50–74 `C`, otherwise `F`. Print `<score> <grade>` per line, then `passed=<count of A, B and C>` computed with `filter` and an arrow expression body.

Example: `4` then `95 80 50 20` →

```
95 A
80 B
50 C
20 F
passed=3
```

INPUT 1

```
4
95 80 50 20
```

OUTPUT 1

```
95 A
80 B
50 C
20 F
passed=3
```

INPUT 2

```
3
90 75 49
```

OUTPUT 2

```
90 A
75 B
49 F
passed=2
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
const scores = tokens.slice(1, n + 1).map(Number);
const grade = (s) ⇒ (s ⩾ 90 ? "A" : s ⩾ 75 ? "B" : s ⩾ 50 ? "C" : "F");
for (const s of scores) console.log(`${s} ${grade(s)}`);
const passed = scores.filter((s) ⇒ grade(s) ≠= "F").length;
console.log(`passed=${passed}`);
```

EXERCISE 2

What strict mode catches

Read `n` names of mistakes and, for each, perform it inside `try` / `catch` in strict mode, printing `<name>: <error class name>` or `<name>: no error`: `undeclared` (assign to a name that was never declared), `frozen` (assign a property on `Object.freeze({})`), `delete-builtin` (`delete Object.prototype`), `this-plain-call` (call a plain function that returns `this` — print `this-plain-call: undefined` when it is, else `this-plain-call: defined`), `nan-math` (`Math.sqrt(-1)` — no error, print as such).

Example: `3` then `undeclared frozen nan-math` →

```
undeclared: ReferenceError
frozen: TypeError
nan-math: no error
```

INPUT 1

```
3
undeclared frozen nan-math
```

OUTPUT 1

```
undeclared: ReferenceError
frozen: TypeError
nan-math: no error
```

INPUT 2

```
2
this-plain-call delete-builtin
```

OUTPUT 2

```
this-plain-call: undefined
delete-builtin: TypeError
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);

function attempt(name) {
  try {
    switch (name) {
      case "undeclared": {
        // eslint-disable-next-line no-undef
        neverDeclared = 1;
        return "no error";
      }
      case "frozen": {
        const o = Object.freeze({});
        o.x = 1;
        return "no error";
      }
      case "delete-builtin": {
        delete Object.prototype;
        return "no error";
      }
      case "this-plain-call": {
        const f = function () { return this; };
        return f() === undefined ? "undefined" : "defined";
      }
      default: {
        Math.sqrt(-1);
        return "no error";
      }
    }
  } catch (e) {
    return e.constructor.name;
  }
}

for (let i = 1; i <= n; i++) console.log(`${tokens[i]}: ${attempt(tokens[i])}`);
```

QUIZ

1. Predict the output:

```
function f() {
  return
    { ok: true };
}
console.log(f());
```

A. `{ ok: true }`

B. `undefined` — ASI ends the `return` at the line break

C. A syntax error

D. `true`

2. Which line is dangerous without a leading semicolon in semicolon-free code?

A. `const x = 1`

B. `[1, 2].forEach(f)` — it continues the previous line as an index

C. `if (x) y()`

D. `let y`

3. An arrow function with an expression body…

A. Needs `return`

B. Returns the expression's value implicitly: `x ⇒ x * 2`

C. Cannot use the conditional operator

D. Must have braces

4. In strict mode, `this` inside a plain function call is…

A. The global object

B. `undefined`

C. The function itself

D. `null`

5. `const obj = { class: 1 }; obj.class` …

A. Is a syntax error — `class` is reserved

B. Is fine: reserved words may be property names, just not variable names

C. Returns `undefined`

D. Creates a class

ANSWER KEY · STATEMENTS, EXPRESSIONS AND THE SEMICOLON RULES

1. B Keep the returned value on the same line as `return`.

2. B Lines starting with `(`, `[`, a template literal, `+`, `-` or `/` continue the previous statement.

3. B With braces it is a block body and needs an explicit `return`.

4. B Sloppy mode substitutes the global object — one of the silent behaviours strict mode removes.

5. B `const class = 1` would be the error.

LESSON 5 · 14 MIN

The event loop — a first look

JavaScript runs your code on one thread, and yet a Node server handles thousands of connections and a browser stays responsive while a download runs. The trick is the **event loop**: the engine runs one piece of code to completion, then picks up the next piece from a queue — a timer that fired, a file that finished reading, a click. Nothing runs *at the same time* as your code; things run *after* it. This lesson is the first look — enough to predict the order of `console.log`s in code with `setTimeout` and promises, which is one of the most common interview questions in the language. Module 8 goes deep.

The call stack

Every function call pushes a frame; returning pops it. When the stack is empty, the current *task* is finished. There is exactly one stack, and while it is non-empty nothing else — no timer callback, no click handler — can run. This is why `while (true) {}` freezes a page and why a long synchronous loop delays every timer: the loop must finish before the event loop gets a turn.

```
console.log("a");
setTimeout(() ⇒ console.log("c"), 0);
console.log("b");
// a, b, c — the timer callback cannot run until the current script finishes,
```

Tasks and the queue

Host APIs — timers, I/O, events — do their waiting *outside* the engine and, when done, put a **task** (a callback) on the task queue. The event loop's job: when the stack is empty, take the oldest task, run it to completion, repeat. `setTimeout(fn, 0)` therefore means "run `fn` in a later turn, not now", and a delay is a *minimum*: if the current task takes 200 ms, a 10 ms timer fires after 200 ms.

Microtasks: the queue that jumps the line

Promises (and `queueMicrotask`) schedule **microtasks**, which run **after the current task finishes and before the next task starts** — the whole microtask queue is drained, including microtasks queued by microtasks. That gives the famous ordering:

```
console.log("1");
setTimeout(() ⇒ console.log("timeout"), 0);
Promise.resolve().then(() ⇒ console.log("promise"));
queueMicrotask(() ⇒ console.log("microtask"));
console.log("2");
// 1, 2, promise, microtask, timeout
```

Synchronous code first (`1`, `2`), then every microtask in order (`promise`, `microtask`), then the task (`timeout`). Node adds `process.nextTick(fn)`, which runs even before promise microtasks, and `setImmediate(fn)`, a task that runs after I/O callbacks in the current loop iteration — details for module 8; the sync → microtasks → tasks order is what to internalise now.

Why this design

One thread means no data races: two callbacks never interleave in the middle of a statement, so JavaScript needs no locks and shared state is safe *within* a task. The cost is that any blocking work blocks everything — CPU-heavy loops belong in a Worker thread or in chunks yielded between turns, and I/O must be asynchronous, which is why every Node API that touches the outside world takes a callback or returns a promise. "Don't block the event loop" is the first rule of Node.

Predicting output: the method

1. Run the whole script top to bottom, printing synchronous logs, and *collecting* what was scheduled: microtasks in one list, tasks (timers) in another.
2. Drain the microtask list in order — adding any microtasks they schedule to the end of the same list.
3. Take the first task; run it as a new step 1 (its own logs, its own microtasks drained before the next task).
4. Repeat until nothing is scheduled.

```
setTimeout(() => { console.log("t1"); Promise.resolve().then(() => console.log
setTimeout(() => console.log("t2"), 0);
Promise.resolve().then(() => console.log("p1")).then(() => console.log("p2"));
console.log("sync");
// sync, p1, p2, t1, p-in-t1, t2
```

`p2` runs before `t1` because it is a microtask queued by a microtask; `p-in-t1` runs before `t2` because each task's microtasks drain before the next task starts.

Timers, precisely

`setTimeout(fn, ms, ...args)` schedules once; `setInterval(fn, ms)` repeats; both return an id for `clearTimeout` / `clearInterval`. Delays are clamped (Node treats anything under 1 ms as 1 ms), never exact, and never early. Two timers with the same delay fire in creation order. A `setTimeout` inside a `setInterval` handler is the standard way to build a *self-adjusting* interval; a plain `setInterval` drifts.

Blocking, measured

```
const start = Date.now();
setTimeout(() => console.log("fired after", Date.now() - start, "ms"), 10);
while (Date.now() - start < 100) {}   // busy-wait 100 ms
// fired after ~100 ms — not 10
```

The timer was *due* at 10 ms; the loop kept the stack busy; the callback ran at the first opportunity. Every janky UI and every "my server stopped responding" has this shape.

What comes next

Callbacks were how JavaScript expressed "later" for fifteen years and led to nesting hell; promises (ES2015) made "later" a value you can pass around; `async` / `await` (2017) made it read like synchronous code while still running on this same loop. Module 8 covers all three. For now, the mental model: **one stack, a task queue, a microtask queue that drains first.**

Interview angle

"Is JavaScript single-threaded?" Yes — one call stack; concurrency comes from the event loop scheduling callbacks between tasks (plus Worker threads for real parallelism).

"Order of `console.log` with `setTimeout(0)` and `Promise.then`?" Synchronous logs, then promise callbacks (microtasks), then the timeout (task).

"What is a microtask?" A callback (promise reaction, `queueMicrotask`) run after the current task and before any other task, with the queue drained completely.

"Does `setTimeout(fn, 0)` run immediately?" No — after the current task and all microtasks; the delay is a minimum.

"Why must you not block the event loop?" Everything — timers, I/O callbacks, UI events — waits behind the running task.

Key takeaways

One thread, one call stack; a task runs to completion before anything else runs.

Host APIs queue tasks; the loop runs the oldest when the stack is empty; timer delays are minimums.

Microtasks (promises, `queueMicrotask`) drain fully between tasks — hence sync → microtasks → tasks.

Predict output by collecting, then draining, then taking one task at a time.

Blocking delays everything; heavy work goes to Workers or asynchronous APIs.

EXERCISE 1

Schedule it for real

Read `n` tokens of the form `<kind>:<label>` where kind is `sync`, `micro` or `timeout`. Process them in order: `sync` logs its label immediately; `micro` schedules `console.log(label)` with `Promise.resolve().then(...)`; `timeout` schedules it with `setTimeout(..., 0)`. Let the runtime run and observe the order: every `sync`, then every `micro` in order, then every `timeout` in order — regardless of how the tokens were interleaved.

Example: `4` then `timeout:t1 sync:s1 micro:m1 sync:s2` →

```
s1
s2
m1
t1
```

INPUT 1

```
4
timeout:t1 sync:s1 micro:m1 sync:s2
```

OUTPUT 1

```
s1
s2
m1
t1
```

INPUT 2

```
3
micro:a micro:b timeout:c
```

OUTPUT 2

```
a
b
c
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
for (let i = 1; i <= n; i++) {
  const [kind, label] = tokens[i].split(":");
  if (kind === "sync") console.log(label);
  else if (kind === "micro") Promise.resolve().then(() => console.log(label));
  else setTimeout(() => console.log(label), 0);
}
```

EXERCISE 2

Simulate the loop

The same input as before — `n` tokens `sync:<label>`, `micro:<label>`, `timeout:<label>` — but this time compute the order **without scheduling anything**: walk the tokens once, printing `sync` labels immediately and collecting `micro` labels into one list and `timeout` labels into another; then print the microtask list, then the timeout list. Finish with `order=<all labels in output order, joined by ,>`.

Example: `4` then `timeout:t1 sync:s1 micro:m1 sync:s2` →

```
s1
s2
m1
t1
order=s1,s2,m1,t1
```

INPUT 1

```
4
timeout:t1 sync:s1 micro:m1 sync:s2
```

OUTPUT 1

```
s1
s2
m1
t1
order=s1,s2,m1,t1
```

INPUT 2

```
2
timeout:only sync:first
```

OUTPUT 2

```
first
only
order=first,only
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
const micro = [], tasks = [], order = [];
const emit = (label) ⇒ { console.log(label); order.push(label); };
for (let i = 1; i ⩽ n; i++) {
  const [kind, label] = tokens[i].split(":");
  if (kind ⩵ "sync") emit(label);
  else if (kind ⩵ "micro") micro.push(label);
  else tasks.push(label);
}
micro.forEach(emit);
tasks.forEach(emit);
console.log(`order=${order.join(",")}`);
```

QUIZ

1. Predict the output order:

```
console.log(1);
setTimeout(() ⇒ console.log(2), 0);
Promise.resolve().then(() ⇒ console.log(3));
console.log(4);
```

A. 1 2 3 4

B. 1 4 3 2

C. 1 4 2 3

D. 1 3 4 2

2. `setTimeout(fn, 0)` guarantees that `fn` runs…

A. Immediately

B. After the current task and all pending microtasks — the delay is a minimum

C. Exactly 0 ms later

D. Before any promise callback

3. Microtasks queued by a microtask run…

A. In the next task

B. Before the next task — the microtask queue drains completely

C. Never

D. Only in browsers

4. Why does `while (true) {}` freeze a page?

A. It uses too much memory

B. The call stack never empties, so no event — click, timer, response — can be handled

C. It throws after a while

D. It blocks the network only

5. Two `setTimeout` callbacks with the same delay run…

A. In random order

B. In the order they were scheduled

C. Simultaneously

D. In reverse order

ANSWER KEY · THE EVENT LOOP — A FIRST LOOK

1. B Synchronous logs first, then the microtask (promise), then the task (timeout).

2. B A busy stack delays every timer.

3. B Which is why a promise chain of ten `then`s finishes before one `setTimeout(0)`.

4. B One thread: whatever is on the stack must finish first.

5. B The task queue is FIFO for equal due times.

CHECKPOINT · 24 MIN

Checkpoint — JavaScript and its runtime

This checkpoint covers what JavaScript is and how ECMAScript is specified, engines and the JIT pipeline, Node as a runtime with its globals, reading stdin and writing exact output, statements versus expressions and automatic semicolon insertion, strict mode, and the event loop's ordering of synchronous code, microtasks and tasks.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- The difference between JavaScript, ECMAScript and V8.
- The engine pipeline from source to machine code, and why predictable object shapes are fast.
- What `readFileSync(0, "utf8")` does and why `"".split(/\s+/)` needs a check.
- Why `console.log` of an array is wrong for a judge, and how to print many lines fast.
- The three places ASI bites, and what strict mode changes.
- The order of synchronous logs, promise callbacks and `setTimeout(0)` callbacks.

The programs are a token-cursor input parser that survives ragged lines, a report formatter that builds exact output, and an event-loop simulator that prints the order a script's logs would appear in.

EXERCISE 1

Ragged input census

Read all of standard input. Print `lines=<number of non-blank lines>`, `tokens=<number of whitespace-separated tokens>`, `numbers=<how many tokens are finite numbers by Number()>` and `sum=<sum of those numbers>` (print integers without a decimal point, otherwise up to 2 decimals via `Number(x.toFixed(2))`). Empty input prints all zeros.

Example input

```
3 apples   4.5
  and 2 more
```

→

```
lines=2
tokens=6
numbers=3
sum=9.5
```

INPUT 1

```
3 apples   4.5
  and 2 more
```

OUTPUT 1

```
lines=2
tokens=6
numbers=3
sum=9.5
```

INPUT 2

```

```

OUTPUT 2

```
lines=0
tokens=0
numbers=0
sum=0
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const lines = input.split("\n").filter((l) => l.trim() !== "");
const tokens = input.trim() === "" ? [] : input.trim().split(/\s+/);
const numeric = tokens.filter((t) => t !== "" && Number.isFinite(Number(t)));
const sum = numeric.reduce((acc, t) => acc + Number(t), 0);
console.log(`lines=${lines.length}`);
console.log(`tokens=${tokens.length}`);
console.log(`numbers=${numeric.length}`);
console.log(`sum=${Number.isInteger(sum) ? sum : Number(sum.toFixed(2))}`);
```

EXERCISE 2

Report card

Read `n` and `n` lines `<name> <s1> <s2> <s3>`. For each print the name left-aligned in 8, the three scores each right-aligned in 4, and the average with one decimal right-aligned in 6 — e.g. `ada 90 80 70 80.0`. Then a line `best=<name with the highest average; first on ties>`. Build all lines first and print once.

Example: `2` then `ada 90 80 70`, `bob 100 100 95` →

```
ada       90  80  70  80.0
bob      100 100  95  98.3
best=bob
```

INPUT 1

```
2
ada 90 80 70
bob 100 100 95
```

OUTPUT 1

```
ada       90  80  70  80.0
bob      100 100  95  98.3
best=bob
```

INPUT 2

```
1
z 0 0 0
```

OUTPUT 2

```
z          0   0   0   0.0
best=z
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
let pos = 0;
const next = () => tokens[pos++];
const n = Number(next());
const out = [];
let best = null, bestAvg = -Infinity;
for (let i = 0; i < n; i++) {
  const name = next();
  const scores = [Number(next()), Number(next()), Number(next())];
  const avg = (scores[0] + scores[1] + scores[2]) / 3;
  out.push(name.padEnd(8) + scores.map((s) => String(s).padStart(4)).join("") + avg.
  if (avg > bestAvg) { bestAvg = avg; best = name; }
}
out.push(`best=${best}`);
console.log(out.join("\n"));
```

EXERCISE 3

Event loop simulator, nested

Simulate the ordering rules with nested scheduling. Read `n` tokens; each is `<kind>:<label>` or `<kind>:<label>><kind>:<label>` — the part after `>` is what the callback schedules when it runs (one level deep). Kinds: `sync`, `micro`, `timeout`. Rules: run the script (sync labels print immediately; scheduled items join their queues); drain the microtask queue fully (a microtask that schedules a microtask adds it to the **end** of the same drain; one that schedules a timeout adds to the task queue); then run tasks one at a time, draining microtasks after **each**. Print the labels in order, then `order=<joined by ,>`.

Example: `3` then `timeout:t1>micro:m2 micro:m1>timeout:t2 sync:s` →

```
s
m1
t1
m2
t2
order=s,m1,t1,m2,t2
```

INPUT 1

```
3
timeout:t1>micro:m2 micro:m1>timeout:t2
sync:s
```

OUTPUT 1

```
s
m1
t1
m2
t2
order=s,m1,t1,m2,t2
```

INPUT 2

```
2
micro:a>micro:b micro:c
```

OUTPUT 2

```
a
c
b
order=a,c,b
```

REFERENCE SOLUTION

```
"use strict";
const tokens = require("fs").readFileSync(0, "utf8").trim().split(/\s+/);
const n = Number(tokens[0]);
const order = [];
const emit = (label) => { console.log(label); order.push(label); };
const micro = [], tasks = [];
const parse = (t) => t.split(":");

function schedule(kind, label, then) {
  if (kind === "sync") run({ label, then });
  else if (kind === "micro") micro.push({ label, then });
  else tasks.push({ label, then });
}
function run(item) {
  emit(item.label);
  if (item.then) schedule(item.then[0], item.then[1], null);
}
function drainMicro() {
  while (micro.length) run(micro.shift());
}

for (let i = 1; i <= n; i++) {
  const [head, tail] = tokens[i].split(">");
  const [kind, label] = parse(head);
  schedule(kind, label, tail ? parse(tail) : null);
}
drainMicro();
while (tasks.length) {
  run(tasks.shift());
  drainMicro();
}
console.log(`order=${order.join(",")}`);
```

CHECKPOINT QUESTIONS

1. TC39 is…

A. A JavaScript engine

B. The committee that specifies ECMAScript in yearly editions

C. Node's package registry

D. A testing framework

2. The engine pipeline, in order, is…

A. JIT → parse → interpret

B. Parse → bytecode interpreted → profile → JIT-compile hot code → deoptimise on failed speculation

C. Compile ahead of time → run

D. Interpret only

3. Which is **not** a Node host global?

A. `process`

B. `Buffer`

C. `document`

D. `require`

4. `readFileSync(0, "utf8")` returns…

A. A line

B. All of stdin as a string

C. A Buffer

D. An array of tokens

5. To print exactly `1 2 3` from an array…

A. `console.log(arr)`

B. `console.log(arr.join(" "))`

C. `console.log(String(arr))`

D. `console.log(...arr, "")`

6. `x.toFixed(2)` returns…

A. A number

B. A string

C. An integer

D. A BigInt

7. Predict: `const f = () ⇒ { return 42; }; f()`

A. `42`

B. `undefined`

C. A syntax error

D. `null`

8. Strict mode makes `undeclaredName = 5` …

A. Create a global

B. Throw a `ReferenceError`

C. Throw a `TypeError`

D. Be ignored

9. Predict the order: `setTimeout(a, 0); Promise.resolve().then(b); c();`

A. a b c

B. c b a

C. c a b

D. b c a

10. A microtask that queues another microtask…

A. Runs it in the next task

B. Runs it in the same drain, before any timer

C. Runs it immediately, recursively

D. Is an error

11. Node stays alive after the script ends when…

A. Always

B. Something is pending — a timer, a server socket, an unfinished read

C. `"use strict"` is set

D. Never

12. `const` in JavaScript means…

A. The value is immutable

B. The binding cannot be reassigned; an object it points to can still change

C. Compile-time constant only

D. Same as `var`

ANSWER KEY · CHECKPOINT — JAVASCRIPT AND ITS RUNTIME

1. B Features pass through four stages before landing in a yearly edition.

2. B Fast start-up from the interpreter, peak speed from the optimising compiler.

3. C `document` belongs to the browser's DOM.

4. B `trim().split(/\s+/)` turns it into tokens.

5. B `console.log(arr)` gives `[ 1, 2, 3 ]`; `String(arr)` gives `1,2,3`.

6. B Arithmetic on it would concatenate; convert back with `Number` if needed.

7. B ASI terminates `return` at the newline.

8. B In sloppy mode it silently creates a global variable.

9. B Synchronous `c`, then the microtask `b`, then the timer `a`.

10. B The microtask queue is drained to empty before the loop proceeds.

11. B `process.exit` ends it regardless.

12. B `const o = {}; o.x = 1` is fine; `o = {}` is not.