Module 9 of 16

Modules and Node

CommonJS and ES modules, resolution and interop; package.json, npm, semver and lockfiles; the core modules — path, fs, process, events, url, crypto; Buffers and streams with pipeline and backpressure; arguments, environment, exit codes, signals, child processes and workers.

A JavaScript program bigger than one file lives inside a module system, a package, and a process — and each has rules that are invisible until they break: `Cannot use import statement outside a module`, a dependency that installs a different version in CI, a script that exits 0 after failing, a stream that silently stops on error. This module is the Node platform as a working developer meets it.

It opens with the two module systems — how `require` and `import` load, cache and bind, how Node decides which one a file is, and how they interoperate on Node 16 — then packages: the `package.json` fields that matter, the resolution algorithm precisely enough to debug it, semver ranges and why the lockfile is committed. The core-module lesson is a tour of `path`, `fs`, `process`, `util`, `events`, `url` and `crypto` with the idiom and the trap for each. Streams get a full lesson — Buffers and encodings, the four stream kinds, `pipeline` over `pipe`, backpressure, object-mode Transforms. The last lesson is the process itself: parsing arguments, configuration through the environment, exit codes and signals, safe child processes, and when to reach for workers.

The exercises run on real Node: import a live ES module from a `data:` URL, implement the `require` resolution algorithm over a virtual file tree, write a semver matcher, audit a `package.json`, round-trip files through a temp directory, decode Buffers across a split UTF-8 boundary, build object-mode stream pipelines, parse argv and `.env` files — and finish with a module-graph loader that detects cycles, a log-summarising pipeline, and a configuration CLI with flag > env > default precedence.

CONTENTS

1	CommonJS and ES modules — two module systems, one runtime	14 min
2	Packages, package.json, npm and semver	12 min
3	The core modules you use every week — fs, path, os, process, util, events, url, crypto	14 min
4	Buffers and streams — bytes, and data that does not fit in memory	14 min
5	The process — arguments, environment, exit codes, signals, child processes and workers	13 min
✓	Checkpoint — Modules and Node	24 min

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

LESSON 1 · 14 MIN

CommonJS and ES modules — two module systems, one runtime

JavaScript had no module system for its first fifteen years; Node invented one (**CommonJS**, `require` / `module.exports`) and the language later standardised another (**ES modules**, `import` / `export`). Today both run in Node, they differ in ways that matter — synchronous versus asynchronous loading, copied values versus live bindings, `__dirname` versus `import.meta.url` — and most real projects mix them through the interop rules. This lesson covers what a module *is* (a file with its own scope), each system's mechanics and caching, the differences, the interop rules as they stand on Node 16, circular imports, and dynamic `import()`.

A module is a file with private scope

Every file is a module: top-level `const` / `function` declarations are private to it, and only what it exports is visible outside. There is no shared global namespace to pollute (a browser `<script>` without `type="module"` is the exception — its top level *is* global). Modules are evaluated **once** and cached: two `require`s or `import`s of the same file return the same instance, so module-level state is a process-wide singleton — a cache, a connection pool, a config object.

CommonJS

```
// math.js
const PI = 3.14159;
function area(r) { return PI * r * r; }
module.exports = { PI, area };            // the exports object is what require()
// or: exports.area = area;   (exports is an alias for module.exports — do not rea

// app.js
const math = require("./math");          // relative path: ./ or ../ — resolves .j
const { area } = require("./math");      // destructure a copy
const fs = require("fs");                // a core module (also "node:fs")
const express = require("express");      // a package: walks up node_modules/ dire
```

`require` is a **synchronous function call**: the file is read, wrapped in `function (exports, require, module, __filename, __dirname) { … }`, executed, and `module.exports` is returned and cached in `require.cache` keyed by absolute path. Because it is a plain function you can call it conditionally, in a loop, with a computed path. Values are **copied** at the moment of `require`: a destructured `count` does not change when the module later changes its own `count`. JSON files can be required directly.

ES modules

```
// math.mjs  (or .js with "type": "module" in package.json)
export const PI = 3.14159;
export function area(r) { return PI * r * r; }
export default class Circle { … }

// app.mjs
import Circle, { area, PI as pi } from "./math.mjs";   // default + named; extensi
import * as math from "./math.mjs";                     // namespace object (froze
import fs from "node:fs";                               // core modules
export { area } from "./math.mjs";                      // re-export
```

Imports and exports are **static** — declared at the top level, resolved before any code runs — which lets bundlers tree-shake unused exports and lets the engine link modules before executing them. Loading is **asynchronous**: the whole graph is fetched and linked, then evaluated in dependency order. Bindings are **live**: `import { count }` reflects every later change the exporting module makes to `count` (you cannot assign to it from outside). ES modules are always strict mode, top-level `this` is `undefined`, `await` is allowed at the top level, and `__dirname` / `__filename` / `require` do not exist — use `import.meta.url` (a `file://` URL string) with `fileURLToPath`, and `createRequire(import.meta.url)` when you must `require`.

Which system a file is

Node decides by extension and the nearest `package.json`: `.mjs` is always ESM, `.cjs` always CommonJS, `.js` follows `"type"` in package.json (`"module"` → ESM, absent or `"commonjs"` → CJS). Getting this wrong produces the two most-searched errors: `Cannot use import statement outside a module` (ESM syntax in a CJS file) and `require is not defined in ES module scope` (the reverse).

Interop on Node 16

ESM importing CJS works: `import pkg from "./legacy.cjs"` gives `module.exports` as the default export; named imports of CJS work only for exports Node can detect statically (a heuristic — when it fails, import the default and destructure).

CJS requiring ESM does **not** work on Node 16 (`ERR_REQUIRE_ESM`); use `await import("./x.mjs")` from CJS — dynamic import works in both systems. (Node 22 later added `require(esm)` for graphs without top-level `await`.)

A package can offer both through `"exports"` conditions (`"import"` / `"require"`) — the dual-package pattern, which risks two copies of the module's state if both are loaded.

Circular dependencies

A requires B, B requires A. CommonJS returns B the **partial** `exports` of A as filled in so far — often an empty object — so `B` sees `undefined` for things A defines later; ESM's live bindings are in their temporal dead zone until evaluated, so reading one throws `ReferenceError: Cannot access 'x' before initialization`. Both are symptoms of the same design problem; break the cycle (move the shared piece to a third module) or defer the use to call time. CommonJS's "sometimes undefined" is the nastier failure because it is silent.

Dynamic `import()`

```
const { default: heavy } = await import("./heavy.js");   // returns a promise of t
if (needsChart) await import("chart.js");                // code-splitting: bundle
const ns = await import("data:text/javascript,export const x = 1");   // any URL N
```

Use it for conditional or lazy loading, for loading ESM from CJS, and for plugins named at runtime. It is the only form of import that can appear inside functions.

Choosing, and migrating

New code: ESM (`"type": "module"`), the standard, the one browsers and TypeScript target, the one bundlers optimise. Existing Node projects: often still CJS, and fine. Libraries: ship ESM, add a CJS build only if users need `require`. Migrating a CJS codebase means renaming or setting

`"type"`, adding file extensions to relative imports, replacing `__dirname` / `require.resolve` / JSON requires, and dealing with any dependency that is ESM-only or CJS-only.

Common mistakes

- ESM syntax in a `.js` file without `"type": "module"` (or the reverse).
- Omitting the file extension in ESM relative imports (`import "./math"` fails in Node; bundlers hid this).
- Expecting a destructured CJS value to update; expecting an ESM import to be assignable.
- `require` of an ESM package on Node 16; expecting named imports from every CJS package.
- Reassigning `exports = …` (breaks the alias; assign `module.exports`).
- Module-level state assumed per-import — it is per-process.

Interview angle

- *"CommonJS versus ESM?"* Synchronous `require` returning copied values versus static, asynchronously-linked `import` with live bindings; ESM is strict, supports top-level `await`, lacks `__dirname` / `require`.
- *"How does Node decide the module type of a file?"* Extension (`.mjs` / `.cjs`) or the nearest `package.json` `"type"`.
- *"Can CJS require ESM?"* Not on Node 16 — use `import()`; ESM can import CJS (default = `module.exports`).
- *"What is a live binding?"* An import that reflects the exporter's current value instead of a copy.
- *"What happens with circular imports?"* CJS hands out partially-filled exports; ESM throws on reading an uninitialised binding.

Key takeaways

- Every file is a private scope, evaluated once and cached — module state is process-wide.
- CJS: `require` / `module.exports`, synchronous, dynamic, copies; ESM: `import` / `export`, static, asynchronous, live, strict, `import.meta.url`.
- Type by `.mjs` / `.cjs` or `package.json` `"type"`; extensions are mandatory in ESM imports.
- ESM imports CJS (default export); CJS uses `import()` for ESM on Node 16; dual packages risk duplicate state.
- Break circular dependencies; use `import()` for lazy, conditional and runtime-named loading.

EXERCISE 1

Live bindings, observed

The starter defines an ES module as a `data:` URL exporting `let count`, `inc()` and a default. Import it with `await import(url)`, print `count before=<>`, call `inc()` the number of times given on input, print `count after=<>` (the binding is live), import the same URL again and print `sameInstance=<again === ns> default=<ns.default>`, then `namespaceKeys=<sorted keys>`, then try `ns.count = 99` and print `assignToImport=<changed or the error's constructor name> stillCount=<ns.count>`, and finally the CommonJS side of the same file: `requireCached=<require("fs") === require("fs")> typeofRequire=<typeof require> dirnameIsString=<typeof __dirname === "string">`.

Example: `3` →

```
count before=0
count after=3
sameInstance=true default=the default
namespaceKeys=count,default,inc
assignToImport=TypeError stillCount=3
requireCached=true typeofRequire=function dirnameIsString=true
```

INPUT 1

```
3
```

OUTPUT 1

```
count before=0
count after=3
sameInstance=true default=the default
namespaceKeys=count,default,inc
assignToImport=TypeError stillCount=3
requireCached=true typeofRequire=function
dirnameIsString=true
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const n = Number(lines[0]);
const url = "data:text/javascript,export let count = 0; export function inc() { count++;
async function main() {
  const ns = await import(url);
  console.log(`count before=${ns.count}`);
  for (let i = 0; i < n; i++) ns.inc();
  console.log(`count after=${ns.count}`);              // live binding: the namespace ref
  const again = await import(url);
  console.log(`sameInstance=${again === ns} default=${ns.default}`);
  console.log(`namespaceKeys=${Object.keys(ns).sort().join(",")}`);
  let assign;
  try { ns.count = 99; assign = "changed"; } catch (err) { assign = err.constructor.name;
  console.log(`assignToImport=${assign} stillCount=${ns.count}`);
  console.log(`requireCached=${require("fs") === require("fs")} typeofRequire=${typeof re
}
main();
```

EXERCISE 2

Implement require's resolution algorithm

Over a virtual file tree, resolve CommonJS specifiers the way Node does. Input: `file <absolute path>` declares a file, `pkg <dir> <main>` declares a `package.json` with that `main`, `require <fromDir> <specifier>` asks for a resolution. Rules: a core module (`fs`, `path`, `os`, `http`, `events`, `util`, `url`, `crypto`, `stream`, with or without `node:`) → `<core name>`; a specifier starting with `./`, `../` or `/` → normalise against `fromDir`, then try the exact path, `.js`, `.json`, then as a directory (its `main`, then `index.js` / `index.json`); a bare name → for `fromDir` and each parent up to `/`, try `<dir>/node_modules/<name>` with the same file-then-directory logic. Print `<fromDir> require(<spec>) → <path>` or `→ MODULE_NOT_FOUND`. Use `path.posix`.

Example (excerpt): with `/app/src/util.js`, `/app/src/lib/index.js`, `/app/node_modules/lodash/lodash.js` + `pkg /app/node_modules/lodash lodash.js`:

```
/app/src require(./util) → /app/src/util.js
/app/src require(./lib) → /app/src/lib/index.js
/app/src/lib require(lodash) → /app/node_modules/lodash/lodash.js
/app/src require(node:fs) → <core fs>
/app/src require(express) → MODULE_NOT_FOUND
```

INPUT 1

```
file /app/src/util.js
file /app/src/lib/index.js
file /app/node_modules/lodash/lodash.js
pkg /app/node_modules/lodash lodash.js
file /app/node_modules/tiny/index.js
file /app/src/data.json
require /app/src ./util
require /app/src ./lib
require /app/src/lib ../data
require /app/src/lib lodash
require /app/src tiny
require /app/src node:fs
require /app/src ./nope
require /app/src express
```

OUTPUT 1

```
/app/src require(./util) →
/app/src/util.js
/app/src require(./lib) →
/app/src/lib/index.js
/app/src/lib require(../data) →
/app/src/data.json
/app/src/lib require(lodash) →
/app/node_modules/lodash/lodash.js
/app/src require(tiny) →
/app/node_modules/tiny/index.js
/app/src require(node:fs) → <core fs>
/app/src require(./nope) →
MODULE_NOT_FOUND
/app/src require(express) →
MODULE_NOT_FOUND
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const path = require("node:path").posix;
const CORE = new Set(["fs", "path", "os", "http", "events", "util", "url", "crypto", "str
const files = new Set();        // absolute file paths
const mains = new Map();        // package dir → main
function tryFile(p) {
  for (const candidate of [p, `${p}.js`, `${p}.json`]) if (files.has(candidate)) return c
  return null;
}
function tryDirectory(dir) {
  if (mains.has(dir)) {
    const viaMain = tryFile(path.join(dir, mains.get(dir)));
    if (viaMain) return viaMain;
  }
  return tryFile(path.join(dir, "index"));      // index.js / index.json
}
function loadAs(base) { return tryFile(base) ?? tryDirectory(base); }
function resolve(fromDir, spec) {
  if (CORE.has(spec.replace(/^node:/, ""))) return `<core ${spec.replace(/^node:/, "")}>`
  if (spec.startsWith("./") || spec.startsWith("../") || spec.startsWith("/")) return loa
  for (let dir = fromDir; ; dir = path.dirname(dir)) {
    const found = loadAs(path.join(dir, "node_modules", spec));
    if (found) return found;
    if (dir === "/") return null;
  }
}
for (const line of lines) {
  const [cmd, a, b] = line.trim().split(/\s+/);
  if (cmd === "file") files.add(path.normalize(a));
  else if (cmd === "pkg") mains.set(path.normalize(a), b);
  else if (cmd === "require") console.log(`${a} require(${b}) → ${resolve(a, b) ?? "MODU
}
```

QUIZ

1. A `.js` file is treated as an ES module when…

A. It contains `import`

B. The nearest `package.json` has `"type": "module"` (or the file is `.mjs`)

C. It is in `src/`

D. Node is version 16 or later

2. `const { count } = require("./counter")` — when the module later increments its own `count`, the destructured variable…

   A. Updates

   B. Keeps the old value — CommonJS hands out copies at require time; ESM imports would be live

   C. Becomes `undefined`

   D. Throws

3. On Node 16, a CommonJS file that needs an ESM-only package must…

   A. `require` it

   B. Use `await import("pkg")` — `require` of ESM throws `ERR_REQUIRE_ESM`

   C. Rename itself to `.mjs` only

   D. Bundle it

4. In a CommonJS cycle A → B → A, what does B receive from `require("./a")`?

   A. A `ReferenceError`

   B. A's **partial** exports as filled in so far — often an empty object — silently

   C. The complete module

   D. `null`

5. `import { area } from "./math"` fails in Node ESM because…

   A. Named imports are not supported

   B. Relative ESM specifiers require the file extension: `"./math.js"`

   C. `math` is a core module

   D. It must be a default import

ANSWER KEY · COMMONJS AND ES MODULES — TWO MODULE SYSTEMS, ONE RUNTIME

1. B Get it wrong and you see `Cannot use import statement outside a module` or `require is not defined`.

2. B Read `counter.count` through the exports object, or use ESM.

3. B ESM importing CJS works the other way round (default export = `module.exports`).

4. B ESM throws on reading an uninitialised live binding instead; both mean the cycle should be broken.

5. B Bundlers hid this for years; Node does not guess.

LESSON 2 · 12 MIN

Packages, package.json, npm and semver

A **package** is a directory with a `package.json`; npm is the registry and the tool that installs packages into `node_modules`, and `require` / `import` of a bare name is the resolution algorithm that finds them there. Almost every JavaScript project is a package, depends on dozens more, and is shaped by a handful of `package.json` fields and one versioning convention — semver — that decides what `npm install` gives you today versus six months from now. This lesson covers the fields that matter, the resolution algorithm precisely enough to debug it, semver ranges and lockfiles, scripts, and the security realities of installing other people's code.

`package.json`, the fields that matter

```
{
  "name": "@acme/tools",                  // scoped name; lower-case, URL-safe
  "version": "2.3.1",                     // semver
  "description": "…",
  "type": "module",                       // .js files are ESM (omit or "commonjs"
  "main": "./dist/index.cjs",             // legacy entry point for require()
  "exports": {                            // modern entry points — the public surfa
    ".": { "import": "./dist/index.js", "require": "./dist/index.cjs" },
    "./utils": "./dist/utils.js"
  },
  "bin": { "acme": "./bin/cli.js" },      // commands installed on PATH (npx acme,
  "scripts": { "build": "tsc", "test": "node --test", "start": "node dist/index.js
  "dependencies": { "express": "^4.18.2" },
  "devDependencies": { "typescript": "~5.4.0", "eslint": "8.57.0" },
  "peerDependencies": { "react": "⩾18" },
  "engines": { "node": "⩾18" },
  "files": ["dist"],                      // what gets published (plus package.json
  "private": true                         // refuse to publish
}
```

`dependencies` are needed at runtime by whoever installs you; `devDependencies` only to develop the package itself (not installed by consumers); `peerDependencies` say "the host must provide this" (plugins, React components). `exports` restricts what consumers can import (deep imports of internal files break when you refactor otherwise) and supports **conditions** — `import` / `require` / `node` / `default` / `types`. `engines` documents the Node range and, with `engine-strict`, enforces it.

Resolution: how `require("x")` finds a file

Core module? (`fs`, `node:fs`) → built in.

Relative or absolute (`./`, `../`, `/`) → try the exact path, then `.js`, `.json`, `.node`; then as a directory: its `package.json` `main`, then `index.js`.

Bare specifier (`lodash`, `@acme/tools/utils`) → look in `./node_modules/`, then `../node_modules/`, then `../../node_modules/` … up to the filesystem root. Inside the package directory, honour `exports` if present (only the listed subpaths are allowed), else `main`, else `index.js`.

That upward walk is why nested `node_modules` work, why a monorepo can hoist shared packages to the root, and why the same package at two versions can exist twice in one tree (each dependant finds its nearest copy). `require.resolve("x")` prints the resolved path; `node --trace-warnings` and `NODE_DEBUG=module` show the search.

Semver

`MAJOR.MINOR.PATCH` (+ optional prerelease `-beta.1`): patch = bug fixes, minor = backwards-compatible features, major = breaking changes. **Zero-major** versions (`0.x.y`) promise nothing — any change may break. Range operators:

Range	Means	Matches
`1.2.3`	exactly	`1.2.3`
`^1.2.3`	same major, ≥ 1.2.3 (default for `npm install`)	`1.2.3` … `1.99.99`, not `2.0.0`
`^0.2.3`	for 0.x: same **minor**	`0.2.3` … `0.2.99`
`~1.2.3`	same minor, ≥ 1.2.3	`1.2.3` … `1.2.99`
`⩾1.2.0 <2.0.0`	explicit	
`1.x` / `1.*`	any 1.y.z	
`*` / `latest`	anything	never in a real project

Ranges are a promise about what *you* accept; whether the author kept the semver contract is another matter — hence lockfiles.

Lockfiles and reproducible installs

`package-lock.json` (npm), `yarn.lock`, `pnpm-lock.yaml` record the **exact** version and integrity hash of every installed package, transitive ones included. `npm install` respects the lockfile where it can and updates it when `package.json` changed; `npm ci` installs **exactly** the lockfile or fails, deleting `node_modules` first — use it in CI and deployments. Commit the lockfile; a project without one is a different program on every install. `npm outdated` and `npm update` are how you move ranges forward deliberately.

Scripts

`npm run <name>` runs `scripts.<name>` with `node_modules/.bin` on `PATH` (so `"test": "jest"` finds the local jest) and the package's env exposed as `npm_package_*`. `npm test`, `npm start`, `npm run build` are conventions; `pre<name>` / `post<name>` scripts run around a script automatically (`pretest`, `postinstall`). **Lifecycle scripts** — `prepare` (runs on local `npm install` and before publish), `postinstall` (runs after *anyone* installs your package) — are powerful and a security concern: a compromised package's `postinstall` runs arbitrary code on every developer's machine and CI. `npx <pkg>` runs a package's binary, downloading it if needed.

Publishing, in one paragraph

`npm version patch|minor|major` bumps and tags; `npm publish` uploads what `files` / `.npmignore` allow (check with `npm pack --dry-run`); scoped packages publish private by default (`--access public`). Never publish secrets; never publish `node_modules`; use `"private": true` on applications.

Supply chain

Installing a package runs its install scripts and gives its code full access to your process, files and network. Defences: a committed lockfile and `npm ci`; `npm audit` (with judgement — many findings are irrelevant devDependency noise); `--ignore-scripts` where possible; fewer dependencies (a `left-pad` is a liability, not a saving); pinning in applications, ranges in libraries; provenance/attestations and 2FA for maintainers. Typosquatting (`lodahs`) and dependency

confusion (a public package shadowing a private name) are real attacks — scoped names and registry configuration mitigate them.

Monorepos and workspaces

`"workspaces": ["packages/*"]` lets one repository hold several packages that depend on each other by name; npm symlinks them into the root `node_modules` so resolution finds the local source. Tools (Turborepo, Nx, pnpm) add task orchestration and caching on top.

Common mistakes

- Runtime dependencies in `devDependencies` (works locally, breaks in production installs with `--omit=dev`).
- No lockfile, or `npm install` in CI instead of `npm ci`.
- `*` / `latest` ranges; relying on a `0.x` package as if `^` protected you.
- Deep imports into a package's internals (`lodash/internal/…`) — `exports` exists to stop this.
- Publishing without `files` / `private`; `postinstall` scripts you do not need.
- Two copies of a package in the tree causing `instanceof` failures or duplicated state (check `npm ls <pkg>`).

Interview angle

- *"`dependencies` versus `devDependencies` versus `peerDependencies`?"* Runtime for consumers / development only / must be provided by the host.
- *"How does `require('x')` find `x`?"* Core module, else relative path with extension and index probing, else walk up `node_modules` directories, honouring `exports` / `main`.
- *"`^1.2.3` versus `~1.2.3`?"* Same major (minor and patch may rise) versus same minor (patch may rise); `^0.x` behaves like `~`.
- *"Why commit the lockfile? `npm ci`?"* Reproducible installs — exact versions and hashes; `ci` installs exactly that or fails.
- *"What is the `exports` field for?"* Defining the package's public entry points with conditions, blocking deep imports.

Key takeaways

- `package.json`: `name` / `version`, `type`, `main` / `exports` / `bin`, `scripts`, the three dependency kinds, `engines`, `files` / `private`.
- Resolution: core → relative (extensions, `index.js`) → walk up `node_modules`; `exports` gates the public surface.
- Semver `^` (major), `~` (minor), `0.x` is unstable; lockfile + `npm ci` for reproducibility.
- Scripts get `.bin` on PATH; `pre` / `post` and lifecycle scripts run automatically — a supply-chain risk.
- Fewer dependencies, audited, locked, scoped; `npm ls` to find duplicates.

EXERCISE 1

A semver range matcher

Implement `satisfies(version, range)` for the ranges npm users type: an exact version; `^x.y.z` (same major; for `^0.y.z` same minor; for `^0.0.z` exact); `~x.y.z` (same minor); comparators `⩾`, `>`, `⩽`, `<`; wildcards `1.x`, `1.2.x`, `*`; and space-separated parts that must all hold. Commands: `check <version>`

`check <range> <version>` prints `<range> vs <version>: <bool>`; `max <range> <versions…>` prints `max <range>: <highest satisfying version or none>`.

Example (excerpt):

```
^1.2.3 vs 1.9.0: true
^1.2.3 vs 2.0.0: false
~1.2.3 vs 1.3.0: false
^0.2.3 vs 0.3.0: false
⩾1.2.0 <2.0.0 vs 1.999.0: true
max ^1.0.0: 1.10.0
```

`1.10.0` beats `1.4.2` — compare numerically, never as strings.

INPUT 1

```
check 1.9.0 ^1.2.3
check 2.0.0 ^1.2.3
check 1.2.9 ~1.2.3
check 1.3.0 ~1.2.3
check 0.2.9 ^0.2.3
check 0.3.0 ^0.2.3
check 1.5.2 1.x
check 1.999.0 ⩾1.2.0 <2.0.0
check 2.0.0 ⩾1.2.0 <2.0.0
check 9.9.9 *
check 1.2.3 1.2.3
check 1.2.4 1.2.3
max ^1.0.0 0.9.0 1.0.0 1.4.2 2.0.0 1.10.0
max ~3.1.0 3.0.9 3.2.0
```

OUTPUT 1

```
^1.2.3 vs 1.9.0: true
^1.2.3 vs 2.0.0: false
~1.2.3 vs 1.2.9: true
~1.2.3 vs 1.3.0: false
^0.2.3 vs 0.2.9: true
^0.2.3 vs 0.3.0: false
1.x vs 1.5.2: true
⩾1.2.0 <2.0.0 vs 1.999.0: true
⩾1.2.0 <2.0.0 vs 2.0.0: false
* vs 9.9.9: true
1.2.3 vs 1.2.3: true
1.2.3 vs 1.2.4: false
max ^1.0.0: 1.10.0
max ~3.1.0: none
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim() ⇒
const parse = (v) ⇒ v.split(".").map(Number);   // [major, minor, patch]
const cmp = (a, b) ⇒ { for (let i = 0; i < 3; i++) if (a[i] ⩵⩵ b[i]) return a[i] - b[i]
function satisfiesOne(v, part) {
  if (part ⩵⩵ "*" || part ⩵⩵ "x") return true;
  const m = /^(\^|~|⩾|<=▷|<)?(\d+)(?:\.(\d+|x|\*))?(?:\.(\d+|x|\*))?$/.exec(part);
  if (!m) throw new SyntaxError(`bad range part '${part}'`);
  const [, op = "", maj, min, pat] = m;
  const wild = min ⩵⩵ undefined || min ⩵⩵ "x" || min ⩵⩵ "*" || pat ⩵⩵ undefined || pa
  const lower = [Number(maj), min ⩵⩵ undefined || /x|\*/.test(min) ? 0 : Number(min), pa
  let upper;                                          // exclusive
  if (op ⩵⩵ "^") upper = lower[0] > 0 ? [lower[0] + 1, 0, 0] : lower[1] > 0 ? [0, lower[
  else if (op ⩵⩵ "~") upper = [lower[0], lower[1] + 1, 0];
  else if (op ⩵⩵ "") upper = wild ? (min ⩵⩵ undefined || /x|\*/.test(min) ? [lower[0] +
  if (op ⩵⩵ "⩾") return cmp(v, lower) ⩾ 0;
  if (op ⩵⩵ ">") return cmp(v, lower) > 0;
  if (op ⩵⩵ "⩽") return cmp(v, lower) ⩽ 0;
  if (op ⩵⩵ "<") return cmp(v, lower) < 0;
  if (upper ⩵⩵ null) return cmp(v, lower) ⩵⩵ 0;     // exact
  return cmp(v, lower) ⩾ 0 && cmp(v, upper) < 0;
}
function satisfies(version, range) {
  const v = parse(version);
  return range.split(/\s+/).every((part) ⇒ satisfiesOne(v, part));
}
for (const line of lines) {
  const [cmd, ...rest] = line.trim().split(/\s+/);
  if (cmd ⩵⩵ "check") { const [version, ...range] = rest; console.log(`${range.join(" ")
  else if (cmd ⩵⩵ "max") {
    const [range, ...versions] = rest;
    const ok = versions.filter((v) ⇒ satisfies(v, range)).sort((a, b) ⇒ cmp(parse(a), p
    console.log(`max ${range}: ${ok.length ? ok[ok.length - 1] : "none"}`);
  }
}
```

EXERCISE 2

Audit a package.json

Read a `package.json` from stdin and print a report: `name=<name or (missing)> version=<version if valid semver else invalid>`; `type=<type or commonjs (default)> entry=<exports | main <main> | index.js (implicit)>`; `scripts=<sorted names or none>`; `deps=<n> devDeps=<n> peerDeps=<n>`; `unpinned: <sorted names whose range is *, x or latest, or none>`; and `warnings: <joined by "; " or none>` for: no `engines.node`; a `postinstall` script; publishable (`private` not true) without a `files` list; a package listed in both `dependencies` and `devDependencies`.

Example (a library with several problems) →

```
name=@acme/tools version=2.3.1
type=module entry=exports
scripts=build,postinstall,test
deps=2 devDeps=2 peerDeps=1
unpinned: express,left-pad
warnings: no engines.node; postinstall script runs on every install; publishable without
```

INPUT 1

```
{"name":"@acme/tools","version":"2.3.1","type":"module","exports":
{".":"./dist/index.js"},"scripts":{"test":"node --
test","build":"tsc","postinstall":"node setup.js"},"dependencies":
{"express":"^4.18.2","left-pad":"*"},"devDependencies":
{"typescript":"~5.4.0","express":"latest"},"peerDependencies":
{"react":"⩾18"}}
```

OUTPUT 1

```
name=@acme/tools version=2.3.1
type=module entry=exports
scripts=build,postinstall,test
deps=2 devDeps=2 peerDeps=1
unpinned: express,left-pad
warnings: no engines.node;
postinstall script runs on
every install; publishable
without a files allow-list;
listed in both dependencies
and devDependencies: express
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const pkg = JSON.parse(input);
const semver = /^\d+\.\d+\.\d+(-[0-9A-Za-z.-]+)?$/;
console.log(`name=${pkg.name ?? "(missing)"} version=${semver.test(pkg.version ?? "") ? p
console.log(`type=${pkg.type ?? "commonjs (default)"} entry=${pkg.exports ? "exports" : p
console.log(`scripts=${Object.keys(pkg.scripts ?? {}).sort().join(",") || "none"}`);
const deps = pkg.dependencies ?? {}, dev = pkg.devDependencies ?? {}, peer = pkg.peerDepe
console.log(`deps=${Object.keys(deps).length} devDeps=${Object.keys(dev).length} peerDeps
const unpinned = Object.entries({ ...deps, ...dev }).filter(([, r]) => r === "*" || r ===
console.log(`unpinned: ${unpinned.join(",") || "none"}`);
const warnings = [];
if (!pkg.engines?.node) warnings.push("no engines.node");
if (pkg.scripts?.postinstall) warnings.push("postinstall script runs on every install");
if (!pkg.private && !pkg.files) warnings.push("publishable without a files allow-list");
const both = Object.keys(deps).filter((d) => d in dev).sort();
if (both.length) warnings.push(`listed in both dependencies and devDependencies: ${both.j
console.log(`warnings: ${warnings.join("; ") || "none"}`);
```

QUIZ

1. `^1.2.3` accepts…

A. Only `1.2.x`

B. Any `1.y.z` ≥ 1.2.3 — same major; `~1.2.3` would fix the minor too

C. Any version ≥ 1.2.3

D. Exactly 1.2.3

2. `npm ci` differs from `npm install` in that it…

A. Is faster only

B. Installs exactly the lockfile (failing if `package.json` disagrees) after removing `node_modules` — reproducible CI installs

C. Updates dependencies

D. Skips devDependencies

3. A package used only at runtime by consumers was put in `devDependencies`. The effect is…

A. None

B. Consumers (and `--omit=dev` production installs) do not get it — the module fails to load in production

C. It installs twice

D. npm warns

4. The `exports` field in `package.json` …

A. Lists files to publish

B. Defines the package's public entry points (with `import` / `require` conditions) and blocks deep imports of anything else

C. Replaces `scripts`

D. Is for TypeScript only

5. Why is a `postinstall` script a supply-chain concern?

A. It slows installs

B. It runs arbitrary code on every machine that installs the package — a compromised dependency executes there

C. It is deprecated

D. It cannot use `node`

ANSWER KEY · PACKAGES, PACKAGE.JSON, NPM AND SEMVER

1. B `^0.2.3` behaves like `~`: 0.x versions promise nothing across minors.

2. B Commit the lockfile; use `ci` in CI and deployments.

3. B `dependencies` for runtime, `devDependencies` for developing the package itself.

4. B Refactoring internals no longer breaks consumers who reached into them.

5. B Lockfiles, `npm audit`, fewer dependencies and `--ignore-scripts` where possible.

LESSON 3 · 14 MIN

The core modules you use every week — fs, path, os, process, util, events, url, crypto

Node's standard library is small by design and almost entirely I/O and OS: files, paths, the process, events, URLs, hashing, and utilities. Every one of these you will reach for weekly, and each has one or two rules that are not obvious — `path.join` versus `path.resolve`, three flavours of `fs`, `process.exit` cutting off pending output, `EventEmitter` throwing on an unhandled `"error"`. This lesson is a working tour with the idioms and the traps, organised by module; streams and child processes get their own lessons.

`path`

```
const path = require("node:path");
path.join("/app", "src", "../lib", "x.js");   // "/app/lib/x.js"  — joins and norm
path.resolve("src", "x.js");                  // "/current/working/dir/src/x.js" —
path.basename("/a/b/file.txt");               // "file.txt";  basename(p, ".txt")
path.extname("archive.tar.gz");               // ".gz"
path.dirname("/a/b/file.txt");                // "/a/b"
path.relative("/a/b", "/a/c/d");              // "../c/d"
path.parse("/a/b/file.txt");                  // { root: "/", dir: "/a/b", base: "
path.sep; path.delimiter;                     // "/" and ":" (POSIX), "\\" and ";"
path.posix.join(…); path.win32.join(…);       // force a flavour — for URLs use po
```

Never concatenate paths with `+` and `/`; `join` handles separators, doubled slashes and `..`. `resolve` is `cd` semantics — the first absolute segment from the right wins. Build paths relative to the source file with `path.join(__dirname, …)` (CJS) or `new URL("./x", import.meta.url)` (ESM), never relative to the current working directory, which is wherever the user ran the command.

`fs`, three ways

```
const fs = require("node:fs");
const fsp = require("node:fs/promises");
fs.readFileSync("a.txt", "utf8");                                  // sync: fine f
fs.readFile("a.txt", "utf8", (err, text) => { … });                // callback: le
const text = await fsp.readFile("a.txt", "utf8");                  // promises: th
await fsp.writeFile("out.json", JSON.stringify(data, null, 2));    // creates or t
await fsp.appendFile("log.txt", line + "\n");
await fsp.mkdir("dir/sub", { recursive: true });                   // no error if
await fsp.rm("dir", { recursive: true, force: true });             // rm -rf
const entries = await fsp.readdir("dir", { withFileTypes: true }); // Dirent: entr
const st = await fsp.stat("a.txt");  st.size; st.mtime; st.isFile();
await fsp.access("a.txt").then(() => true, () => false);           // exists? — bu
const tmp = await fsp.mkdtemp(path.join(os.tmpdir(), "app-"));     // a unique tem
```

Without an encoding argument you get a `Buffer`, not a string. Errors carry a `code`: `ENOENT` (no such file), `EACCES` (permission), `EEXIST`, `EISDIR`, `ENOTDIR` — branch on `err.code`. Checking existence then opening is a race; open and handle `ENOENT`. For large files use streams (next lesson); `readFile` loads the whole thing. `fs.watch` is platform-inconsistent — use chokidar for real watching.

`os` and `process`

```
os.tmpdir(); os.homedir(); os.platform(); os.cpus().length; os.EOL; os.totalmem();
process.argv;            // ["/path/node", "/path/script.js", ...args]
process.env.PORT;        // strings only; undefined when unset — validate at start
process.cwd(); process.chdir(dir); process.pid; process.version; process.platform;
process.exitCode = 1;    // set and let the loop drain — flushes stdout; process.e
process.on("exit", (code) => …);             // synchronous only — no async work
process.stdout.write("no newline");          // console.log is stdout + newline;
process.hrtime.bigint(); process.memoryUsage().heapUsed;
```

`process.exit()` kills pending writes and skips `finally` blocks in flight — set `exitCode` instead unless you truly must stop now. Environment variables are the configuration channel for deployments (twelve-factor); read them once into a validated config object.

`util`

`util.promisify(fn)` for error-first callbacks; `util.inspect(obj, { depth: null, colors: true })` for full nested printing (what `console.log` uses, at depth 2); `util.format("%s has %d items", name, n)`; `util.types.isPromise(x)`, `util.isDeepStrictEqual(a, b)`; `util.parseArgs` (Node 18.3+ — hand-roll on 16); `util.deprecate(fn, msg)`; `util.TextEncoder` / `TextDecoder` (also global). `util.inherits` is the pre-class inheritance helper you will meet in old code.

`events` — `EventEmitter`

```
const { EventEmitter, once } = require("node:events");
class Job extends EventEmitter { run() { this.emit("progress", 50); this.emit("don
const job = new Job();
job.on("progress", (p) => …);                // any number of listeners, called s
job.once("done", handler);                   // auto-removed after the first call
job.off("progress", fn);                     // remove (same function reference)
job.emit("done", payload);                   // returns true if anyone listened
await once(job, "done");                     // promise of the first emission's a
```

`emit` is **synchronous**: listeners run before `emit` returns, in registration order — so a slow listener slows the emitter, and a throw in a listener propagates to the `emit` call. An emitted `"error"` with **no listener throws** — always attach one on emitters that can fail (streams, sockets, servers). More than 10 listeners on one event prints a leak warning (`setMaxListeners` if intentional). The browser's `EventTarget` is the same idea with `addEventListener`; Node has `EventTarget` too.

`url`, `querystring`

```
const u = new URL("https://user:pw@example.com:8443/a/b?x=1&y=2#frag");   // WHATW
u.hostname; u.port; u.pathname; u.search; u.hash; u.origin;
u.searchParams.get("x"); u.searchParams.append("z", "3"); [...u.searchParams];  //
new URL("../c", "https://example.com/a/b/");                              // r
const { fileURLToPath, pathToFileURL } = require("node:url");              //
```

Never build query strings by hand; `URLSearchParams` encodes correctly. The legacy `url.parse` and `querystring` modules are superseded.

`crypto`

`crypto.randomUUID()` (ids), `crypto.randomBytes(16).toString("hex")` (tokens — never `Math.random`), `crypto.createHash("sha256").update(text).digest("hex")` (checksums — **not** for passwords), `crypto.scrypt` / `pbkdf2` or the `bcrypt` / `argon2` packages for passwords, `crypto.createHmac("sha256", secret)` for signatures, `crypto.timingSafeEqual` when

comparing secrets, `crypto.subtle` (WebCrypto) for the standard API. The rule: use the library's high-level primitive for the job; never invent a scheme.

`Buffer` (preview) and others

`Buffer.from("héllo", "utf8").length` is bytes, not characters; `buf.toString("base64")`; next lesson. Also worth knowing by name: `readline` (line-by-line stdin/files), `zlib` (gzip), `http` / `https` (servers and requests — frameworks sit on these), `net` / `dgram`, `dns`, `timers/promises`, `worker_threads`, `assert`, `test` (Node 18+).

Common mistakes

Paths relative to `process.cwd()` instead of the module; string concatenation instead of `path.join`.

`readFile` without encoding, then treating the Buffer as a string; checking `existsSync` before opening (race).

`process.exit()` right after `console.log` in a script with buffered output — the log is lost.

No `"error"` listener on an emitter; expecting `emit` to be asynchronous.

Hand-built query strings; `Math.random` for anything security-relevant; `sha256` for passwords.

Reading `process.env` all over the code instead of once into a validated config.

Interview angle

"`path.join` versus `path.resolve`?" `join` concatenates and normalises (may stay relative); `resolve` produces an absolute path, processing right-to-left until an absolute segment.

"Sync versus async fs?" Sync blocks the loop — acceptable at startup and in CLIs; async (promises) in anything serving concurrent work.

"What happens on `emit('error')` with no listener?" It throws.

"`process.exit(1)` versus `process.exitCode = 1`?" Immediate termination (pending writes lost) versus a graceful exit when the loop drains.

"How do you generate a secure token?" `crypto.randomBytes` / `randomUUID`, never `Math.random`.

Key takeaways

`path.join` / `resolve` / `basename` / `extname` / `relative`; build from `__dirname` / `import.meta.url`, `posix` for URLs.

`fs/promises` by default, sync at startup; encoding for strings; branch on `err.code`; temp dirs via `mkdtemp`.

`process.argv` / `env` / `exitCode`; `exit()` is abrupt; env is strings, validate once.

`EventEmitter` is synchronous; attach `"error"`; `once` returns a promise.

`URL` / `URLSearchParams` for URLs; `crypto` primitives for ids, tokens, hashes, HMACs — never roll your own.

EXERCISE 1

A path toolkit

Using `require("node:path").posix` (so the answers are identical on every OS) and a fixed working directory `/home/user/project` for `resolve`, execute each command and print `<command> <args> ⇒ <result>`: `join a b …`, `resolve a b …` (resolved from the fixed directory), `normalize p`, `basename p`

`[ext]`, `extname p` (JSON-quoted so an empty result is visible), `dirname p`, `relative from to`, `parse p` (JSON).

Example (excerpt):

```
join /app src ../lib x.js ⇒ /app/lib/x.js
resolve src x.js ⇒ /home/user/project/src/x.js
resolve /etc x ⇒ /etc/x
normalize /a//b/./c/../d ⇒ /a/b/d
basename /a/b/file.txt .txt ⇒ file
extname archive.tar.gz ⇒ ".gz"
extname .bashrc ⇒ ""
relative /a/b /a/c/d ⇒ ../c/d
```

INPUT 1

```
join /app src
../lib x.js
resolve src
x.js
resolve /etc x
normalize
/a//b/./c/../d
basename
/a/b/file.txt
basename
/a/b/file.txt
.txt
extname
archive.tar.gz
extname .bashrc
dirname
/a/b/file.txt
relative /a/b
/a/c/d
parse
/a/b/file.txt
```

OUTPUT 1

```
join /app src ../lib x.js ⇒ /app/lib/x.js
resolve src x.js ⇒ /home/user/project/src/x.js
resolve /etc x ⇒ /etc/x
normalize /a//b/./c/../d ⇒ /a/b/d
basename /a/b/file.txt ⇒ file.txt
basename /a/b/file.txt .txt ⇒ file
extname archive.tar.gz ⇒ ".gz"
extname .bashrc ⇒ ""
dirname /a/b/file.txt ⇒ /a/b
relative /a/b /a/c/d ⇒ ../c/d
parse /a/b/file.txt ⇒
{"root":"/","dir":"/a/b","base":"file.txt","ext":".txt","name":"file"}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim() ⇒
const path = require("node:path").posix;   // posix on purpose: the same answers on every
const CWD = "/home/user/project";
for (const line of lines) {
  const [cmd, ...args] = line.trim().split(/\s+/);
  let out;
  if (cmd ⇒⇒ "join") out = path.join(...args);
  else if (cmd ⇒⇒ "resolve") out = path.resolve(CWD, ...args);
  else if (cmd ⇒⇒ "normalize") out = path.normalize(args[0]);
  else if (cmd ⇒⇒ "basename") out = path.basename(args[0], args[1]);
  else if (cmd ⇒⇒ "extname") out = JSON.stringify(path.extname(args[0]));
  else if (cmd ⇒⇒ "dirname") out = path.dirname(args[0]);
  else if (cmd ⇒⇒ "relative") out = path.relative(args[0], args[1]);
  else if (cmd ⇒⇒ "parse") out = JSON.stringify(path.parse(args[0]));
  console.log(`${cmd} ${args.join(" ")} ⇒ ${out}`);
}
```

EXERCISE 2

Files in a temp directory, round trip

Create a unique directory with `fsp.mkdtemp(path.join(os.tmpdir(), "study-"))` and run commands against it with `fs/promises`: `write name text…` (writes text plus a newline; prints `wrote name`), `append name text…` (`appended name`), `read name` (prints `name: <JSON of contents>`), `list` (`list: <sorted entries, directories with a trailing />` or `(empty)`), `stat name` (`name: <size> bytes file=<isFile>`), `rename a b`, `remove name`, `mkdir name`, `exists name` (`name exists=<bool>`, via `access`). A failing command prints `<cmd> <name>: error <err.code>`. Remove the directory in a `finally`.

Example (excerpt):

```
wrote a.txt
appended a.txt
a.txt: "hello world\nsecond line\n"
a.txt: 24 bytes file=true
list: a.txt sub/
read a.txt: error ENOENT
remove ghost.txt: error ENOENT
```

INPUT 1

```
write a.txt hello world
append a.txt second line
read a.txt
stat a.txt
mkdir sub
list
rename a.txt b.txt
list
read a.txt
exists b.txt
remove b.txt
exists b.txt
remove ghost.txt
```

OUTPUT 1

```
wrote a.txt
appended a.txt
a.txt: "hello world\nsecond line\n"
a.txt: 24 bytes file=true
mkdir sub
list: a.txt sub/
renamed a.txt → b.txt
list: b.txt sub/
read a.txt: error ENOENT
b.txt exists=true
removed b.txt
b.txt exists=false
remove ghost.txt: error ENOENT
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const fsp = require("node:fs/promises");
const os = require("node:os");
const path = require("node:path");
async function main() {
  const dir = await fsp.mkdtemp(path.join(os.tmpdir(), "study-"));
  const at = (name) => path.join(dir, name);
  try {
    for (const line of lines) {
      const [cmd, name, ...rest] = line.trim().split(/\s+/);
      const text = rest.join(" ");
      try {
        if (cmd === "write") { await fsp.writeFile(at(name), text + "\n"); console.log(`w
        else if (cmd === "append") { await fsp.appendFile(at(name), text + "\n"); console
        else if (cmd === "read") console.log(`${name}: ${JSON.stringify(await fsp.readFil
        else if (cmd === "list") {
          const entries = await fsp.readdir(dir, { withFileTypes: true });
          console.log(`list: ${entries.map((e) => `${e.name}${e.isDirectory() ? "/" : ""}
        } else if (cmd === "stat") { const st = await fsp.stat(at(name)); console.log(`${
        else if (cmd === "rename") { await fsp.rename(at(name), at(rest[0])); console.log
        else if (cmd === "remove") { await fsp.rm(at(name)); console.log(`removed ${name}
        else if (cmd === "mkdir") { await fsp.mkdir(at(name), { recursive: true }); consc
        else if (cmd === "exists") console.log(`${name} exists=${await fsp.access(at(name
      } catch (err) {
        console.log(`${cmd} ${name}: error ${err.code}`);
      }
    }
  } finally {
    await fsp.rm(dir, { recursive: true, force: true });
  }
}
main();
```

QUIZ

1. `path.join("/a", "../b")` and `path.resolve("/a", "../b")` …

   A. Both `/b`

   B. `join` → `/b` (normalised concatenation); `resolve` → `/b` too here, but `resolve("x")` would be absolute from the cwd while `join("x")` stays relative

   C. Both throw

   D. `join` keeps `..`

2. `fs.readFile("a.txt", cb)` without an encoding gives…

   A. A string

   B. A `Buffer` — pass `"utf8"` for a string

   C. An array of lines

   D. A stream

3. `emitter.emit("error", err)` with no `"error"` listener…

   A. Is ignored

   B. Throws `err` — always attach an error listener on emitters that can fail

   C. Logs a warning

   D. Rejects a promise

4. `console.log("done"); process.exit(0);` may…

   A. Always print `done`

   B. Lose the output — `exit()` is immediate and does not wait for pending writes; set `process.exitCode` instead

   C. Print twice

   D. Hang

5. To build a query string safely use…

   A. String concatenation with `&`

   B. `URLSearchParams` (or `URL#searchParams`) — it encodes values correctly

   C. `JSON.stringify`

   D. `querystring.escape` on the whole string

ANSWER KEY · THE CORE MODULES YOU USE EVERY WEEK — FS, PATH, OS, PROCESS, UTIL, EVENTS, URL, CRYPTO

1. B `resolve` always returns an absolute path; `join` only normalises.

2. B Same for `readFileSync` and `fsp.readFile`.

3. B Streams, sockets and servers are all emitters.

4. B `exit()` also skips in-flight `finally` blocks.

5. B The legacy `url.parse` / `querystring` modules are superseded by the WHATWG `URL`.

LESSON 4 · 14 MIN

Buffers and streams — bytes, and data that does not fit in memory

Two things separate a Node program that handles a 10 GB file from one that crashes on it: knowing that a `Buffer` is bytes and a string is characters, and processing data as a **stream** — chunk by chunk, with backpressure — instead of loading it whole. Streams are also how HTTP bodies, sockets, stdin, compression and file I/O are exposed, so even code that never touches a large file reads them constantly. This lesson covers `Buffer` and encodings, the four stream types, `pipeline` and why `pipe` alone is not enough, backpressure, object mode, async iteration over streams, `readline`, and writing a `Transform`.

`Buffer`

```
const b = Buffer.from("héllo", "utf8");     // <Buffer 68 c3 a9 6c 6c 6f>
b.length;                                    // 6 bytes — the string has 5 charact
Buffer.byteLength("héllo");                  // 6, without allocating
b.toString("utf8"); b.toString("hex"); b.toString("base64");
Buffer.from("aGk=", "base64").toString();    // "hi"
Buffer.alloc(4);                             // zero-filled; allocUnsafe is faster
Buffer.concat([b1, b2]); b.slice(0, 2) /* shares memory */; b.subarray(0, 2);
b.readUInt32BE(0); b.writeInt16LE(v, off);   // binary protocols
Buffer.compare(a, b); a.equals(b);
```

A `Buffer` is a fixed-size byte array (a `Uint8Array` subclass) that knows encodings: `utf8` (default), `utf16le`, `latin1`, `ascii`, `hex`, `base64`, `base64url`. The one rule that bites: a **UTF-8 chunk boundary can split a multi-byte character** — concatenating `chunk.toString()` per chunk produces garbage at the seams. Set an encoding on the stream (`setEncoding("utf8")`) or use a `StringDecoder`, which buffers partial characters, or collect the buffers and decode once.

The four kinds of stream

Type	Direction	Examples
`Readable`	you read from it	`fs.createReadStream`, `process.stdin`, an HTTP request body, `Readable.from(iterable)`
`Writable`	you write into it	`fs.createWriteStream`, `process.stdout`, an HTTP response
`Duplex`	both, independent	a TCP socket
`Transform`	writable in, readable out — a function over chunks	`zlib.createGzip()`, `crypto.createCipheriv()`, your own parsers

Streams are `EventEmitter`s: `"data"`, `"end"`, `"error"`, `"finish"`, `"close"`. Modern code rarely subscribes to those directly; it uses `pipeline` and `for await`.

`pipeline`, not `pipe`

```
const { pipeline } = require("node:stream/promises");
const zlib = require("node:zlib");
await pipeline(
  fs.createReadStream("access.log"),
  zlib.createGzip(),
  fs.createWriteStream("access.log.gz"),
);                                          // resolves on completion, rejects on
```

`a.pipe(b)` connects two streams but does **not** forward errors: an error in `a` leaves `b` open and unfinished, a classic leak. `pipeline` wires every stage, propagates errors, cleans up on failure, and returns a promise. Use it for anything with more than one stage or where a failure matters — always, in practice.

Backpressure

A fast producer and a slow consumer would fill memory without a brake. Writable streams signal it: `write()` returns `false` when the internal buffer is above `highWaterMark` (16 KB by default for bytes, 16 objects in object mode), and emits `"drain"` when it is safe to continue. `pipe` / `pipeline` handle this for you — pausing the readable until the writable drains. Writing in a loop by hand (`for (…) ws.write(chunk)`) ignores the signal and buffers everything; if you must write manually, await `drain` when `write` returns `false`, or use `for await` on a source and `await` the write's promise wrapper.

Reading a stream

```
// async iteration — the modern way; handles backpressure and errors
let bytes = 0;
for await (const chunk of fs.createReadStream(path)) bytes += chunk.length;

// line by line
const readline = require("node:readline");
const rl = readline.createInterface({ input: fs.createReadStream(path), crlfDelay:
for await (const line of rl) { … }          // also how a CLI reads stdin line by

// collect a whole body (when it is known to be small)
const chunks = [];
for await (const c of req) chunks.push(c);
const body = Buffer.concat(chunks).toString();

Readable.from(["a", "b", "c"]);              // any iterable or async iterable → a
```

`for await` pulls one chunk at a time, so the producer cannot outrun you. Do not mix it with `"data"` listeners on the same stream (both consume).

Writing a `Transform`

```
const { Transform } = require("node:stream");
const upper = new Transform({
  transform(chunk, encoding, callback) {           // called per input chunk
    this.push(chunk.toString().toUpperCase());     // push zero or more output chu
    callback();                                    // or callback(err); or callbac
  },
  flush(callback) { this.push("\n-- end --\n"); callback(); },   // once, before e
});
```

A byte-oriented transform must remember that chunks are arbitrary slices: a line splitter keeps the trailing partial line in a field and emits it in `flush`. **Object mode** (`objectMode: true`, or

`readableObjectMode` / `writableObjectMode` separately) makes chunks arbitrary JavaScript values — parsed records, events — with `highWaterMark` counted in objects. A CSV parser is a byte→object transform; a JSON-lines writer is object→byte.

Where streams show up

HTTP: `req` is a Readable, `res` a Writable — `pipeline(fs.createReadStream(file), res)` serves a file in constant memory. `process.stdin` / `stdout` / `stderr`. `child_process` stdio. WebSockets and sockets. Compression, hashing (`createHash` is a Transform), encryption. The Web Streams API (`ReadableStream`, `fetch` bodies) is the browser's equivalent; Node has it too and can convert (`Readable.toWeb` / `fromWeb`, Node 17+).

When not to stream

A 2 KB config file: `readFile`. A response you need entirely before acting on (JSON): collect it. Streams are for data that is large, unbounded, or produced over time — logs, uploads, exports, live feeds. Streaming a small file adds code and event-ordering hazards for nothing.

Common mistakes

- `chunk.toString()` per chunk on multi-byte text (split characters); treating `buffer.length` as a character count.
- `pipe` chains with no error handling; forgetting an `"error"` listener on a hand-driven stream.
- Writing in a loop without honouring `write()`'s return value.
- Mixing `for await` with `"data"` handlers; consuming a stream twice.
- A Transform that assumes one input chunk equals one line/record.
- Reading a 4 GB file with `readFile`.

Interview angle

- *"Buffer versus string?"* Bytes with an encoding versus UTF-16 characters; lengths differ; decode across chunk boundaries carefully.
- *"Why streams?"* Constant memory for large or unbounded data; processing starts before the data ends; composition via `pipeline`.
- *"`pipe` versus `pipeline`?"* `pipeline` propagates errors across every stage, destroys on failure, and returns a promise; `pipe` does none of that.
- *"What is backpressure?"* The consumer's signal to slow a producer — `write()` returning `false`, `"drain"` to resume; `pipeline` handles it.
- *"Write a Transform that upper-cases."* `new Transform({ transform(chunk, enc, cb) { cb(null, chunk.toString().toUpperCase()); } })`.

Key takeaways

- `Buffer` = bytes + encoding; `byteLength ≠ length`; never decode chunk-by-chunk without a `StringDecoder` or `setEncoding`.
- Readable / Writable / Duplex / Transform; `Readable.from` for iterables; object mode for records.
- `stream/promises` `pipeline` for every multi-stage flow; `pipe` alone leaks on error.
- `for await` and `readline` to consume; backpressure is automatic with `pipeline` / `for await`, manual with raw `write`.

Transforms see arbitrary chunk boundaries — buffer partial records, emit the rest in `flush`.

EXERCISE 1

Buffers, encodings and the split character

Line 1 is base64 text, line 2 hex. Decode each: print `base64 → <JSON string> bytes=<length> hex=<hex>` and `hex → <JSON string> base64=<base64> uint16be=<readUInt16BE(0) or n/a if under 2 bytes>`. Then build the word `caf` + `String.fromCodePoint(0xe9)` and print `length=<string length> byteLength=<UTF-8 bytes> bytes=<hex>`; split its buffer after byte 4 (inside the two-byte character), decode the chunks naively (`chunk.toString()` each, concatenated) and carefully (a `StringDecoder`), and print `naive=<code points in hex> careful=<code points in hex> equal=<careful === word>`; finally `compare=<Buffer.compare("a","b")> equals=<Buffer "hi" equals "hi"> concat=<concat of "ab" and "cd">`.

Example: `aGVsbG8=` / `4142` →

```
base64 → "hello" bytes=5 hex=68656c6c6f
hex → "AB" base64=QUI= uint16be=16706
length=4 byteLength=5 bytes=636166c3a9
naive=63 61 66 fffd fffd careful=63 61 66 e9 equal=true
compare=-1 equals=true concat=abcd
```

The naive decode produced two replacement characters (`fffd`) where the split `é` was.

INPUT 1

```
aGVsbG8=
4142
```

OUTPUT 1

```
base64 → "hello" bytes=5 hex=68656c6c6f
hex → "AB" base64=QUI= uint16be=16706
length=4 byteLength=5 bytes=636166c3a9
naive=63 61 66 fffd fffd careful=63 61 66
e9 equal=true
compare=-1 equals=true concat=abcd
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const { StringDecoder } = require("node:string_decoder");
const [b64, hex] = lines;
const fromB64 = Buffer.from(b64.trim(), "base64");
console.log(`base64 → ${JSON.stringify(fromB64.toString("utf8"))} bytes=${fromB64.length
const fromHex = Buffer.from(hex.trim(), "hex");
console.log(`hex → ${JSON.stringify(fromHex.toString("utf8"))} base64=${fromHex.toString
const accent = String.fromCodePoint(0xe9);              // one character, two UTF-8 bytes
const word = `caf${accent}`;
const bytes = Buffer.from(word, "utf8");
console.log(`length=${word.length} byteLength=${Buffer.byteLength(word)} bytes=${bytes.to
const chunks = [bytes.subarray(0, 4), bytes.subarray(4)];  // the boundary splits the two
const naive = chunks.map((c) => c.toString("utf8")).join("");
const decoder = new StringDecoder("utf8");
const careful = chunks.map((c) => decoder.write(c)).join("") + decoder.end();
const codes = (s) => [...s].map((ch) => ch.codePointAt(0).toString(16)).join(" ");
console.log(`naive=${codes(naive)} careful=${codes(careful)} equal=${careful === word}`);
console.log(`compare=${Buffer.compare(Buffer.from("a"), Buffer.from("b"))} equals=${Buffe
```

EXERCISE 2

Object-mode transforms and pipeline

Line 1 is a minimum level (`DEBUG < INFO < WARN < ERROR`); the rest are log lines `LEVEL message`. Build a `Transform` with `readableObjectMode` that parses each line into `{ level, message }` (and fails with `Error("corrupt line")` on the text `BAD`), a second object-mode `Transform` that drops records below the minimum, and an object-mode `Writable` that collects; run them with `stream/promises` `pipeline` over `Readable.from(lines without BAD)` and print `kept <n> of <m>: <LEVEL:message joined by " | "> and counts: <LEVEL=n in level order>`. If the input contained `BAD`, run a second pipeline **with** it and print `pipeline with BAD: failed - <message>; sinkDestroyed=<sink.destroyed>`. Finally count

non-empty lines of the whole input with `readline` over `Readable.from([input])` and print `readline lines=<n>`.

Example: `WARN`, then `INFO server started`, `DEBUG cache warm`, `WARN disk 91% full`, `ERROR db timeout`, `INFO request served`, `BAD` →

```
kept 2 of 5: WARN:disk 91% full | ERROR:db timeout
counts: WARN=1 ERROR=1
pipeline with BAD: failed - corrupt line; sinkDestroyed=true
readline lines=7
```

INPUT 1

```
WARN
INFO server started
DEBUG cache warm
WARN disk 91% full
ERROR db timeout
INFO request served
BAD
```

OUTPUT 1

```
kept 2 of 5: WARN:disk 91% full | ERROR:db
timeout
counts: WARN=1 ERROR=1
pipeline with BAD: failed - corrupt line;
sinkDestroyed=true
readline lines=7
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const { Readable, Transform, Writable } = require("node:stream");
const { pipeline } = require("node:stream/promises");
const readline = require("node:readline");
const LEVELS = ["DEBUG", "INFO", "WARN", "ERROR"];
async function main() {
  const [minLevel, ...logLines] = input.split("\n").filter((l) => l.trim() !== "");
  const parse = new Transform({
    readableObjectMode: true,                       // bytes/strings in, records out
    transform(chunk, _enc, cb) {
      const text = chunk.toString().trim();
      if (text === "BAD") return cb(new Error("corrupt line"));
      const [level, ...rest] = text.split(/\s+/);
      cb(null, { level, message: rest.join(" ") });
    },
  });
  const atLeast = (min) => new Transform({
    objectMode: true,
    transform(rec, _enc, cb) { if (LEVELS.indexOf(rec.level) >= LEVELS.indexOf(min)) this
  });
  const kept = [];
  const collect = new Writable({ objectMode: true, write(rec, _enc, cb) { kept.push(rec);
  const good = logLines.filter((l) => l.trim() !== "BAD");
  await pipeline(Readable.from(good), parse, atLeast(minLevel), collect);
  console.log(`kept ${kept.length} of ${good.length}: ${kept.map((r) => `${r.level}:${r.m
  const counts = kept.reduce((m, r) => m.set(r.level, (m.get(r.level) ?? 0) + 1), new Map
  console.log(`counts: ${LEVELS.filter((l) => counts.has(l)).map((l) => `${l}=${counts.ge
  if (good.length !== logLines.length) {
    const parse2 = new Transform({ readableObjectMode: true, transform(chunk, _enc, cb) {
    const sink = new Writable({ objectMode: true, write(_r, _e, cb) { cb(); } });
    await pipeline(Readable.from(logLines), parse2, sink).then(
      () => console.log("pipeline with BAD: completed"),
      (err) => console.log(`pipeline with BAD: failed - ${err.message}; sinkDestroyed=${s
    );
  }
  let count = 0;
  for await (const line of readline.createInterface({ input: Readable.from([input]) })) i
  console.log(`readline lines=${count}`);
}
main();
```

QUIZ

1. `Buffer.from("héllo").length` is…

   A. 5

   B. 6 — bytes, not characters; `é` is two bytes in UTF-8

   C. 10

   D. It throws

2. Decoding a UTF-8 stream with `chunk.toString()` per chunk risks…

   A. Nothing

   B. Garbled characters where a multi-byte sequence is split across two chunks — use `setEncoding` or a `StringDecoder`

   C. Memory leaks

   D. Slower code

3. `a.pipe(b)` versus `pipeline(a, b)`:

   A. Identical

   B. `pipeline` forwards errors from every stage, destroys the others on failure and returns a promise; `pipe` leaves `b` open when `a` errors

   C. `pipe` is asynchronous

   D. `pipeline` buffers everything

4. Backpressure means…

   A. Compressing data

   B. A writable signals it is full (`write()` returns `false`) and the producer waits for `"drain"` — `pipeline` and `for await` handle it for you

   C. An error

   D. A timeout

5. Object mode…

   A. Is for JSON only

   B. Lets chunks be arbitrary JavaScript values (records, events) with `highWaterMark` counted in objects

   C. Disables backpressure

   D. Requires TypeScript

ANSWER KEY · BUFFERS AND STREAMS — BYTES, AND DATA THAT DOES NOT FIT IN MEMORY

1. B `Buffer.byteLength(str)` gives the same without allocating.

2. B The boundary is arbitrary; the decoder must buffer partial sequences.

3. B Use `stream/promises` `pipeline` for any real flow.

4. B A hand-written `for` loop of `write()` calls ignores it and buffers unboundedly.

5. B A parser is a byte→object Transform; `Readable.from(array)` is object mode by default.

LESSON 5 · 13 MIN

The process — arguments, environment, exit codes, signals, child processes and workers

A Node program is an operating-system process: it is started with arguments and an environment, it reads stdin and writes stdout/stderr, it ends with an exit code, it receives signals, and it can start other processes or threads. Command-line tools and servers live or die on getting these right — a CLI that exits 0 on failure breaks every script that calls it; a server that ignores `SIGTERM` loses in-flight requests on every deploy. This lesson covers each surface, the hand-rolled argument parser you can write in twenty lines (Node 16 has no `util.parseArgs`), configuration through environment variables, graceful shutdown, `child_process` and its injection hazard, and when `worker_threads` are the right tool.

Arguments

`process.argv` is `[nodePath, scriptPath, ...userArgs]` — slice from 2. Conventions every CLI follows: `--flag` (boolean), `--key value` or `--key=value`, `-v` short flags (and clusters `-abc`), `--no-color` to negate, positional arguments, and `--` meaning "everything after is positional, even if it starts with a dash". A minimal parser:

```
function parseArgs(argv) {
  const opts = {}, positional = [];
  for (let i = 0; i < argv.length; i++) {
    const a = argv[i];
    if (a === "--") { positional.push(...argv.slice(i + 1)); break; }
    if (a.startsWith("--")) {
      const [key, inline] = a.slice(2).split("=", 2);
      if (inline !== undefined) opts[key] = inline;
      else if (key.startsWith("no-")) opts[key.slice(3)] = false;
      else if (i + 1 < argv.length && !argv[i + 1].startsWith("-")) opts[key] = ar
      else opts[key] = true;
    } else if (a.startsWith("-") && a.length > 1) {
      for (const ch of a.slice(1)) opts[ch] = true;
    } else positional.push(a);
  }
  return { opts, positional };
}
```

Real tools use `commander`, `yargs` or Node 18's `util.parseArgs` for help text, types and validation — but the shape above is what they all produce.

Environment variables

`process.env` is a plain object of **strings** (or `undefined`); `process.env.DEBUG = "1"` sets it for this process and its children only. Configuration belongs here in deployments (twelve-factor): ports, URLs, credentials, feature flags — never in code, never committed. The `.env` file convention (`KEY=value` lines, `#` comments, quotes for spaces, loaded by `dotenv` or Node 20's `--env-file`) is for local development; production injects real environment variables. Read them **once at startup** into a validated, typed config object with defaults and required checks, so a missing `DATABASE_URL` fails at boot with a clear message rather than at the first query — and precedence is explicit: command-line flag > environment variable > config file > default.

stdin, stdout, stderr, exit codes

`process.stdout` is for program **output** (the thing a pipe would consume), `process.stderr` for diagnostics and progress — `console.log` versus `console.error`. Read stdin with `for await`

`(const line of readline.createInterface({ input: process.stdin }))` or by collecting chunks; check `process.stdin.isTTY` to know whether a human or a pipe is on the other end. Exit code **0 means success**, anything else failure — shells, CI systems and `&&` chains depend on it. Set `process.exitCode = 1` and return normally so buffered output flushes; reserve `process.exit(code)` for "stop now" and know it skips pending I/O and `finally` blocks. Uncaught exceptions exit 1; unhandled rejections exit 1 on Node 15+.

Signals and graceful shutdown

```
let shuttingDown = false;
for (const sig of ["SIGINT", "SIGTERM"]) {
  process.on(sig, async () => {
    if (shuttingDown) return;
    shuttingDown = true;
    console.error(`${sig}: draining…`);
    server.close();                                  // stop accepting; finish in-
    await db.end();
    process.exit(0);                                 // now nothing pending is wor
  });
}
```

`SIGINT` is Ctrl-C; `SIGTERM` is what `kill`, Docker and Kubernetes send before `SIGKILL` (which cannot be handled — you have a grace period, typically 10–30 s). Installing a handler **replaces** Node's default exit, so you must exit yourself. Graceful shutdown: stop taking new work, finish or fail current work quickly, close connections, flush logs, exit. Also `process.on("uncaughtException")` / `("unhandledRejection")` — log and exit (module 7), and `"beforeExit"` fires when the loop drains (async work allowed) versus `"exit"` (synchronous only).

Child processes

```
const { execFile, spawn, exec } = require("node:child_process");
const { promisify } = require("node:util");
const { stdout } = await promisify(execFile)("git", ["rev-parse", "HEAD"]);    //
const child = spawn("ffmpeg", ["-i", input, output], { stdio: "inherit" });     //
child.on("exit", (code, signal) => …);
exec(`ls ${userInput}`, cb);      // runs through a SHELL — userInput of "; rm -rf
```

`execFile` / `spawn` take an argument array and start the program directly — no shell parsing, so no injection and no quoting problems. `exec` (and `spawn` with `shell: true`) run a shell command string: only for trusted, fixed commands that need shell features. `execFileSync` / `spawnSync` block the loop — fine in scripts. Buffered variants (`exec`, `execFile`) have a `maxBuffer` (1 MB default) and reject beyond it; use `spawn` and stream for big output. `fork()` starts another Node script with an IPC channel (`child.send` / `process.on("message")`). Always handle the `"error"` event (program not found) and check the exit code.

Threads: `worker_threads`

The event loop is one thread; a CPU-bound task (image processing, big JSON, crypto, a long computation) blocks every request for its duration. `worker_threads` run JavaScript on other threads with **separate** event loops and heaps, communicating by message passing (`postMessage` — structured clone, or transferable `ArrayBuffer`s, or `SharedArrayBuffer` for true sharing). A worker pool (`piscina`) reuses a fixed number of them. Use workers for CPU work; I/O is already non-blocking and gains nothing. `cluster` (or a process manager) runs several whole Node processes sharing a port for multi-core HTTP throughput.

A well-behaved CLI, summarised

Parse arguments; support `--help` and `--version`; read config in the flag > env > file > default order; write results to stdout and messages to stderr; exit non-zero on failure with a one-line reason; handle `SIGINT` if you hold resources; never print secrets; behave sanely when stdout is a pipe (no colours unless `isTTY`).

Common mistakes

- Exiting 0 on failure; `process.exit()` before output flushes.
- Reading `process.env` everywhere, treating values as numbers/booleans without parsing, committing `.env`.
- `exec` with interpolated user input.
- A `SIGTERM` handler that never calls `process.exit`, or none at all so deploys drop requests.
- Using workers for I/O, or `spawnSync` in a server.
- Progress messages on stdout, corrupting the output a pipe consumes.

Interview angle

- *"How does a CLI receive its arguments?"* `process.argv.slice(2)`; parse `--key=value`, `--flag`, `-x`, `--`, positionals.
- *"Where does configuration come from?"* Environment variables (strings) validated once at startup, with flags overriding and defaults below.
- *"`exec` versus `spawn` / `execFile`?"* `exec` runs a shell string (injection risk, buffered); `execFile` / `spawn` run a program with an argument array (safe, `spawn` streams).
- *"What is graceful shutdown?"* On `SIGTERM`, stop accepting work, finish in-flight work, close connections, then exit within the grace period.
- *"When do you use `worker_threads`?"* CPU-bound work that would block the event loop; not for I/O.

Key takeaways

- `argv.slice(2)` with the standard flag conventions; help/version; positionals after `--`.
- Env is strings, read once, validated, precedence flag > env > file > default; `.env` for local only.
- stdout for output, stderr for messages; exit 0 on success, `exitCode` over `exit()`.
- Handle `SIGINT` / `SIGTERM` for graceful shutdown and exit yourself; `SIGKILL` cannot be caught.
- `execFile` / `spawn` with argument arrays (no shell); workers for CPU work, `cluster` for multi-core serving.

EXERCISE 1

An argv parser

Write `parseArgs(argv)` returning `{ opts, positional }` with the standard conventions: `--key=value`; `--key value` when the next token does not start with `-`, otherwise `--key` is `true`; `--no-key` sets `key` to `false`; a cluster `-xy` sets `x` and `y` to `true`; `--` ends option parsing and everything after is positional; anything else is positional. The input line is the argv; print the result as JSON.

Example: `--name=ada --verbose -n 3 in.txt --tags a --no-color -xy out.txt -- --literal -z` →

```
{"opts":{"name":"ada","verbose":true,"n":true,"tags":"a","color":false,"x":true,"y":true}
```

Short flags are booleans here, so `-n 3` makes `3` positional — the rule set is the exercise's, and a real tool documents its own.

INPUT 1

```
--name=ada --verbose -n 3 in.txt --tags a --no-color -xy out.txt -- --literal -z
```

OUTPUT 1

```
{"opts":{"name":"ada","verbose":true,"n":true,"tags":"a","color":false,"x":true,"y":true},"positional":["3","in.txt","out.txt","--literal","-z"]}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
function parseArgs(argv) {
  const opts = {}, positional = [];
  for (let i = 0; i < argv.length; i++) {
    const a = argv[i];
    if (a === "--") { positional.push(...argv.slice(i + 1)); break; }
    if (a.startsWith("--")) {
      const eq = a.indexOf("=");
      if (eq !== -1) { opts[a.slice(2, eq)] = a.slice(eq + 1); continue; }
      const key = a.slice(2);
      if (key.startsWith("no-")) opts[key.slice(3)] = false;
      else if (i + 1 < argv.length && !argv[i + 1].startsWith("-")) opts[key] = argv[++i]
      else opts[key] = true;
    } else if (a.startsWith("-") && a.length > 1) {
      for (const ch of a.slice(1)) opts[ch] = true;      // -abc → a, b, c
    } else positional.push(a);
  }
  return { opts, positional };
}
console.log(JSON.stringify(parseArgs(lines[0].trim().split(/\s+/))));
```

EXERCISE 2

Parse a .env file and check what is required

Write `parseDotenv(text)` : skip blank lines and `#` comments; accept `KEY=VALUE` with an optional `export` prefix; strip matching single or double quotes (a `#` inside quotes is kept); on unquoted values drop a trailing `# comment` ; expand `${NAME}` from keys defined earlier (unknown → empty). The input is the file, then a line `---` , then the required key names. Print `env=<JSON>` , then `missing=<JSON array of required keys absent or empty> wouldExitWith=<1 if any, else 0>` , and register a `process.on("exit")` handler that prints `exit handler: code=<code>` .

Example:

```
# database
export DB_HOST=db.local
DB_PORT=5432
DB_URL=postgres://${DB_HOST}:${DB_PORT}/app
NAME="My App"
EMPTY=
MOTTO='keep # this'
PLAIN=value # a comment
---
DB_URL API_KEY EMPTY
```

→

```
env={"DB_HOST":"db.local","DB_PORT":"5432","DB_URL":"postgres://db.local:5432/app","NAME"
missing=["API_KEY","EMPTY"] wouldExitWith=1
exit handler: code=0
```

INPUT 1

```
# database
export DB_HOST=db.local
DB_PORT=5432
DB_URL=postgres://${DB_HOST}:${DB_PORT}/app
NAME="My App"
EMPTY=
MOTTO='keep # this'
PLAIN=value # a comment
---
DB_URL API_KEY EMPTY
```

OUTPUT 1

```
env=
{"DB_HOST":"db.local","DB_PORT":"5432","DB_URL":"postgres://db.local
App","EMPTY":"","MOTTO":"keep # this","PLAIN":"value"}
missing=["API_KEY","EMPTY"] wouldExitWith=1
exit handler: code=0
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
function parseDotenv(text) {
  const out = {};
  for (const raw of text.split("\n")) {
    const line = raw.trim();
    if (!line || line.startsWith("#")) continue;
    const m = /^(?:export\s+)?([A-Za-z_][A-Za-z0-9_]*)\s*=\s*(.*)$/.exec(line);
    if (!m) continue;
    let value = m[2].trim();
    if ((value.startsWith('"') && value.endsWith('"')) || (value.startsWith("'") && value
    else value = value.replace(/\s+#.*$/, "");                    // trailing comment on u
    value = value.replace(/\$\{([A-Za-z_][A-Za-z0-9_]*)\}/g, (_, name) => out[name] ?? "
    out[m[1]] = value;
  }
  return out;
}
const [envText, requiredLine] = input.split("\n---\n");
const env = parseDotenv(envText);
const required = (requiredLine ?? "").trim().split(/\s+/).filter(Boolean);
const missing = required.filter((k) => !(k in env) || env[k] === "");
console.log(`env=${JSON.stringify(env)}`);
console.log(`missing=${JSON.stringify(missing)} wouldExitWith=${missing.length ? 1 : 0}`)
process.on("exit", (code) => console.log(`exit handler: code=${code}`));   // synchronous
```

QUIZ

1. `process.argv[0]` and `[1]` are…

A. The first two user arguments

B. The Node executable and the script path — user arguments start at index 2

C. The cwd and the home directory

D. Always empty

2. Values in `process.env` are…

A. Typed

B. Always strings (or `undefined`) — parse and validate them once at startup

C. Numbers when numeric

D. Frozen

3. `exec("ls " + userInput)` is dangerous because…

A. `ls` is slow

B. It runs through a shell — input like `; rm -rf /` executes; use `execFile` / `spawn` with an argument array

C. It is asynchronous

D. It buffers output

4. After installing a `SIGTERM` handler, the process…

A. Exits automatically after the handler

B. Keeps running until you call `process.exit` yourself — the handler replaces the default exit

C. Ignores the signal

D. Restarts

5. `worker_threads` are appropriate for…

A. Database queries

B. CPU-bound work that would block the event loop; I/O is already non-blocking and gains nothing

C. Reading files

D. HTTP requests

ANSWER KEY · THE PROCESS — ARGUMENTS, ENVIRONMENT, EXIT CODES, SIGNALS, CHILD PROCESSES AND WORKERS

1. B `process.argv.slice(2)`.

2. B `PORT="3000"` is a string; precedence flag > env > file > default.

3. B `execFile` / `spawn` start the program directly with no shell parsing.

4. B Stop accepting work, drain, close, then exit within the grace period; `SIGKILL` cannot be handled.

5. B Workers have separate loops and heaps and communicate by message passing.

CHECKPOINT · 24 MIN

Checkpoint — Modules and Node

This checkpoint covers CommonJS versus ES modules (loading, caching, copies versus live bindings, module type detection, interop, cycles, dynamic `import()`), `package.json` fields, the resolution algorithm, semver ranges and lockfiles, the core modules (`path`, `fs`, `process`, `events`, `url`, `crypto`), `Buffer` and streams with `pipeline` and backpressure, and the process surface — arguments, environment, exit codes, signals, child processes and workers.

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:

- How Node decides whether a `.js` file is CJS or ESM; why ESM imports need extensions; what a live binding is; what CJS does on a circular `require`.
- The three steps of resolving `require("x")`; `^` versus `~` and what `^0.x` means; why `npm ci` and a committed lockfile.
- `path.join` versus `resolve`; the three `fs` styles; what `emit("error")` with no listener does; `exitCode` versus `exit()`.
- Why `pipeline` beats `pipe`; what backpressure is; why `chunk.toString()` per chunk breaks on UTF-8.
- `exec` versus `execFile` / `spawn`; what a `SIGTERM` handler must do; when to use `worker_threads`.

The programs are a module-graph loader that reproduces CommonJS load order and detects cycles, a stream pipeline that parses and summarises log lines with a `Transform` in object mode, and a configuration CLI that merges flags, environment and defaults with the standard precedence.

EXERCISE 1

Load a module graph the CommonJS way

Lines `module <name>: <dep> <dep>…` declare modules and what they require (in order); `entry <name>` names the entry point. Simulate CommonJS loading from the entry: a module is marked *loading*, pushed on the stack and recorded in the **start** order; each dependency is required in turn — a *loaded* module is a cache hit, a *loading* one is a **cycle** recorded as `a → b → a` (the stack from the repeated module, plus it) and returns its partial exports, an undeclared one is recorded as **missing**; then the module is marked loaded and recorded in the **finish** order. Print `start: …`, `finish: …`, `cycles: <"; "-joined or none>`, `missing: <names or none> unused: <declared modules never loaded, or none>`.

Example:

```
module app: config db routes
module config:
module db: config logger
module logger: config
module routes: db handlers
module handlers: routes
module unused: config
entry app
```

→

```
start: app config db logger routes handlers
finish: config logger db handlers routes app
cycles: routes → handlers → routes
missing: none unused: unused
```

INPUT 1

```
module app: config db routes
module config:
module db: config logger
module logger: config
module routes: db handlers
module handlers: routes
module unused: config
entry app
```

OUTPUT 1

```
start: app config db logger routes
handlers
finish: config logger db handlers routes
app
cycles: routes → handlers → routes
missing: none unused: unused
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const deps = new Map();       // name → [deps]
let entry;
for (const line of lines) {
  const [cmd, name, ...rest] = line.trim().split(/\s+/);
  if (cmd === "module") deps.set(name.replace(/:$/, ""), rest);
  else if (cmd === "entry") entry = name;
}
const started = [], finished = [], cycles = [], missing = [];
const state = new Map();      // name → "loading" | "loaded"
const stack = [];
function load(name) {
  if (state.get(name) === "loaded") return;                                    // require
  if (state.get(name) === "loading") {                                         // a cycle
    cycles.push(`${stack.slice(stack.indexOf(name)).join(" → ")} → ${name}`);
    return;
  }
  if (!deps.has(name)) { missing.push(name); return; }
  state.set(name, "loading");
  stack.push(name);
  started.push(name);
  for (const d of deps.get(name)) load(d);                                      // each r
  stack.pop();
  state.set(name, "loaded");
  finished.push(name);
}
load(entry);
console.log(`start: ${started.join(" ")}`);
console.log(`finish: ${finished.join(" ")}`);
console.log(`cycles: ${cycles.length ? cycles.join("; ") : "none"}`);
console.log(`missing: ${missing.length ? [...new Set(missing)].join(",") : "none"} unused
```

EXERCISE 2

A log-summarising stream pipeline

Line 1 is the minimum level; each following line is `<ISO time> <LEVEL> <message>`. Build a `pipeline` of `Readable.from(lines)` → a parsing `Transform` (readable object mode; a line that does not match `^(\S+)\s+(DEBUG|INFO|WARN|ERROR)\s+(.*)$` is counted as malformed and dropped) → a filtering `Transform` (keep levels ≥ minimum) → a summarising object-mode `Writable` that counts per level, per hour (characters 11–12 of the time), and remembers the first `ERROR`. Print `seen=<lines> malformed=<n> kept=<n>`, `levels: <LEVEL=n in DEBUG,INFO,WARN,ERROR order, or none>`, `byHour: <HHh=n sorted, or none>`, `firstError: <time message or none>`.

Example: `WARN` and five log lines including one garbage line →

```
seen=6 malformed=1 kept=3
levels: WARN=1 ERROR=2
byHour: 10h=1 11h=2
firstError: 2024-01-01T10:16:00Z db timeout
```

INPUT 1

```
WARN
2024-01-01T10:15:00Z INFO server started
2024-01-01T10:16:00Z ERROR db timeout
this line is garbage
2024-01-01T11:02:00Z WARN disk 91% full
2024-01-01T11:03:00Z ERROR db timeout
again
2024-01-01T11:04:00Z DEBUG noise
```

OUTPUT 1

```
seen=6 malformed=1 kept=3
levels: WARN=1 ERROR=2
byHour: 10h=1 11h=2
firstError: 2024-01-01T10:16:00Z db
timeout
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const { Readable, Transform, Writable } = require("node:stream");
const { pipeline } = require("node:stream/promises");
const LEVELS = ["DEBUG", "INFO", "WARN", "ERROR"];
async function main() {
  const [minLevel, ...raw] = input.split("\n").filter((l) => l.trim() !== "");
  let seen = 0, malformed = 0;
  const parse = new Transform({
    readableObjectMode: true,
    transform(chunk, _enc, cb) {
      seen++;
      const m = /^(\S+)\s+(DEBUG|INFO|WARN|ERROR)\s+(.*)$/.exec(chunk.toString().trim());
      if (!m) { malformed++; return cb(); }          // drop, but keep going
      cb(null, { time: m[1], level: m[2], message: m[3] });
    },
  });
  const filter = new Transform({ objectMode: true, transform(rec, _enc, cb) { if (LEVELS.
  const counts = new Map(), byHour = new Map();
  let firstError = null, kept = 0;
  const summarise = new Writable({
    objectMode: true,
    write(rec, _enc, cb) {
      kept++;
      counts.set(rec.level, (counts.get(rec.level) ?? 0) + 1);
      const hour = rec.time.slice(11, 13);
      byHour.set(hour, (byHour.get(hour) ?? 0) + 1);
      if (rec.level === "ERROR" && firstError === null) firstError = `${rec.time} ${rec.m
      cb();
    },
  });
  await pipeline(Readable.from(raw), parse, filter, summarise);
  console.log(`seen=${seen} malformed=${malformed} kept=${kept}`);
  console.log(`levels: ${LEVELS.filter((l) => counts.has(l)).map((l) => `${l}=${counts.ge
  console.log(`byHour: ${[...byHour].sort(([a], [b]) => a.localeCompare(b)).map(([h, n])
  console.log(`firstError: ${firstError ?? "none"}`);
}
main();
```

EXERCISE 3

A configuration CLI: flag > env > default

Defaults are `{ port: 3000, host: "localhost", debug: false, workers: 1 }`, read from the environment as `APP_PORT`, `APP_HOST`, `APP_DEBUG`, `APP_WORKERS`. Line 1 of the input is the argv (possibly blank) — only `--key=value`, `--flag` (true) and `--no-flag` (false) matter; the following lines are `KEY=VALUE` environment entries. For each key take the flag, else the env value, else the default, coercing to the default's type (integers for numbers; `true` / `1` / `false` / `0` for booleans; anything else is a `TypeError("<key> must be an integer, got <JSON>")` / `("<key> must be true/false, got <JSON>")`). Validate `port` 1–65535 and `workers` ≥ 1. Print `config=<JSON>` and `sources: <key=flag|env|default …>` and `wouldExitWith=0`, or `invalid: <messages joined by "; ">` and `wouldExitWith=1`.

Example: `--port=8080 --debug` then `APP_HOST=0.0.0.0`, `APP_WORKERS=4`, `APP_PORT=9999` →

```
config={"port":8080,"host":"0.0.0.0","debug":true,"workers":4}
sources: port=flag host=env debug=flag workers=env
wouldExitWith=0
```

The flag beats `APP_PORT`; `--workers=0 --port=70000` → `invalid: port must be 1-65535, got 70000; workers must be at least 1, got 0`.

INPUT 1

```
--port=8080 --debug
APP_HOST=0.0.0.0
APP_WORKERS=4
APP_PORT=9999
```

OUTPUT 1

```
config=
{"port":8080,"host":"0.0.0.0","debug":true,"workers":4}
sources: port=flag host=env debug=flag workers=env
wouldExitWith=0
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const DEFAULTS = { port: 3000, host: "localhost", debug: false, workers: 1 };
const ENV_KEYS = { port: "APP_PORT", host: "APP_HOST", debug: "APP_DEBUG", workers: "APP_
const allLines = input.split("\n");
const argvLine = allLines[0] ?? "";                                    // line 1 is argv ev
const envLines = allLines.slice(1).filter((l) => l.includes("="));
const flags = {};
for (const a of argvLine.trim().split(/\s+/).filter(Boolean)) {
  if (!a.startsWith("--")) continue;
  const eq = a.indexOf("=");
  if (eq !== -1) flags[a.slice(2, eq)] = a.slice(eq + 1);
  else if (a.startsWith("--no-")) flags[a.slice(5)] = "false";
  else flags[a.slice(2)] = "true";
}
const env = Object.fromEntries(envLines.map((l) => { const i = l.indexOf("="); return [l.
function coerce(key, text) {
  const kind = typeof DEFAULTS[key];
  if (kind === "number") { const n = Number(text); if (!Number.isInteger(n)) throw new Ty
  if (kind === "boolean") { if (text === "true" || text === "1") return true; if (text ==
  return text;
}
const config = {}, sources = {}, errors = [];
for (const key of Object.keys(DEFAULTS)) {
  try {
    if (key in flags) { config[key] = coerce(key, flags[key]); sources[key] = "flag"; }
    else if (ENV_KEYS[key] in env) { config[key] = coerce(key, env[ENV_KEYS[key]]); sourc
    else { config[key] = DEFAULTS[key]; sources[key] = "default"; }
  } catch (err) { errors.push(err.message); }
}
if (Number.isInteger(config.port) && (config.port < 1 || config.port > 65535)) errors.pus
if (Number.isInteger(config.workers) && config.workers < 1) errors.push(`workers must be
if (errors.length) {
  console.log(`invalid: ${errors.join("; ")}`);
  console.log("wouldExitWith=1");
} else {
  console.log(`config=${JSON.stringify(config)}`);
  console.log(`sources: ${Object.entries(sources).map(([k, s]) => `${k}=${s}`).join(" ")}
  console.log("wouldExitWith=0");
}
```

CHECKPOINT QUESTIONS

1. Module-level state (a `const cache = new Map()` at the top of a file) is…

A. Re-created per import

B. Created once per process — modules are evaluated once and cached, so it is a process-wide singleton

C. Per function call

D. Per request

2. ESM's `import.meta.url` replaces…

A. `process.argv`

B. `__filename` / `__dirname`, which do not exist in ES modules — convert with `fileURLToPath`

C. `require.cache`

D. `module.exports`

3. Dynamic `import()` …

A. Only works in ESM

B. Works in both module systems, returns a promise of the namespace, and is the only import form usable inside functions

C. Is synchronous

D. Cannot load packages

4. Resolving a bare specifier like `lodash` walks…

A. Only the project root

B. Up the directory tree checking each `node_modules/` until the filesystem root

C. The global installation only

D. `PATH`

5. `peerDependencies` say…

A. Install this for me

B. The host application must provide this package (plugins, React components) — the package does not bundle its own copy

C. Optional extras

D. Development tools

6. `fs.existsSync(p)` followed by `fs.readFileSync(p)` …

A. Is the safe pattern

B. Is a race — the file can vanish between the two calls; open it and handle `ENOENT`

C. Is faster

D. Is required by Node

7. `EventEmitter#emit` calls listeners…

A. On the next tick

B. Synchronously, in registration order, before `emit` returns

C. In parallel

D. Randomly

8. `crypto.createHash("sha256")` is appropriate for…

A. Storing passwords

B. Checksums and content addressing — passwords need `scrypt` / `bcrypt` / `argon2`

C. Generating session tokens

D. Encryption

9. A `Transform` that assumes each chunk is exactly one line…

A. Is correct

B. Is wrong — byte chunks are arbitrary slices; buffer the partial tail and emit it in `flush`

C. Is faster

D. Works for files only

10. Program output goes to stdout and messages to stderr because…

A. stderr is faster

B. A pipe or file consumes stdout; progress and diagnostics on it would corrupt the data

C. stdout is unbuffered

D. It is required by POSIX

11. `process.exitCode = 1` followed by returning normally…

A. Does nothing

B. Exits with code 1 once the event loop drains, after pending output is flushed

C. Exits immediately

D. Throws

12. `spawn("ffmpeg", args, { stdio: "inherit" })` versus `exec("ffmpeg …")`:

A. Identical

B. `spawn` streams the child's I/O and takes an argument array (no shell); `exec` buffers output (1 MB cap) and runs a shell string

C. `exec` is safer

D. `spawn` is synchronous

ANSWER KEY · CHECKPOINT — MODULES AND NODE

1. B Handy for pools and caches; a trap when you expected isolation.

2. B `createRequire(import.meta.url)` gives a `require` when you must have one.

3. B Lazy loading, conditional loading, plugins named at runtime.

4. B Which is why nested and hoisted `node_modules` both work.

5. B Two copies of React in one tree is the classic failure this prevents.

6. B Branch on `err.code`.

7. B A slow listener slows the emitter; a throw propagates to the `emit` call.

8. B Tokens: `randomBytes` / `randomUUID`; signatures: `createHmac`.

9. B `readline` exists so you do not have to write that buffering.

10. B `console.log` versus `console.error`.

11. B Prefer it to `process.exit(1)` in all but emergencies.

12. B `execFile` is the buffered-but-shell-free middle option.

CodeKairo · JavaScript study plan · Module 9: Modules and Node. Solve the exercises online to track your progress and earn XP.