Module 14 of 16

Modern JavaScript and its tooling

ES2020–2022 in everyday code; the 2023+ additions, TC39 stages, feature detection and spec-faithful polyfills; bundlers, tree-shaking, code splitting and source maps; ESLint, Prettier, test runners and CI; Node, Deno, Bun, edge runtimes and the web-standard core.

The language you have been learning is a moving target with a yearly release, a public proposal process, and a toolchain that rewrites, checks, tests and bundles code before it runs anywhere. Knowing the features is half of being current; knowing how they reach your runtime — transpiled, polyfilled, tree-shaken, split — and what checks stand between a keystroke and a deploy is the other half.

The first lesson covers the ES2020–2022 features that changed everyday code and their precise semantics; the second covers 2023 onward, the TC39 stages, and how to detect and polyfill correctly on a runtime that lags — which this plan's Node 16 does, so the exercises implement several newer features by hand. The build lesson walks the pipeline from resolution to source maps and explains what tree-shaking needs and what defeats it. The quality lesson covers ESLint's flat config, Prettier, the shape of a good test, mocking with restraint, and the hook-plus-CI wiring. The last lesson maps the runtimes — browsers, Node, Deno, Bun, edge isolates — and the web-standard core that makes code portable between them.

The exercises use every 2020–2022 feature, polyfill the 2023–2024 array and grouping methods to the spec, tree-shake a module graph, plan chunks, write a linter and a TAP test runner, detect features with `new Function`, resolve an import map — then a `structuredClone` polyfill with cycles, a miniature bundler, and a runtime-support planner.

CONTENTS

1	ES2020–ES2022 — the features that changed everyday code	13 min
2	ES2023 onward, and how features get into the language	12 min
3	Bundlers, transpilers and the build — what happens to your code before it runs	13 min
4	Linting, formatting and testing — the quality toolchain	12 min
5	Runtimes and the ecosystem — Node, Deno, Bun, browsers, edge, and the web-standard core	12 min
✓	Checkpoint — Modern JavaScript and its tooling	24 min

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

LESSON 1 · 13 MIN

ES2020–ES2022 — the features that changed everyday code

JavaScript gains features every year; most are small, a few rewire how ordinary code looks. Between 2020 and 2022 the language got the two operators that removed most defensive `if`s (`?.` and `??`), logical assignment, BigInt, the last two promise combinators, real class privacy, `Error.cause`, top-level `await` and a handful of methods that end common workarounds (`at`, `replaceAll`, `Object.hasOwn`). Everything here runs on Node 16 and in every evergreen browser, and this module's exercises use all of it. This lesson is the feature list with the precise semantics and the one pitfall of each.

Optional chaining `?.`

```
user?.address?.city          // undefined if user or address is null/undefined — i
user?.["home-address"]       // computed access
onDone?.(result)             // call only if defined
arr?.[0]
```

Short-circuits the **whole chain** on the first nullish link: `a?.b.c.d` evaluates to `undefined` without touching `.c` when `a` is nullish. It only guards `null` / `undefined` — a `0` or `""` continues. Do not sprinkle it everywhere: `?.` on a value that *should* exist hides a bug; use it where absence is a legitimate state.

Nullish coalescing `??` and logical assignment

```
const port = config.port ?? 3000;        // only null/undefined trigger the defaul
const label = config.label || "untitled";  // any falsy triggers — "" becomes "unt
opts.retries ??= 3;                       // assign only if nullish
opts.verbose ||= false;                   // assign if falsy
cache.entry &&= refresh(cache.entry);     // assign only if truthy
```

`??` versus `||` is the difference between "missing" and "falsy" — choose deliberately (module 7). Mixing `??` with `||` / `&&` without parentheses is a syntax error by design. The logical assignments **short-circuit**: `a ??= f()` does not call `f` when `a` is already set (unlike `a = a ?? f()`, which also always assigns — a difference visible with setters).

BigInt

```
const big = 9007199254740993n;                    // beyond Number.MAX_SAFE_INTEGER;
big + 1n;  big * 2n;  big ** 2n;  7n / 2n;      // 3n — integer division truncates
typeof big;                                       // "bigint"
big + 1;                                          // TypeError: cannot mix BigInt
BigInt(1.5);                                      // RangeError
JSON.stringify({ big });                          // TypeError — provide toJSON or
0n === 0; 0n == 0;                                // false, true
```

Arbitrary-precision integers for ids, timestamps in nanoseconds, cryptography, exact money arithmetic. Mixed arithmetic is an error on purpose; `Math` functions do not accept BigInt.

`Promise.allSettled` (2020) and `Promise.any` (2021)

Module 8 covered them: every outcome, and first success with `AggregateError` on total failure. `AggregateError` itself is 2021.

`globalThis`, `import()`, `import.meta`

`globalThis` is the global object in every environment (`window`, `self`, `global` were platform-specific). Dynamic `import()` and `import.meta` (`import.meta.url`) are 2020 and the basis of module 9's ESM story. `export * as ns from "mod"` completes the re-export forms.

Strings and arrays

```
"a-b-c".replaceAll("-", "+");                      // every occurrence without a r
[..."a1b2".matchAll(/\d/g)];                       // every match with groups (2020
[1, 2, 3].at(-1);  "abc".at(-1);                   // negative indices (2022)
Object.hasOwn(obj, "k");                           // the safe hasOwnProperty (2022
[1, [2, [3]]].flat(Infinity);  arr.flatMap(f);     // 2019, still underused
1_000_000; 0xFF_FF; 1_000n;                        // numeric separators (2021) — pu
```

Classes: fields, private members, static blocks (2022)

```
class Counter {
  static instances = 0;
  static { Counter.instances = 0; }        // static initialisation block — severa
  #count = 0;                              // private field
  count = 0;                               // public field
  #bump() { this.#count++; }               // private method
  static isCounter(x) { return #count in x; }   // brand check (16.4+)
  get value() { return this.#count; }
}
```

Module 5 covered the semantics; this is the year they became standard everywhere.

`Error.cause` (2022) and top-level `await` (2022)

`new Error("save failed", { cause })` — module 7's wrapping idiom. Top-level `await` in ES modules makes module evaluation asynchronous and importers wait — for configuration and connections at startup; not available in CommonJS.

Regular expressions

The `d` flag / `hasIndices` (2022) gives `match.indices` with `[start, end]` per group — for highlighting and error positions (module 10). `matchAll` requires `g`. Named groups and lookbehind were 2018.

`WeakRef` and `FinalizationRegistry` (2021)

Module 6: hold without retaining; rarely appropriate.

Feature detection, and when to polyfill

Everything above is syntax or built-ins. **Syntax** (`?.` , `??=` , `#x` , numeric separators) cannot be polyfilled — a runtime that lacks it fails to parse the file; only a transpiler (Babel, esbuild, TypeScript with a lower `target`) can rewrite it. **Built-ins** (`replaceAll` , `at` , `Object.hasOwn` , `Promise.any`) can be detected and filled: `if (!Array.prototype.at) { Object.defineProperty(Array.prototype, "at", { value(i) { … }, writable: true, configurable: true }); }` — non-enumerable, guarded, following the spec (`core-js` does this for everything). Detect features, never user agents.

Common mistakes

- `?.` on values that must exist (masking bugs); `a?.b.c` expecting `.c` to be guarded too (it is — the chain short-circuits — but `(a?.b).c` is not).
- `||` where `??` was meant for `0` / `""` ; `??` mixed with `||` without parentheses.
- Mixing BigInt and Number; `JSON.stringify` on BigInt.
- Polyfilling syntax (impossible) or patching prototypes with enumerable properties.
- Assuming a feature exists because it is "standard" — check the runtime's version.

Interview angle

- *"`??` versus `||`?"* Nullish (null/undefined) versus falsy; `0` and `""` survive `??` .

"What does `a?.b.c` do when `a` is null?" The whole chain short-circuits to `undefined`; `.c` is never evaluated.

"Why can't BigInt mix with Number?" Precision would be ambiguous; the language forces explicit conversion.

"Can you polyfill optional chaining?" No — syntax needs transpiling; only built-ins can be polyfilled.

"Name three ES2022 features." Class fields/private members/static blocks, `Error.cause`, top-level `await`, `at()`, `Object.hasOwn`, the `d` flag.

Key takeaways

`?.` guards nullish links and short-circuits the chain; `??` / `??=` mean "missing", `||` / `||=` mean "falsy".

BigInt for big integers, never mixed implicitly with Number; `n` suffix; no JSON.

`allSettled` / `any` / `AggregateError`, `globalThis`, `import()` / `import.meta`, `matchAll`, `replaceAll`, `at`, `Object.hasOwn`, numeric separators.

2022: class fields, `#private`, static blocks, brand checks, `Error.cause`, top-level `await`, regex `d`.

Syntax must be transpiled; built-ins can be detected and polyfilled — detect features, not browsers.

EXERCISE 1

The 2020–2021 toolkit in one program

Line 1 is a JSON config, line 2 a number `n`, line 3 a sentence. Merge the config with the defaults `{ port: 3000, host: "localhost", retries: 3, debug: false }` using `??=` and, for contrast, with `||=`; print `merged=<JSON>` and `viaOr=<JSON> differs=<bool>`. Print `city=<config.owner?.address?.city ?? "unknown"> firstTag=<config.tags?.[0] ?? "none"> onReady=<typeof config.onReady?.call>`. Compute `n!` with BigInt and print `<n>! = <value> digits=<> typeof=bigint safe=<≤ MAX_SAFE_INTEGER>`, then `bigint+number=<constructor name of the error from fact + 1> bigint+1n=true`. Finally title-case the sentence with `matchAll` and print `titleCase=<> dashes=<sentence.replaceAll(" ", "-")> million=<1_000_000> hex=<0xff_ff>`.

Example: `{"port":0,"host":"","owner":{"address":{"city":"Paris"}},"tags":[]}` / `25` / `hello brave new world` →

```
merged={"port":0,"host":"","owner":{"address":{"city":"Paris"}},"tags":[],"retries":3,"de
viaOr={"port":3000,"host":"localhost","owner":{"address":{"city":"Paris"}},"tags":[],"ret
city=Paris firstTag=none onReady=undefined
25! = 15511210043330985984000000 digits=26 typeof=bigint safe=false
bigint+number=TypeError bigint+1n=true
titleCase=Hello Brave New World dashes=hello-brave-new-world million=1000000 hex=65535
```

`??=` kept the deliberate `0` and `""`; `||=` overwrote them.

INPUT 1

```
{"port":0,"host":"","owner":{"address":
{"city":"Paris"}},"tags":[]}
25
hello brave new world
```

OUTPUT 1

```
merged={"port":0,"host":"","owner":
{"address":{"city":"Paris"}},"tags":
[],"retries":3,"debug":false}
viaOr=
{"port":3000,"host":"localhost","owner":
{"address":{"city":"Paris"}},"tags":
[],"retries":3,"debug":false} differs=true
city=Paris firstTag=none onReady=undefined
25! = 15511210043330985984000000 digits=26
typeof=bigint safe=false
bigint+number=TypeError bigint+1n=true
titleCase=Hello Brave New World
dashes=hello-brave-new-world
million=1000000 hex=65535
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) ⇒ l.trim() ≠
const DEFAULTS = { port: 3000, host: "localhost", retries: 3, debug: false };
const config = JSON.parse(lines[0]);
const n = Number(lines[1]);
const text = lines[2] ?? "";
const merged = { ...config };
for (const [k, v] of Object.entries(DEFAULTS)) merged[k] ??= v;          // only null/und
const viaOr = { ...config };
for (const [k, v] of Object.entries(DEFAULTS)) viaOr[k] |⊨ v;            // for contrast:
console.log(`merged=${JSON.stringify(merged)}`);
console.log(`viaOr=${JSON.stringify(viaOr)} differs=${JSON.stringify(merged) ≠≡ JSON.str
console.log(`city=${config.owner?.address?.city ?? "unknown"} firstTag=${config.tags?.[0]
let fact = 1n;
for (let i = 2n; i ⩽ BigInt(n); i++) fact *= i;
console.log(`${n}! = ${fact} digits=${String(fact).length} typeof=${typeof fact} safe=${f
let mixed;
try { mixed = fact + 1; } catch (err) { mixed = err.constructor.name; }
console.log(`bigint+number=${mixed} bigint+1n=${fact + 1n ≡≡ fact + BigInt(1)}`);
const words = [...text.matchAll(/\b(\w)(\w*)\b/g)].map((m) ⇒ m[1].toUpperCase() + m[2]);
console.log(`titleCase=${words.join(" ")} dashes=${text.replaceAll(" ", "-")} million=${1
```

EXERCISE 2

ES2022 class features, cause and indices

Write `Counter` with `static instances`, a `static registry` filled in a **static block**, private `#count` and `#history`, a private method `#bump()`, `inc(times)`, `get value()`, `get last()` (via `this.#history.at(-1) ?? null`) and `static isCounter(x)` using the brand check `#count in x`. Commands: `new <name>` → `new <name>: instances=<n>`; `inc <name> <times>` → `<name>: value=<n>`; `check <name>` → `isCounter(<name>)=<> isCounter(plain)=<for a plain {name}> hasOwn(name)=<Object.hasOwn(counter, "name")> hasOwn(value)=<…>`; `last <name>` → `<name>: last=<n|null>`; `fail <name> <message>` throws a `RangeError("inner: <message>")`, wraps it in `Error("outer: <name> failed", { cause })` and prints `<message> ← <cause.name>: <cause.message>`. Finally print `indices: word=<start-end> host=<start-end>` from `/(?<word>\w+)@(?<host>\w+)/d` on `"mail ada@example now"`.

Example: `new a`, `inc a 3`, `check a`, `last a`, `fail a disk full` →

```
new a: instances=1
a: value=3
isCounter(a)=true isCounter(plain)=false hasOwn(name)=true hasOwn(value)=false
a: last=3
outer: a failed ← RangeError: inner: disk
indices: word=5-8 host=9-16
```

(`value` is an accessor on the prototype, so it is not an own property.)

INPUT 1

```
new a
new b
inc a 3
inc b 1
check a
last a
last b
fail a disk full
```

OUTPUT 1

```
new a: instances=1
new b: instances=2
a: value=3
b: value=1
isCounter(a)=true isCounter(plain)=false
hasOwn(name)=true hasOwn(value)=false
a: last=3
b: last=1
outer: a failed ← RangeError: inner: disk
indices: word=5-8 host=9-16
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
class Counter {
  static instances = 0;
  static registry;
  static { Counter.registry = new Map(); }        // static initialisation block: several
  #count = 0;
  #history = [];
  constructor(name) { this.name = name; Counter.instances++; Counter.registry.set(name, t
  #bump() { this.#count++; this.#history.push(this.#count); }   // private method
  inc(times = 1) { for (let i = 0; i < times; i++) this.#bump(); return this; }
  get value() { return this.#count; }
  get last() { return this.#history.at(-1) ?? null; }           // at(-1): the last eleme
  static isCounter(x) { return typeof x === "object" && x !== null && #count in x; }   //
}
for (const line of lines) {
  const [cmd, name, arg] = line.trim().split(/\s+/);
  try {
    if (cmd === "new") console.log(`new ${name}: instances=${new Counter(name) && Counter
    else if (cmd === "inc") console.log(`${name}: value=${Counter.registry.get(name).inc(
    else if (cmd === "check") console.log(`isCounter(${name})=${Counter.isCounter(Counter
    else if (cmd === "last") console.log(`${name}: last=${Counter.registry.get(name).last
    else if (cmd === "fail") {
      try { throw new RangeError(`inner: ${arg ?? name}`); }
      catch (inner) { throw new Error(`outer: ${name} failed`, { cause: inner }); }
    }
  } catch (err) {
    console.log(`${err.message} ← ${err.cause?.name}: ${err.cause?.message}`);
  }
}
const m = /(?<word>\w+)@(?<host>\w+)/d.exec("mail ada@example now");
console.log(`indices: word=${m.indices.groups.word.join("-")} host=${m.indices.groups.hos
```

QUIZ

1. `config.port ??= 3000` when `config.port` is `0` …

A. Sets it to 3000

B. Leaves it `0` — `??=` assigns only for `null` / `undefined`; `|=` would overwrite the falsy 0

C. Throws

D. Sets it to `NaN`

2. `a?.b.c` when `a` is `undefined` …

A. Throws on `.c`

B. Evaluates to `undefined` — the whole chain short-circuits at the first nullish link

C. Returns `null`

D. Is a syntax error

3. `10n + 1` …

A. Is `11n`

B. Throws a `TypeError` — BigInt and Number never mix implicitly; convert with `BigInt(1)` or `Number(10n)`

C. Is `11`

D. Is `"101"`

4. Which cannot be polyfilled?

A. `Array.prototype.at`

B. Optional chaining `?.` — it is syntax; a runtime without it fails to parse the file, so it must be transpiled

C. `Object.hasOwn`

D. `Promise.any`

5. `#count in obj` …

A. Checks a public property

B. Tests whether `obj` carries this class's private field — a brand check that cannot be faked with a plain object

C. Throws for non-instances

D. Is a syntax error

ANSWER KEY · ES2020–ES2022 — THE FEATURES THAT CHANGED EVERYDAY CODE

1. B The nullish family means "missing", the logical-or family means "falsy".

2. B `(a?.b).c` would throw — the parentheses end the chain.

3. B `JSON.stringify` also refuses BigInt.

4. B Built-ins can be detected and filled; syntax needs a transpiler.

5. B Available since Node 16.4.

LESSON 2 · 12 MIN

ES2023 onward, and how features get into the language

Node 16 — this plan's runtime — stops at ES2022, so everything in this lesson is **reading**: the features you will meet in newer runtimes and codebases, what each replaces, and the process that produces them, so you can judge a proposal's maturity before adopting it. The exercises write polyfills for several of these on Node 16, which is the best way to understand exactly what they do — and the realistic situation of supporting a runtime older than the feature.

ES2023

`findLast` / `findLastIndex` — search from the end without `reverse()` copies.

Change-array-by-copy: `toSorted(cmp)`, `toReversed()`, `toSpliced(start, del, ...items)`, `with(index, value)` — non-mutating twins of `sort` / `reverse` / `splice` /index assignment. They end the `[...arr].sort()` idiom and make immutable updates (module 11) one call. `with(-1, x)` accepts negative indices.

Hashbang (`#!/usr/bin/env node`) formally allowed at the top of scripts and modules.

Symbols as WeakMap keys (non-registered symbols).

ES2024

`Object.groupBy(items, keyFn)` → a null-prototype object of arrays; `Map.groupBy` → a Map (any key type). The group-by idiom (modules 4, 6, 11) becomes one call.

`Promise.withResolvers()` → `{ promise, resolve, reject }` — the "deferred" pattern without the executor dance, for bridging events and queues (module 8's async queue).

`Array.fromAsync(asyncIterable)` — collect an async iterable into an array (`for await` + push in one call).

Regex `v` flag — set notation and string properties in character classes (`[\p{L}--[a-z]]`), a superset of `u` with stricter escaping.

ArrayBuffer transfer/resizable buffers, `Atomics.waitAsync` — for workers and WASM.

Well-formed Unicode strings: `isWellFormed()` / `toWellFormed()` — detect and repair lone surrogates (module 10).

ES2025

Set methods: `union`, `intersection`, `difference`, `symmetricDifference`, `isSubsetOf`, `isSupersetOf`, `isDisjointFrom` — module 6's hand-written algebra becomes built-in.

Iterator helpers: `iter.map(f).filter(p).take(n).toArray()` on any iterator, plus `Iterator.from` — module 6's lazy pipeline as methods.

`RegExp.escape` — the `escapeRegExp` one-liner, standardised.

JSON modules (`import data from "./x.json" with { type: "json" }`) and **import attributes**.

`Promise.try` — run a sync-or-async function inside a promise chain.

Duplicate named groups in regex alternatives.

Near the finish line (stage 3 at the time of writing)

Explicit resource management: `using res = open()` / `await using` — deterministic disposal via `Symbol.dispose`, like C#'s `using`/Python's `with`; TypeScript 5.2 ships it.

Temporal — the replacement for `Date`: immutable, timezone-aware `PlainDate`, `ZonedDateTime`, `Duration`, correct arithmetic; polyfills exist and it is the most anticipated addition in a decade.

Decorators — `@log` on classes and members, standardised after years of divergent TypeScript/Babel versions; TypeScript 5 implements the standard.

`Array.prototype.group` was renamed to `Object.groupBy` (web compatibility); **Records & Tuples** were withdrawn (2025) in favour of exploring composites; the **pipeline operator** (`▷`) and **pattern matching** remain stage 1–2 — do not plan on them.

How a feature becomes standard: TC39 stages

Ecma's TC39 committee advances proposals through stages: **0** (strawperson), **1** (problem accepted for exploration), **2** (draft spec text — the shape may still change), **2.7** (spec complete, awaiting tests — added 2023), **3** (candidate — implementations begin; the design is stable), **4** (finished — two shipping implementations and tests; goes into the next yearly edition). Practical rules: use stage 4 freely (transpile if your runtime lacks it); use stage 3 with a transpiler/polyfill only if the cost of a late change is acceptable; treat stage 2 and below as reading material. Babel's `preset-env` and TypeScript follow the same line — stage 3 or later.

Knowing your runtime

Feature availability is a table of engines and versions: Node's release schedule (even majors are LTS; Node 16 reached end-of-life in 2023, 18 in 2025), browser evergreen releases, and Safari's slower cadence. Tools read **browserslist** (`"browserslist": ["defaults", "not IE 11"]`) to decide what to transpile and polyfill; `caniuse.com`, `node.green` and MDN's compatibility tables are the references. In practice: pick a `target` matching the *oldest* runtime you support, let the transpiler handle syntax, and load polyfills (`core-js`) only for built-ins that runtime lacks — feature-detected, not version-sniffed.

Writing a polyfill correctly

```
if (!Array.prototype.toSorted) {
  Object.defineProperty(Array.prototype, "toSorted", {
    value(compareFn) { return [...this].sort(compareFn); },   // spec: copy, then
    writable: true, configurable: true, enumerable: false,    // enumerable: false
  });
}
```

Guard (`if (!…)`), define non-enumerable, match the spec's edge cases (negative indices in `with`, `RangeError` for out-of-range, `toSpliced` argument defaults), and prefer `core-js` for anything you did not read the spec for. Never polyfill something differently from the standard — pages that did that for `Array.prototype.flatten` are why the method is called `flat`.

Common mistakes

Using `toSorted` / `groupBy` / `structuredClone` on Node 16 or older Safari without a polyfill (a `TypeError: x is not a function` in production).

Adopting stage 1–2 proposals (▷, pattern matching) via Babel plugins in application code.

Version-sniffing instead of feature detection; enumerable prototype polyfills.

Mixing up `Object.groupBy` (null-prototype object) and `Map.groupBy` (Map) return types.

Expecting `with()` to mutate, or `toSpliced` to return the removed items (it returns the new array).

Interview angle

"Name the change-array-by-copy methods." `toSorted`, `toReversed`, `toSpliced`, `with` — non-mutating versions of the four mutators.

"What does `Object.groupBy` return?" A null-prototype object of arrays keyed by the callback's result; `Map.groupBy` returns a Map.

"What are the TC39 stages?" 0 strawperson → 1 proposal → 2 draft → 2.7 spec complete → 3 candidate (stable, implementations) → 4 finished (in the next edition).

"When is it safe to use a proposal?" Stage 4 freely; stage 3 with transpiler support and eyes open; below that, no.

"How do you support an older runtime?" Transpile syntax to its `target` and feature-detect polyfills for missing built-ins (browserslist + core-js).

Key takeaways

2023: `findLast`, `toSorted` / `toReversed` / `toSpliced` / `with`; 2024: `Object.groupBy` / `Map.groupBy`, `Promise.withResolvers`, `Array.fromAsync`, regex `v`, well-formed strings; 2025: Set methods, iterator helpers, `RegExp.escape`, JSON modules, `Promise.try`.

Coming: `using` disposal, Temporal, decorators; withdrawn/distant: records & tuples, pipeline, pattern matching.

TC39 stages 0–4; adopt at 4, cautiously at 3.

Know the oldest runtime you support; transpile syntax, feature-detect and polyfill built-ins, follow the spec exactly.

Node 16 lacks everything in this lesson — which is why the exercises implement several of these features by hand.

EXERCISE 1

Polyfill the change-array-by-copy methods

Node 16 lacks the ES2023 array methods. Using the starter's guarded `define(name, fn)` (non-enumerable, only when missing), implement `toSorted(cmp)`, `toReversed()`, `with(index, value)` (negative indices allowed; out of range throws a `RangeError`), `toSpliced(start, deleteCount, ...items)`, `findLast(pred)` and `findLastIndex(pred)` on `Array.prototype`. For the input array print `toSorted=<numeric asc> toReversed=<>`; `with(0, 99)=<> with(-1, 0)=<>`; `with(99)=<error constructor name> toSpliced(1, 2, 7)=<>`; `findLast(even)=<> findLastIndex(even)=<>`; and `original unchanged=<bool> forInKeys=<indices seen by for…in> enumerable(toSorted)=<descriptor flag>`.

Example: `[5, 3, 8, 1, 4]` →

```
toSorted=[1,3,4,5,8] toReversed=[4,1,8,3,5]
with(0, 99)=[99,3,8,1,4] with(-1, 0)=[5,3,8,1,0]
with(99)=RangeError toSpliced(1, 2, 7)=[5,7,1,4]
findLast(even)=4 findLastIndex(even)=4
original unchanged=true forInKeys=0,1,2,3,4 enumerable(toSorted)=false
```

An enumerable polyfill would have made `for…in` over every array yield `toSorted`.

INPUT 1

```
[5, 3, 8, 1, 4]
```

OUTPUT 1

```
toSorted=[1,3,4,5,8] toReversed=
[4,1,8,3,5]
with(0, 99)=[99,3,8,1,4] with(-1, 0)=
[5,3,8,1,0]
with(99)=RangeError toSpliced(1, 2, 7)=
[5,7,1,4]
findLast(even)=4 findLastIndex(even)=4
original unchanged=true
forInKeys=0,1,2,3,4
enumerable(toSorted)=false
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
function define(name, fn) {
  if (!Array.prototype[name]) Object.defineProperty(Array.prototype, name, { value: fn, w
}
define("toSorted", function (compareFn) { return [...this].sort(compareFn); });
define("toReversed", function () { return [...this].reverse(); });
define("with", function (index, value) {
  const i = index < 0 ? this.length + index : index;
  if (i < 0 || i >= this.length) throw new RangeError(`Invalid index : ${index}`);
  const copy = [...this]; copy[i] = value; return copy;
});
define("toSpliced", function (start, deleteCount, ...items) { const copy = [...this]; cop
define("findLast", function (pred) { for (let i = this.length - 1; i >= 0; i--) if (pred(
define("findLastIndex", function (pred) { for (let i = this.length - 1; i >= 0; i--) if (
const arr = JSON.parse(lines[0]);
const snapshot = JSON.stringify(arr);
console.log(`toSorted=${JSON.stringify(arr.toSorted((a, b) => a - b))} toReversed=${JSON.
console.log(`with(0, 99)=${JSON.stringify(arr.with(0, 99))} with(-1, 0)=${JSON.stringify(
let outOfRange;
try { arr.with(99, 1); outOfRange = "no error"; } catch (err) { outOfRange = err.construc
console.log(`with(99)=${outOfRange} toSpliced(1, 2, 7)=${JSON.stringify(arr.toSpliced(1,
console.log(`findLast(even)=${arr.findLast((x) => x % 2 === 0)} findLastIndex(even)=${arr
const keysSeen = [];
for (const k in arr) keysSeen.push(k);
console.log(`original unchanged=${JSON.stringify(arr) === snapshot} forInKeys=${keysSeen.
```

EXERCISE 2

Polyfill groupBy, withResolvers and fromAsync

Add guarded polyfills (`if (!X) X = …`) for `Object.groupBy(items, keyFn)` (a **null-prototype** object of arrays), `Map.groupBy` (a Map — any key type), `Promise.withResolvers()` (`{ promise, resolve, reject }`) and `Array.fromAsync(asyncIterable, mapFn)` . Over the input people (`name city age`) print `Object.groupBy: <city=names… sorted> nullProto=<Object.getPrototypeOf(groups) === null> inheritsToString=<"toString" in groups>` ; `Map.groupBy: adults=<names⟼ minors=<names⟼ keysAreBooleans=<bool>` grouping by `age >= 18` ; `withResolvers: <a value resolved from a setTimeout>` ; `fromAsync: <ages doubled via an async generator>` .

Example: `ada london 36` , `bob paris 17` , `cy london 22` , `dee rome 41` →

```
Object.groupBy: london=ada,cy paris=bob rome=dee nullProto=true inheritsToString=false
Map.groupBy: adults=ada,cy,dee minors=bob keysAreBooleans=true
withResolvers: settled from a timer
fromAsync: [72,34,44,82]
```

INPUT 1

```
ada london 36
bob paris 17
cy london 22
dee rome 41
```

OUTPUT 1

```
Object.groupBy: london=ada,cy paris=bob
rome=dee nullProto=true
inheritsToString=false
Map.groupBy: adults=ada,cy,dee minors=bob
keysAreBooleans=true
withResolvers: settled from a timer
fromAsync: [72,34,44,82]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
if (!Object.groupBy) Object.groupBy = (items, key) => {
  const groups = Object.create(null);                          // null prototype, like th
  let i = 0;
  for (const item of items) (groups[key(item, i++)] ??= []).push(item);
  return groups;
};
if (!Map.groupBy) Map.groupBy = (items, key) => {
  const groups = new Map();
  let i = 0;
  for (const item of items) { const k = key(item, i++); if (!groups.has(k)) groups.set(k,
  return groups;
};
if (!Promise.withResolvers) Promise.withResolvers = () => {
  let resolve, reject;
  const promise = new Promise((res, rej) => { resolve = res; reject = rej; });
  return { promise, resolve, reject };
};
if (!Array.fromAsync) Array.fromAsync = async (iterable, mapFn = (x) => x) => {
  const out = [];
  let i = 0;
  for await (const x of iterable) out.push(await mapFn(x, i++));
  return out;
};
const people = lines.map((l) => { const [name, city, age] = l.trim().split(/\s+/); returr
async function main() {
  const byCity = Object.groupBy(people, (p) => p.city);
  console.log(`Object.groupBy: ${Object.keys(byCity).sort().map((c) => `${c}=${byCity[c].
  const byAdult = Map.groupBy(people, (p) => p.age >= 18);
  console.log(`Map.groupBy: adults=${(byAdult.get(true) ?? []).map((p) => p.name).join(",
  const { promise, resolve } = Promise.withResolvers();
  setTimeout(() => resolve("settled from a timer"), 5);        // the deferred pattern: r
  console.log(`withResolvers: ${await promise}`);
  async function* ages() { for (const p of people) { await null; yield p.age; } }
  console.log(`fromAsync: ${JSON.stringify(await Array.fromAsync(ages(), (a) => a * 2))}`
}
main();
```

QUIZ

1. `arr.toSorted(cmp)` differs from `arr.sort(cmp)` in that it…

   A. Sorts descending

   B. Returns a sorted copy and leaves `arr` untouched

   C. Is faster

   D. Accepts no comparator

2. `Object.groupBy(items, fn)` returns…

   A. A `Map`

   B. A null-prototype object whose values are arrays — `Map.groupBy` returns a Map for non-string keys

   C. An array of pairs

   D. An iterator

3. A proposal at TC39 stage 3 is…

   A. Finished and in the spec

   B. A candidate: design stable, implementations underway, tests pending — usable with a transpiler if a late change is acceptable

   C. Just an idea

   D. Withdrawn

4. `Promise.withResolvers()` gives you…

   A. A resolved promise

   B. `{ promise, resolve, reject }` — the deferred pattern without capturing the executor's callbacks by hand

   C. A cancellable promise

   D. An async iterator

5. A prototype polyfill should be defined…

   A. With plain assignment

   B. Guarded (`if (!Array.prototype.x)`), non-enumerable via `Object.defineProperty` , and matching the spec's edge cases

   C. Always, overriding the native

   D. As an enumerable property

ANSWER KEY · ES2023 ONWARD, AND HOW FEATURES GET INTO THE LANGUAGE

1. B `toReversed`, `toSpliced` and `with` are the other copying twins (ES2023).

2. B Both landed in ES2024.

3. B Stage 4 goes into the next yearly edition; stages 0–2 are reading material.

4. B Handy for bridging events and queues to promises.

5. B Non-standard polyfills are why `flatten` had to become `flat`.

LESSON 3 · 13 MIN

Bundlers, transpilers and the build — what happens to your code before it runs

Almost no front-end JavaScript runs as written: a **build** parses every module, resolves imports, rewrites new syntax for older targets, removes code nothing uses, splits the result into chunks, minifies it, and emits source maps so errors still point at your lines. Servers increasingly go through the same pipeline (TypeScript, path aliases, bundling for cold-start speed). Understanding the pipeline is what lets you debug "works locally, breaks in production", keep bundles small, and configure a tool instead of copying a config from a blog post. This lesson covers the stages, the major tools and where each fits, tree-shaking and side effects, code splitting, targets and polyfills, source maps, and the package fields that make a library bundle well.

The stages

Resolve — turn `import x from "lib"` into a file path (module 9's algorithm plus `exports` conditions, aliases, extensions).

Load & transform — read the file; run loaders/plugins: TypeScript and JSX → JavaScript, CSS modules, images to URLs or inline data, JSON to objects.

Transpile — rewrite syntax newer than the **target** (optional chaining → ternaries for old browsers, class fields → constructor assignments) and inject **polyfills** for missing built-ins.

Build the graph — every import edge; detect circularities; mark side-effect-free modules.

Tree-shake — drop exports nothing imports (statically analysable only with ESM).

Split — decide chunks: entry points, dynamic `import()` boundaries, shared vendor code.

Minify — shorten names, drop whitespace and dead branches (`if (false)`), inline constants.

Emit — hashed file names for caching, an HTML/manifest that references them, and **source maps**.

Development builds skip 5–7 and serve modules individually for fast reloads; production builds do everything.

The tools

esbuild — Go, extremely fast transpile + bundle + minify; the engine inside Vite's dev server and many build scripts; less plugin depth.

Rollup — the ESM-native bundler with the best tree-shaking and output control; the classic choice for **libraries**; Vite's production bundler (Rolldown, its Rust successor, is replacing both — this repository's frontend builds with it).

Vite — the application dev server + build: native ESM in development (no bundling, instant start, HMR), Rollup/Rolldown for production, framework plugins for React/Vue/Svelte.

webpack — the long-standing, most configurable bundler; huge ecosystem; slower and more config-heavy; still common in large existing apps and Next.js (moving to Turbopack).

Parcel — zero-config bundler. **Turbopack**, **Rspack** — Rust rewrites of the webpack model.

Babel — the transpiler with the plugin ecosystem (JSX, proposals, `preset-env` with browserslist); slower than esbuild/swc, still used where a plugin is needed.

swc — Rust transpiler used by Next.js and Jest alternatives; **tsc** for type-checking (and emitting, if you want).

Rule of thumb today: applications → Vite; libraries → Rollup/tsup (esbuild-based); scripts → tsx/esbuild; type-check with tsc separately.

Tree-shaking and side effects

Tree-shaking removes **unused exports**, which requires knowing statically what is imported — ESM's static `import` / `export` make that possible; CommonJS's dynamic `require` largely defeats it. Two things block it:

Side effects: a module that *does* something at import time (registers a polyfill, patches a global, adds CSS) cannot be dropped even if nothing imports its exports. `"sideEffects": false` in a package's `package.json` tells bundlers every module is pure (or an array lists the impure ones); libraries that omit it ship whole.

Non-analysable patterns: `export default { a, b }` (one object — importing `a` keeps `b`), re-exporting a whole CommonJS module, or property access on a namespace at runtime.

Import named exports from ESM packages (`import { debounce } from "lodash-es"` not `import _ from "lodash"`), keep modules pure, and check the output: a bundle analyser (`rollup-plugin-visualizer`, `source-map-explorer`) shows exactly what shipped and why.

Code splitting

One giant bundle means every visitor downloads every page's code. Splitting points:

Dynamic `import()` — the bundler turns each into a separate chunk loaded on demand (`React.lazy`, route-level splitting — this repository lazy-loads every route).

Multiple entries and **shared chunks** — modules imported by several chunks are hoisted into a common chunk so they download once (module 9's `node_modules` dedupe, at the bundle level).

Vendor splitting — third-party code changes rarely; a separate hashed chunk stays cached across your deploys.

Too many tiny chunks cost round trips (HTTP/2 helps); too few cost download size. Hashed names (`main.3f9a1c.js`) with immutable cache headers make caching safe. Preloading the next likely chunk (`<link rel="modulepreload">`, or a hover-prefetch) hides the latency.

Targets, browserslist, polyfills

The `target` (esbuild/TypeScript) or **browserslist** query (Babel, Vite, Autoprefixer) decides which syntax gets rewritten. Lower targets mean more, slower output — `async/await` into generator state machines, classes into functions. Set the target to the **oldest supported runtime**, not to ES5 "to be safe". Built-ins are separate: `core-js` via `preset-env`'s `useBuiltIns: "usage"` injects only the polyfills the code uses for the targets that lack them. Modern default: `"defaults"` browserslist (roughly 2–3 years of evergreen browsers), no polyfills at all for most apps.

Source maps

A `.map` file (or inline comment) maps every position in the output back to the source file, line, column and original name. Browsers and Node (`--enable-source-maps`) apply them for stack traces and breakpoints; error trackers (Sentry) ingest them so production traces read like development. Emit them for every production build; whether to *serve* them publicly is a policy choice (they expose source) — many teams upload to the tracker only. Without them a production error reads `main.js:1:48213` (module 7).

Environment and constants

Bundlers substitute `process.env.NODE_ENV` / `import.meta.env.*` (Vite: only `VITE_*` variables, inlined at build time — this repository's `VITE_API_URL`) and `define`d constants, then dead-code-eliminate the branches (`if (process.env.NODE_ENV ≠= "production")` vanishes in production). Anything inlined is public — secrets never go into a front-end build.

Shipping a library

`package.json` for a library that bundles well: `"type": "module"`, `"exports"` with `import` / `require` / `types` conditions, `"sideEffects": false` (or the honest list), `"files": ["dist"]`, ESM output (plus CJS if consumers need it), **no** bundling of dependencies (declare them; let the app's bundler dedupe), `.d.ts` alongside, source maps included, and a `target` no lower than your consumers need.

Common mistakes

- Importing a whole CJS utility library (`import _ from "lodash"`) into a tree-shaken app.
- Missing `"sideEffects": false` in a pure library (ships everything); marking it `false` on a library with a CSS import (drops the CSS).
- Transpiling to ES5 for a modern audience — bigger, slower output.
- No source maps in production, or serving them with secrets in comments.

Secrets in `VITE_*` / `NEXT_PUBLIC_*` variables.

Ignoring the bundle analyser until the app is slow.

Interview angle

"What does a bundler do?" Resolves the module graph, transforms and transpiles, tree-shakes, splits into chunks, minifies, emits hashed files with source maps.

"What is tree-shaking and what defeats it?" Removing unused exports using ESM's static structure; side effects and dynamic/CommonJS patterns block it.

"How do you reduce bundle size?" Named ESM imports, route-level dynamic imports, vendor splitting, a modern target, no unneeded polyfills, and an analyser to find the culprits.

"Why source maps?" Production stack traces and breakpoints map back to original code; upload to the error tracker.

"esbuild versus Rollup versus Vite versus webpack?" Fast transpiler/bundler; ESM library bundler with the best output; app dev server + build; the configurable incumbent.

Key takeaways

Build stages: resolve → transform → transpile → graph → tree-shake → split → minify → emit (+ source maps).

Vite for apps, Rollup/tsup for libraries, esbuild/tsx for scripts, tsc for types; Babel/swc where plugins demand.

Tree-shaking needs ESM, named imports and honest `sideEffects`; check with an analyser.

Split at routes with dynamic `import()`, share vendor chunks, hash names for caching; target the oldest supported runtime, polyfill by usage.

Source maps always; inlined env is public; libraries ship ESM with `exports`, `sideEffects` and types.

EXERCISE 1

Tree-shake a module graph

The input JSON is `{ entry, modules }` where each module has `exports` (names), `imports` (`{ dependency: [names] }`) and optionally `sideEffects: true`. Walk the graph from the entry: a module is **reachable** if imported (even with an empty name list); an export is **used** if some reachable module imports it (every export of the entry counts as used). Print `kept: <module[usedExports,…] …>` (alphabetical; `-` when nothing is used), `dropped exports: <module:export …>`, `dropped modules: <unreachable, or reachable with no used exports and no side effects>`, and `kept for side effects: <reachable modules with no used exports but sideEffects: true>`.

Example: an `app` importing `slugify` from `utils`, `log` from `logger` and nothing from `analytics` (a side-effect module), with unreachable `legacy` and `theme` →

```
kept: analytics[-] app[main] logger[log] utils[slugify]
dropped exports: analytics:track logger:warn utils:debounce utils:throttle
dropped modules: legacy theme
kept for side effects: analytics
```

INPUT 1

```
{"entry":"app","modules":{"app":{"exports":
["main"],"imports":{"utils":
["slugify"],"analytics":[],"logger":
["log"]}},"utils":{"exports":
["slugify","debounce","throttle"],"imports":
{"logger":["log"]}},"logger":{"exports":
["log","warn"],"imports":{}},"analytics":
{"exports":["track"],"imports":
{},"sideEffects":true},"legacy":{"exports":
["old"],"imports":{"utils":
["throttle"]}},"theme":{"exports":
[],"imports":{}}}}
```

OUTPUT 1

```
kept: analytics[-] app[main] logger[log]
utils[slugify]
dropped exports: analytics:track
logger:warn utils:debounce utils:throttle
dropped modules: legacy theme
kept for side effects: analytics
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const graph = JSON.parse(input);      // { entry, modules: { name: { exports: [...], impo
const { entry, modules } = graph;
const reachable = new Set();
const used = new Map();                // module → Set of imported names
(function visit(name) {
  if (reachable.has(name) || !modules[name]) return;
  reachable.add(name);
  for (const [dep, names] of Object.entries(modules[name].imports ?? {})) {
    if (!used.has(dep)) used.set(dep, new Set());
    for (const n of names) used.get(dep).add(n);
    visit(dep);
  }
})(entry);
const kept = [], droppedExports = [], droppedModules = [], sideEffectOnly = [];
for (const name of Object.keys(modules).sort()) {
  const mod = modules[name];
  if (!reachable.has(name)) { droppedModules.push(name); continue; }
  const wanted = name === entry ? new Set(mod.exports) : (used.get(name) ?? new Set());
  const keptExports = mod.exports.filter((e) => wanted.has(e));
  const unused = mod.exports.filter((e) => !wanted.has(e));
  if (keptExports.length === 0 && name !== entry) {
    if (mod.sideEffects) sideEffectOnly.push(name);          // nothing imported, but it
    else { droppedModules.push(name); continue; }
  }
  kept.push(`${name}[${keptExports.join(",") || "-"}]`);
  for (const e of unused) droppedExports.push(`${name}:${e}`);
}
console.log(`kept: ${kept.join(" ")}`);
console.log(`dropped exports: ${droppedExports.join(" ") || "-"}`);
console.log(`dropped modules: ${droppedModules.join(" ") || "-"}`);
console.log(`kept for side effects: ${sideEffectOnly.join(" ") || "-"}`);
```

EXERCISE 2

Plan the chunks

The input JSON is `{ routes: { name: [modules] }, sizes: { module: kB }, vendor: [modules] }`. Build chunks: `vendor` (vendor modules used by any route), `shared` (non-vendor modules used by two or more routes), and one `route:<name>` chunk per route holding its remaining modules; modules used by no route are not shipped. Print each chunk as `<name>: [<modules sorted>] <size> kB` (vendor, shared, then routes in input order), then `total shipped=<sum> kB (naive per-route copies would be <sum over routes of their full module lists> kB)`, then `initial load <route>: <vendor + shared + route chunk> kB` per route.

Example (three routes sharing react/router/charts as vendor and shared-ui):

```
vendor: [charts, react, router] 250 kB
shared: [shared-ui] 40 kB
route:home: [hero] 10 kB
route:shop: [cart] 22 kB
route:admin: [admin-table] 30 kB
total shipped=352 kB (naive per-route copies would be 797 kB)
initial load home: 300 kB
```

INPUT 1

```
{"routes":{"home":
["react","router","hero","shared-
ui"],"shop":
["react","router","cart","shared-
ui","charts"],"admin":
["react","router","charts","admin-
table"]},"sizes":
{"react":130,"router":25,"hero":10,"shared-
ui":40,"cart":22,"charts":95,"admin-
table":30,"unused-legacy":50},"vendor":
["react","router","charts"]}
```

OUTPUT 1

```
vendor: [charts, react, router] 250 kB
shared: [shared-ui] 40 kB
route:home: [hero] 10 kB
route:shop: [cart] 22 kB
route:admin: [admin-table] 30 kB
total shipped=352 kB (naive per-route
copies would be 797 kB)
initial load home: 300 kB
initial load shop: 312 kB
initial load admin: 320 kB
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const plan = JSON.parse(input);       // { routes: { name: [modules] }, sizes: { module:
const vendorSet = new Set(plan.vendor ?? []);
const usage = new Map();
for (const mods of Object.values(plan.routes)) for (const m of mods) usage.set(m, (usage.
const chunks = { vendor: [], shared: [] };
for (const m of Object.keys(plan.sizes).sort()) {
  if (!usage.has(m)) continue;                                    // never imported by a r
  if (vendorSet.has(m)) chunks.vendor.push(m);
  else if (usage.get(m) ⩾ 2) chunks.shared.push(m);               // imported by two or mo
}
for (const [route, mods] of Object.entries(plan.routes)) chunks[`route:${route}`] = mods.
const size = (mods) ⇒ mods.reduce((s, m) ⇒ s + plan.sizes[m], 0);
for (const [name, mods] of Object.entries(chunks)) console.log(`${name}: [${mods.join(",
const total = Object.values(chunks).reduce((s, mods) ⇒ s + size(mods), 0);
const naive = Object.values(plan.routes).reduce((s, mods) ⇒ s + size(mods), 0);
console.log(`total shipped=${total} kB (naive per-route copies would be ${naive} kB)`);
for (const route of Object.keys(plan.routes)) console.log(`initial load ${route}: ${size(
```

QUIZ

1. Tree-shaking works well with ESM and poorly with CommonJS because…

A. CommonJS is older

B. ESM imports/exports are static and analysable; `require` is a runtime call whose usage cannot be known before execution

C. CommonJS has no exports

D. Bundlers ignore CommonJS

2. `"sideEffects": false` in a library's package.json tells bundlers…

A. The library has no bugs

B. Every module is pure, so an unused import can be dropped entirely — wrong if a module registers a polyfill or imports CSS

C. To skip minification

D. To inline the library

3. Dynamic `import()` in a route component causes the bundler to…

A. Inline the module

B. Emit a separate chunk loaded on demand — route-level code splitting

C. Fail

D. Duplicate the module

4. Setting the transpile `target` to ES5 for an audience on current browsers…

A. Is safest and free

B. Produces bigger, slower output (async/await into state machines, classes into functions) for no benefit — target the oldest runtime you actually support

C. Is required by Vite

D. Improves tree-shaking

5. Source maps…

A. Reduce bundle size

B. Map minified output positions back to source files and lines so production stack traces and breakpoints are readable

C. Are only for CSS

D. Replace tests

ANSWER KEY · BUNDLERS, TRANSPILERS AND THE BUILD — WHAT HAPPENS TO YOUR CODE BEFORE IT RUNS

1. B `import _ from "lodash"` (CJS) ships everything; `import { debounce } from "lodash-es"` ships one function.

2. B List the impure files instead when some exist.

3. B Shared modules between chunks are hoisted into a common chunk.

4. B Polyfills are separate from syntax: inject only what the targets lack.

5. B Emit always; upload to the error tracker; serving publicly is a policy choice.

LESSON 4 · 12 MIN

Linting, formatting and testing — the quality toolchain

The code you write is checked by three kinds of tool before anyone reviews it: a **linter** that finds likely bugs and enforces rules, a **formatter** that removes every argument about style, and a **test runner** that proves behaviour and keeps proving it. Together with type-checking (module 13) and a pre-commit hook plus CI, they form the pipeline every serious JavaScript project runs. This lesson covers what each does and does not do, how ESLint's flat config and Prettier fit together, the shape of a good test (and what `node:test`, Vitest and Jest share), mocking with restraint, coverage as a signal, and the hook-and-CI wiring that makes the whole thing automatic.

Linting

A linter parses code into an AST and runs **rules** over it: `no-undef` (an identifier that is never declared), `no-unused-vars`, `eqeqeq` (`==` where `===` was meant), `no-var`, `prefer-const`, `no-floating-promises` (a promise nobody awaits — TypeScript-aware), `react-hooks/rules-of-hooks`, and hundreds more. Rules catch **likely bugs** and **agreed conventions**; they do not prove correctness. Every rule can be `off`, `warn` or `error`, with options.

ESLint's **flat config** (`eslint.config.js`, ESLint 9 default) is an array of config objects:

```
import js from "@eslint/js";
import tseslint from "typescript-eslint";
export default [
  { ignores: ["dist/**"] },
  js.configs.recommended,
  ...tseslint.configs.recommended,
  { files: ["src/**/*.ts"], rules: { eqeqeq: "error", "no-console": ["warn", { all
];
```

Later objects override earlier ones for the files they match. Start from the recommended presets, add the type-aware rules where TypeScript is present, and disable rules only with a reason. `eslint --fix` applies the safe autofixes (quote style, unused imports, `let` → `const`); some fixes change semantics and are marked as suggestions instead. A rule that fires hundreds of times in a healthy codebase is the wrong rule, not a hundred bugs.

Formatting

Formatting is not linting: a **formatter** rewrites whitespace, line breaks, quotes and commas to a canonical layout and has (almost) no options by design — **Prettier** for JavaScript/TypeScript/CSS/Markdown, or **Biome** as a fast Rust alternative that lints too. The gain is not the specific style but the end of style discussions and style diffs: every file looks the same, and reviews see only meaning. Wire it to format on save and in the pre-commit hook; disable

ESLint's formatting rules (`eslint-config-prettier`) so the two do not fight. `.editorconfig` covers indentation and line endings for editors that read nothing else.

Testing

A unit test is a function that arranges inputs, acts, and asserts:

```
import { test, describe } from "node:test";           // Node 18+; Vitest/Jest hav
import assert from "node:assert/strict";
import { slugify } from "./slugify.js";

describe("slugify", () ⇒ {
  test("lower-cases and joins with dashes", () ⇒ { assert.equal(slugify("Hello, W
  test("collapses repeated separators", () ⇒ { assert.equal(slugify("a  --  b"),
  test("rejects non-strings", () ⇒ { assert.throws(() ⇒ slugify(42), TypeError);
});
```

What the runners share: `describe` / `test` (or `it`) blocks, `beforeEach` / `afterEach` hooks, an assertion library (`expect(x).toBe(y)` in Jest/Vitest; `assert.equal` / `deepEqual` in Node), async tests (return a promise or use `await`), `test.only` / `test.skip` , watch mode, and output in a standard format (TAP for Node, which `node --test` prints; `ok 1 - name` , `not ok 2 - name` with a YAML diagnostic block, a `1..N` plan). **Vitest** is the default for Vite projects (same config, ESM-native, Jest-compatible API, fast); **Jest** is the incumbent (CommonJS-first, huge ecosystem); **node:test** needs no dependency and suits libraries and scripts.

Good tests: one behaviour per test, a name that reads as a sentence, arrange/act/assert, no logic (loops/conditions) in the test body, deterministic (no real time, randomness or network), and independent (order must not matter). Test the **behaviour** through the public interface, not the implementation — tests coupled to private structure break on every refactor and catch nothing.

Mocking, sparingly

A mock replaces a dependency (the clock, a network call, a module) so the test is fast and deterministic. Prefer **dependency injection** — pass `now` , `fetch` , `random` as parameters (module 11's pure core) — so most tests need no mocking framework. Where you must, use the runner's tools: `mock.method(obj, "name", impl)` in Node, `vi.fn()` / `vi.spyOn` / `vi.mock` in Vitest, `jest.fn()` / `jest.mock` in Jest; fake timers (`vi.useFakeTimers()` , `mock.timers` in Node 20) for debounce/retry code. Over-mocked tests pass while the real system fails: mock at the boundary (I/O), not between your own modules, and keep a few **integration tests** that use real pieces.

Kinds of tests and where they run

Unit (one module, milliseconds, thousands of them), integration (several modules or a real database/HTTP layer — this repository's `billing.test.ts` runs the money path on an in-memory

fake), end-to-end (a browser driving the deployed app — Playwright, as in this repository's `e2e/`; slow, few, catch what unit tests cannot). The pyramid: many unit, some integration, few e2e. **Coverage** (`c8` / `--experimental-test-coverage`, Vitest `--coverage`) shows untested lines — useful for finding gaps, harmful as a target (100% coverage of code with no assertions proves nothing).

Hooks and CI

Pre-commit (husky + lint-staged): run the formatter and `eslint --fix` on staged files, and a type-check or fast tests if they finish in seconds — the developer sees problems before the commit exists (this repository's frontend hook runs `eslint --fix` on staged `src/**/*.{ts,tsx}` and must reach zero errors; the backend's runs `tsc`). **CI** (GitHub Actions or similar) runs the full set on every push/PR: install with `npm ci`, lint, `tsc --noEmit`, unit tests, build, e2e on the built app. CI is the gate; hooks are the convenience. Never bypass the hook (`--no-verify`) — fix the finding.

Common mistakes

- Turning off rules to make the linter quiet instead of understanding them; ESLint formatting rules fighting Prettier.
- Tests with logic, shared mutable state, real timers/network, or names like `test 1`.
- Mocking your own modules until the tests test the mocks.
- Coverage as a goal; snapshot tests of large output that everyone approves blindly.
- Running only a transpiler in CI without types, lint and tests.
- `--no-verify` to skip a failing hook.

Interview angle

- *"Linter versus formatter?"* Linter finds likely bugs and enforces rules via the AST; formatter canonicalises layout with no semantic checks; run both, and stop ESLint from formatting.
- *"What makes a good unit test?"* One behaviour, arrange/act/assert, deterministic, independent, tests behaviour through the public interface, readable name.
- *"When do you mock?"* At I/O boundaries and for time/randomness — preferably by injecting dependencies; not between your own modules.
- *"What is the testing pyramid?"* Many fast unit tests, fewer integration tests, few end-to-end tests.
- *"What should CI run?"* `npm ci`, lint, type-check, tests, build, then e2e against the build.

Key takeaways

- ESLint (flat config, recommended presets, type-aware rules) for likely bugs and conventions; Prettier/Biome for layout; `eslint-config-prettier` so they do not overlap.

Tests: `describe` / `test`, arrange/act/assert, deterministic and independent; `node:test`, Vitest or Jest share the shape; TAP is the common output.

Inject dependencies instead of mocking; mock only at boundaries; keep some integration tests.

Pyramid: unit ≫ integration > e2e; coverage is a map, not a score.

Pre-commit for fast checks on staged files, CI for the full gate; never skip the hook.

EXERCISE 1

A miniature linter with --fix

Line 1 is the mode (`check` or `fix`); the rest is code. Implement six rules as regex checks per line: `no-var` (`var` → fixable to `let`), `eqeqeq` (`==` / `!=` that are not part of `===` / `!==` → fixable by adding `=`), `no-console` (any `console.x` except `console.error`), `no-debugger` (fixable by removing the statement), `max-len` (over 80 characters), `no-trailing-spaces` (fixable). In `check` mode print `<line>:<col> <rule> <message>` per finding (with `(fixable)` where a fix exists) then `<n> problem(s), <m> fixable with --fix`; in `fix` mode apply every fix, print the fixed code, then `--- <n> problem(s) remain: <rules>` (or `--- 0 problem(s) remain`).

Example (`check`):

```
1:1 no-var Unexpected var, use let or const instead (fixable)
1:15 no-trailing-spaces Trailing spaces not allowed (fixable)
2:7 eqeqeq Expected === or !== instead of == or != (fixable)
2:13 no-console Unexpected console statement
4:1 no-debugger Unexpected debugger statement (fixable)
5:1 max-len Line exceeds the maximum length of 80
7 problem(s), 5 fixable with --fix
```

(This linter matches inside strings too — a real one works on the syntax tree, which is why ESLint parses.)

INPUT 1

```
check
var total = 0;
if (a == b) console.log(total);
const ok = x === y && p != q;
debugger;
const reallyLongVariableName = someFunction(argumentNumberOne, argumentNumberTwo, three);
console.error("fine");
```

OUTPUT 1

```
1:1 no-var Unexpected var, use let or const instead (fixable)
1:15 no-trailing-spaces Trailing spaces not allowed (fixable)
2:7 eqeqeq Expected === or !== instead of == or != (fixable)
2:13 no-console Unexpected console statement
3:25 eqeqeq Expected === or !== instead of == or != (fixable)
4:1 no-debugger Unexpected debugger statement (fixable)
5:1 max-len Line exceeds the maximum length of 80
7 problem(s), 5 fixable with --fix
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const [mode, ...code] = input.split("\n");
const rules = [
  { name: "no-var", re: /\bvar\b/, message: "Unexpected var, use let or const instead", f
  { name: "eqeqeq", re: /(?<![=!<>])(==|!=)(?!=)/, message: "Expected === or !== instead
  { name: "no-console", re: /\bconsole\.(?!error\b)\w+/, message: "Unexpected console sta
  { name: "no-debugger", re: /\bdebugger\b/, message: "Unexpected debugger statement", fi
  { name: "max-len", re: /^.{81,}$/, message: "Line exceeds the maximum length of 80" },
  { name: "no-trailing-spaces", re: /[ \t]+$/, message: "Trailing spaces not allowed", fi
];
function lint(lines) {
  const findings = [];
  lines.forEach((line, i) => {
    for (const rule of rules) {
      const m = rule.re.exec(line);
      if (m) findings.push({ line: i + 1, col: m.index + 1, rule: rule.name, message: rul
    }
  });
  return findings;
}
const trimmed = code.filter((_, i) => i < code.length - 1 || code[i] !== "");   // drop t
if (mode.trim() === "fix") {
  const fixed = trimmed.map((line) => rules.reduce((l, r) => (r.fix && r.re.test(l) ? r.f
  console.log(fixed.join("\n"));
  const remaining = lint(fixed);
  console.log(`--- ${remaining.length} problem(s) remain${remaining.length ? `: ${remaini
} else {
  const findings = lint(trimmed);
  for (const f of findings) console.log(`${f.line}:${f.col} ${f.rule} ${f.message}${f.fix
  console.log(`${findings.length} problem(s), ${findings.filter((f) => f.fixable).length}
}
```

EXERCISE 2

A test runner that speaks TAP

The starter defines `sum`, `slug` and `parsePort` plus a `broken` set (from line 1, comma-separated) that sabotages the named functions. Write `describe` / `test` collectors and six tests (two per function, using `node:assert/strict`: `sum(2,3)=5`, commutativity, slugify `"Hello, World!"` → `hello-world`, collapsing separators, `parsePort("8080")=8080`, `parsePort("70000")` throws a `RangeError`). Run them and print TAP: `TAP version 13`, then per test `ok <n> - <describe> > <name>` or `not ok <n> - …` followed by a diagnostic block `---`, `error: <name>`, `actual: <JSON>` / `expected: <JSON, or a function's name>` when present, `...`; then `1..<n>`, `# tests <n>`, `# pass <n>`, `# fail <n>`.

Example with `sum,parse` broken (excerpt):

```
TAP version 13
not ok 1 - sum > adds two numbers
  ---
  error: AssertionError
  actual: -1
  expected: 5
  ...
ok 3 - slug > lower-cases and joins with dashes
not ok 6 - parsePort > rejects 70000
  ---
  error: AssertionError
  expected: RangeError
  ...
1..6
# tests 6
# pass 3
# fail 3
```

INPUT 1

OUTPUT 1

```
TAP version 13
ok 1 - sum > adds two numbers
ok 2 - sum > is commutative
ok 3 - slug > lower-cases and joins with
dashes
ok 4 - slug > collapses separators
ok 5 - parsePort > accepts 8080
ok 6 - parsePort > rejects 70000
1..6
# tests 6
# pass 6
# fail 0
```

INPUT 2

```
sum,parse
```

OUTPUT 2

```
TAP version 13
not ok 1 - sum > adds two numbers
  ---
  error: AssertionError
  actual: -1
  expected: 5
  ...
not ok 2 - sum > is commutative
  ---
  error: AssertionError
  actual: -8
  expected: 8
  ...
ok 3 - slug > lower-cases and joins with
dashes
ok 4 - slug > collapses separators
ok 5 - parsePort > accepts 8080
not ok 6 - parsePort > rejects 70000
  ---
  error: AssertionError
  expected: RangeError
  ...
1..6
# tests 6
# pass 3
# fail 3
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const assert = require("node:assert/strict");
const broken = new Set((lines[0] ?? "").trim().split(/[\s,]+/).filter(Boolean));
// The code under test; `broken` sabotages named functions so some tests fail.
const sum = (a, b) => (broken.has("sum") ? a - b : a + b);
const slug = (s) => (broken.has("slug") ? s.toLowerCase() : s.trim().toLowerCase().replac
const parsePort = (t) => { const n = Number(t); if (!Number.isInteger(n) || n < 1 || n >
const cases = [];
const stack = [];
const describe = (name, fn) => { stack.push(name); fn(); stack.pop(); };
const test = (name, fn) => cases.push({ name: [...stack, name].join(" > "), fn });
describe("sum", () => {
  test("adds two numbers", () => assert.equal(sum(2, 3), 5));
  test("is commutative", () => assert.equal(sum(1, 9), sum(9, 1)));
});
describe("slug", () => {
  test("lower-cases and joins with dashes", () => assert.equal(slug("Hello, World!"), "he
  test("collapses separators", () => assert.equal(slug("a  --  b"), "a-b"));
});
describe("parsePort", () => {
  test("accepts 8080", () => assert.equal(parsePort("8080"), 8080));
  test("rejects 70000", () => assert.throws(() => parsePort("70000"), RangeError));
});
console.log("TAP version 13");
let pass = 0, fail = 0;
cases.forEach((c, i) => {
  try {
    c.fn();
    pass++;
    console.log(`ok ${i + 1} - ${c.name}`);
  } catch (err) {
    fail++;
    console.log(`not ok ${i + 1} - ${c.name}`);
    console.log("  ---");
    console.log(`  error: ${err.name}${err.code === "ERR_ASSERTION" ? "" : `: ${err.messa
    const show = (v) => (typeof v === "function" ? v.name : JSON.stringify(v));
    if ("actual" in err && err.actual !== undefined) console.log(`  actual: ${show(err.ac
    if ("expected" in err && err.expected !== undefined) console.log(`  expected: ${show(
    console.log("  ...");
  }
});
console.log(`1..${cases.length}`);
console.log(`# tests ${cases.length}`);
console.log(`# pass ${pass}`);
console.log(`# fail ${fail}`);
```

QUIZ

1. Linter versus formatter:

A. Synonyms

B. A linter finds likely bugs and enforces rules on the syntax tree; a formatter canonicalises layout only — run both and disable ESLint's formatting rules

C. A formatter finds bugs

D. Only one is needed

2. A good unit test…

A. Covers many behaviours at once

B. Checks one behaviour through the public interface with arrange/act/assert, is deterministic and independent, and has a name that reads as a sentence

C. Mocks every dependency

D. Uses real time and network

3. Prefer dependency injection to mocking frameworks because…

A. Mocking is deprecated

B. Passing `now`, `fetch` or `random` as parameters makes most tests need no mocks at all — mock only at I/O boundaries

C. It is faster

D. Frameworks cannot mock time

4. TAP output `not ok 3 - name` followed by an indented `---` / `...` block is…

A. An error in the runner

B. A failed test with its YAML diagnostic (error, actual, expected) — the common format `node --test` prints

C. A skipped test

D. A comment

5. The pre-commit hook fails on a lint error. You should…

A. Commit with `--no-verify`

B. Fix the finding — the hook is the convenience gate, CI is the real one, and bypassing either ships the problem

C. Disable the rule

D. Delete the hook

ANSWER KEY · LINTING, FORMATTING AND TESTING — THE QUALITY TOOLCHAIN

1. B `eslint-config-prettier` stops the two from fighting.

2. B Tests coupled to private structure break on every refactor and catch nothing.

3. B Over-mocked tests pass while the real system fails.

4. B `1..N` is the plan; `# pass` / `# fail` the summary.

5. B This repository's hooks must reach zero errors.

LESSON 5 · 12 MIN

Runtimes and the ecosystem — Node, Deno, Bun, browsers, edge, and the web-standard core

JavaScript runs in more places than any other language: four browser engines, Node, Deno, Bun, a dozen edge/serverless platforms, and inside other programs. They share the language and increasingly share the **web-standard APIs** — `fetch`, `URL`, `Request` / `Response`, Web Streams, `crypto.subtle`, `TextEncoder` — but differ in globals, module defaults, permissions and what they consider core. This lesson maps the runtimes, the common core that makes code portable, the differences that bite, how to write for several targets at once, the package ecosystem's shape and hazards, and where WebAssembly fits.

The runtimes

Browsers — V8 (Chrome/Edge), SpiderMonkey (Firefox), JavaScriptCore (Safari). Sandboxed, DOM, no file system, ESM natively, permission prompts for capabilities; the only place your code runs on hardware you do not control.

Node — V8 + libuv; the incumbent server runtime and the toolchain host; CommonJS heritage with ESM support; npm; the largest ecosystem; LTS every even major (16 → 18 → 20 → 22 → 24).

Deno — V8, TypeScript first-class, ESM only, URL and npm imports, secure by default (`--allow-net`, `--allow-read` flags), web-standard APIs as the core (`fetch`, `Deno.serve` returning `Response`), built-in formatter/linter/tester.

Bun — JavaScriptCore, written in Zig; a drop-in Node-compatible runtime with a very fast package manager, bundler and test runner built in; TypeScript/JSX without a build step; `Bun.serve`, SQLite built in.

Edge / **serverless** — Cloudflare Workers (V8 isolates, no Node APIs, request/response model), Vercel Edge, Deno Deploy, Fastly; millisecond cold starts, tight CPU/memory limits, web-standard APIs only.

Embedded and others — Electron/Tauri (desktop with Node/browser combined), React Native/Hermes (mobile), QuickJS (tiny embeddable), Node in AWS Lambda (serverless with Node APIs).

The common core: web-standard APIs

The **WinterCG** (Web-interoperable Runtimes Community Group, now WinterTC) standardised a minimum common API so server code can be portable: `fetch`, `Request`, `Response`, `Headers`, `URL`, `URLSearchParams`, `AbortController` / `AbortSignal`, `TextEncoder` / `TextDecoder`, `ReadableStream` / `WritableStream` / `TransformStream`, `crypto.subtle` / `crypto.randomUUID`, `structuredClone`, `queueMicrotask`, `setTimeout`, `atob` / `btoa`, `performance`, `EventTarget` / `Event`, `Blob` / `File`, `FormData`, `WebSocket` (client), `console`. Node reached most of this by version 18–20 (global `fetch` in 18, `WebSocket` in 22), Deno and Bun from the start. Write

against these and your handler runs on Node, Deno, Bun and a Worker with no changes — the frameworks that do this (Hono, Remix's adapters, SvelteKit adapters) are the portable ones.

Where they differ

Globals: `window` / `document` only in browsers; `process`, `Buffer`, `__dirname`, `require` in Node (Bun provides them; Deno via `node:` compatibility; Workers do not).

Modules: Deno and Workers are ESM-only; Node and Bun accept both; browsers need `type="module"` and full URLs or import maps.

File and network access: browsers have none/limited; Node/Bun unrestricted; Deno permission-gated; Workers have no file system and fetch-only networking (plus their own storage APIs).

Timers and event loop details: `process.nextTick` / `setImmediate` are Node-specific; browsers have `requestAnimationFrame`.

Node compatibility: Bun aims for full; Deno supports `node:` builtins and npm packages; Workers offer a `nodejs_compat` subset. Test on the real target — compatibility layers have gaps (native addons, `fs` semantics, `worker_threads`).

Package management: npm/pnpm/yarn for Node; Bun's `bun install` (npm-compatible, fast); Deno's URL imports, JSR registry and npm specifiers.

Writing for several targets

Detect capabilities, not runtimes: `typeof fetch === "function"`, `"Deno" in globalThis`, `typeof process !== "undefined" && process.versions?.node`. Isolate platform code behind small adapters (a `readConfig()` that uses `fs` on Node and `Deno.readTextFile` on Deno, or `KV` on Workers) and keep the core pure (module 11). Use package `exports` **conditions** — `"node"`, `"deno"`, `"browser"`, `"workerd"`, `"default"` — to ship different entry files per runtime. Avoid Node-only globals in shared code; prefer `globalThis` and the WinterCG set. TypeScript's `lib` and `@types/node` / `@cloudflare/workers-types` keep each target honest.

The ecosystem's shape and hazards

npm is the largest package registry in existence — and the most attacked. Practical rules from module 9 apply everywhere: lockfiles and `npm ci`, few dependencies, scoped names, audit with judgement. Beyond security: **dependency churn** (a framework's major version every year), **duplicate installs** (two Reacts break hooks — `npm ls react`), **CJS/ESM dual-package hazards**, **abandoned packages** (check last publish, open issues, bus factor), and **transitive weight** (one convenience package pulling 300 others). Prefer platform APIs where they exist (`fetch` over `axios`, `URL` over query-string libraries, `crypto.randomUUID` over `uuid`, `Intl` over `moment`) — fewer dependencies and portable by construction. Standard-library gaps (a real date type, a schema validator, a robust CLI parser) are where a dependency earns its place.

WebAssembly

Wasm is a portable binary format every runtime executes: compile Rust, C, Go or AssemblyScript to `.wasm`, load it (`WebAssembly.instantiate`), and call exported functions from JavaScript with near-native speed for CPU-heavy work — image codecs, compression, crypto, physics, SQLite in the browser, ffmpeg.wasm. It is not a JavaScript replacement (no DOM access; JS glue calls it) and not automatically faster for ordinary code (crossing the boundary costs; V8's JIT is excellent at JavaScript). Reach for it when a library already exists in Wasm or a hot numeric kernel dominates a profile.

Staying current without churning

Follow Node's release schedule and upgrade LTS majors yearly; read the yearly ECMAScript summary and the State of JS survey once; watch TC39 stage 3; try a new tool in a side project before a codebase; prefer boring, standard, portable choices in production (`fetch`, ESM, web APIs, TypeScript) — they age best. The knowledge in this plan (the language, the loop, the module system, the runtime model) outlasts every framework cycle.

Common mistakes

- Node-only globals (`Buffer`, `process`, `__dirname`) in code meant for browsers or edge.
- Assuming a compatibility layer equals Node (native addons, `fs` edge cases).
- Sniffing user agents or runtime names instead of features.
- A dependency for something the platform provides; two copies of a singleton-style package.
- Choosing a runtime or framework by novelty rather than by the deployment target and team.

Interview angle

- *"Node versus Deno versus Bun?"* V8 + npm incumbent with CJS heritage; V8 + TypeScript-first, secure-by-default, web-standard core; JSC-based Node-compatible all-in-one with a fast package manager and test runner.
- *"What are the WinterCG APIs?"* The web-standard core (`fetch`, `URL`, Web Streams, `crypto.subtle`, `structuredClone`…) shared by servers and edge runtimes for portability.
- *"How do edge runtimes differ from Node?"* V8 isolates with request/response handlers, millisecond cold starts, no file system, web APIs only, tight limits.
- *"When would you use WebAssembly?"* CPU-heavy kernels or existing native libraries (codecs, compression, SQLite) — not for ordinary application code.
- *"How do you write code that runs on several runtimes?"* Web-standard APIs, feature detection, adapters for platform I/O, `exports` conditions per runtime.

Key takeaways

- Browsers, Node (LTS even majors), Deno (TS-first, permissions, ESM), Bun (Node-compatible, fast tooling), edge isolates (web APIs, no Node) — same language, different globals and capabilities.
- The WinterCG core (`fetch`, `URL`, streams, `crypto.subtle`, `structuredClone`) is the portable surface; write to it.
- Detect features, isolate platform I/O, use `exports` conditions; test on the real target.
- Fewer dependencies, platform APIs first, lockfiles, watch for duplicates and abandonment.
- Wasm for hot kernels and existing native libraries; JavaScript for everything else.

EXERCISE 1

Detect features, do not sniff versions

Build a table of feature checks: **syntax** features detected by compiling a snippet with `new Function(source)` in `try` / `catch` (`optional-chaining`, `nullish-assignment`, `class-static-blocks`, `private-in`, `numeric-separators`, `using-declarations`, `regex-v-flag` via `new RegExp("[a]", "v")`, and `top-level-await` which cannot be detected from CommonJS — report it missing), and **built-ins** detected with `typeof` (`Array.prototype.at`, `Object.hasOwn`, `String.prototype.replaceAll`, `Promise.any`, `AggregateError`, `WeakRef`, `Intl.Segmenter`, `structuredClone`, `fetch`, `Array.prototype.findLast`, `Array.prototype.toSorted`, `Object.groupBy`, `Promise.withResolvers`, `AbortSignal.timeout`). For each requested name print `<name>: native`, `<name>: missing → transpile` (syntax) or `<name>: missing → polyfill` (built-in), or `<name>: unknown feature`. The judge runs Node 16.17, so the expected output is that runtime's honest answer.

Example (excerpt, on Node 16):

```
optional-chaining: native
using-declarations: missing → transpile
Array.prototype.at: native
structuredClone: missing → polyfill
fetch: missing → polyfill
```

INPUT 1

```
optional-chaining
nullish-assignment
class-static-blocks
private-in
numeric-separators
using-declarations
regex-v-flag
Array.prototype.at
Object.hasOwn
String.prototype.replaceAll
Promise.any
AggregateError
WeakRef
Intl.Segmenter
structuredClone
fetch
Array.prototype.findLast
Array.prototype.toSorted
Object.groupBy
Promise.withResolvers
AbortSignal.timeout
top-level-await
teleportation
```

OUTPUT 1

```
optional-chaining: native
nullish-assignment: native
class-static-blocks: native
private-in: native
numeric-separators: native
using-declarations: missing → transpile
regex-v-flag: missing → transpile
Array.prototype.at: native
Object.hasOwn: native
String.prototype.replaceAll: native
Promise.any: native
AggregateError: native
WeakRef: native
Intl.Segmenter: native
structuredClone: missing → polyfill
fetch: missing → polyfill
Array.prototype.findLast: missing →
polyfill
Array.prototype.toSorted: missing →
polyfill
Object.groupBy: missing → polyfill
Promise.withResolvers: missing → polyfill
AbortSignal.timeout: native
top-level-await: missing → transpile
teleportation: unknown feature
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const syntax = (src) => () => { try { new Function(src); return true; } catch { return fa
const checks = {
  "optional-chaining": { kind: "syntax", present: syntax("return a?.b") },
  "nullish-assignment": { kind: "syntax", present: syntax("let a; a ??= 1") },
  "class-static-blocks": { kind: "syntax", present: syntax("class A { static { } }") },
  "private-in": { kind: "syntax", present: syntax("class A { #x; static has(o) { return #
  "numeric-separators": { kind: "syntax", present: syntax("return 1_000") },
  "using-declarations": { kind: "syntax", present: syntax("using x = null") },
  "regex-v-flag": { kind: "syntax", present: () => { try { new RegExp("[a]", "v"); returr
  "top-level-await": { kind: "syntax", present: () => false },       // cannot be detecte
  "Array.prototype.at": { kind: "builtin", present: () => typeof Array.prototype.at === '
  "Object.hasOwn": { kind: "builtin", present: () => typeof Object.hasOwn === "function"
  "String.prototype.replaceAll": { kind: "builtin", present: () => typeof String.prototyp
  "Promise.any": { kind: "builtin", present: () => typeof Promise.any === "function" },
  "AggregateError": { kind: "builtin", present: () => typeof AggregateError === "functior
  "WeakRef": { kind: "builtin", present: () => typeof WeakRef === "function" },
  "Intl.Segmenter": { kind: "builtin", present: () => typeof Intl.Segmenter === "functior
  "structuredClone": { kind: "builtin", present: () => typeof globalThis.structuredClone
  "fetch": { kind: "builtin", present: () => typeof globalThis.fetch === "function" },
  "Array.prototype.findLast": { kind: "builtin", present: () => typeof Array.prototype.fi
  "Array.prototype.toSorted": { kind: "builtin", present: () => typeof Array.prototype.tc
  "Object.groupBy": { kind: "builtin", present: () => typeof Object.groupBy === "functior
  "Promise.withResolvers": { kind: "builtin", present: () => typeof Promise.withResolvers
  "AbortSignal.timeout": { kind: "builtin", present: () => typeof AbortSignal.timeout ===
};
for (const name of lines.map((l) => l.trim())) {
  const check = checks[name];
  if (!check) { console.log(`${name}: unknown feature`); continue; }
  console.log(`${name}: ${check.present() ? "native" : `missing → ${check.kind === "synt
}
```

EXERCISE 2

Resolve bare specifiers with an import map

Line 1 is an import map `{ imports, scopes }`; each following line is `<specifier> <referrer URL>`. Resolve like a browser: relative (`./`, `../`, `/`) and absolute specifiers resolve against the referrer with `new URL`; a bare specifier is looked up first in every **scope** whose key is a prefix of the referrer's path (most specific first, reported as `<url> (scope <key>)`), then in `imports` — an exact key wins, otherwise the longest key ending in `/` that prefixes the specifier maps the remainder; nothing found → `TypeError: Failed to resolve module specifier "<s>": bare specifiers need an import map entry`. Print `<specifier> from <referrer> → <result>`.

Example (excerpt):

```
lodash/debounce from https://site.example/src/main.js → https://cdn.example/lodash-es/4.
app/util/slug.js from https://site.example/src/main.js → /src/util/slug.js
../shared/x.js from https://site.example/src/pages/home.js → https://site.example/src/sh
react from https://site.example/src/main.js → TypeError: Failed to resolve module specif
```

INPUT 1

```
{"imports":{"lodash":"https://cdn.example/lodash-
es/4.17.21/lodash.js","lodash/":"https://cdn.example/lodash-
es/4.17.21/","app/":"/src/"},"scopes":{"/vendor/legacy/":
{"lodash":"https://cdn.example/lodash/3.10.1/lodash.js"}}}
lodash https://site.example/src/main.js
lodash/debounce https://site.example/src/main.js
app/util/slug.js https://site.example/src/main.js
lodash https://site.example/vendor/legacy/old.js
./helpers.js https://site.example/src/pages/home.js
../shared/x.js https://site.example/src/pages/home.js
react https://site.example/src/main.js
https://other.example/lib.js
https://site.example/src/main.js
```

OUTPUT 1

```
lodash from
https://site.example/src/main.js →
https://cdn.example/lodash-
es/4.17.21/lodash.js
lodash/debounce from
https://site.example/src/main.js →
https://cdn.example/lodash-
es/4.17.21/debounce
app/util/slug.js from
https://site.example/src/main.js →
/src/util/slug.js
lodash from
https://site.example/vendor/legacy/o
→ https://cdn.example/lodash-
es/4.17.21/lodash.js
./helpers.js from
https://site.example/src/pages/home.
https://site.example/src/pages/helpe
../shared/x.js from
https://site.example/src/pages/home.
https://site.example/src/shared/x.js
react from
https://site.example/src/main.js →
TypeError: Failed to resolve module
specifier "react": bare specifiers n
an import map entry
https://other.example/lib.js from
https://site.example/src/main.js →
https://other.example/lib.js
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const [mapText, ...queries] = input.split("\n").filter((l) => l.trim() !== "");
const importMap = JSON.parse(mapText);      // { imports: { bare: url, "prefix/": url },
function lookup(table, specifier) {
  if (Object.hasOwn(table, specifier)) return table[specifier];
  const prefixes = Object.keys(table).filter((k) => k.endsWith("/") && specifier.startsWi
  return prefixes.length ? table[prefixes[0]] + specifier.slice(prefixes[0].length) : nul
}
function resolve(specifier, referrer) {
  const isRelative = specifier.startsWith("./") || specifier.startsWith("../") || specifi
  if (!isRelative) {
    const scopes = Object.keys(importMap.scopes ?? {}).filter((s) => referrer.startsWith(
    for (const s of scopes) { const hit = lookup(importMap.scopes[s], specifier); if (hit
    const hit = lookup(importMap.imports ?? {}, specifier);
    if (hit) return hit;
    throw new TypeError(`Failed to resolve module specifier "${specifier}": bare specifie
  }
  return new URL(specifier, referrer).href;
}
for (const q of queries) {
  const [specifier, referrer] = q.trim().split(/\s+/);
  try { console.log(`${specifier} from ${referrer} → ${resolve(specifier, referrer)}`);
  catch (err) { console.log(`${specifier} from ${referrer} → ${err.name}: ${err.message}
}
```

QUIZ

1. The WinterCG common API set exists so that…

A. Browsers can run Node code

B. Server and edge runtimes share the web-standard APIs (`fetch` , `URL` , streams, `crypto.subtle` , `structuredClone`) and code written to them is portable

C. npm works in Deno

D. TypeScript compiles faster

2. A Cloudflare Worker differs from a Node server in that it…

A. Has a bigger standard library

B. Runs in V8 isolates with request/response handlers, millisecond cold starts, no file system and web APIs only (plus a Node-compat subset)

C. Requires CommonJS

D. Cannot use `fetch`

3. Deno's approach to permissions is…

A. Same as Node

B. Secure by default — file, network and environment access need explicit `--allow-*` flags

C. Prompting the user each time

D. Sandboxed per module

4. Feature detection over version sniffing means…

A. Checking `process.version`

B. Testing for the capability itself (`typeof fetch === "function"` , `new Function(src)` for syntax) so polyfills and compatibility layers are honoured

C. Reading the user agent

D. Trusting `engines`

5. WebAssembly is the right tool for…

A. Every hot code path

B. CPU-heavy kernels and existing native libraries (codecs, compression, SQLite) — not ordinary application code, where the boundary cost outweighs gains

C. DOM manipulation

D. Replacing JavaScript

ANSWER KEY · RUNTIMES AND THE ECOSYSTEM — NODE, DENO, BUN, BROWSERS, EDGE, AND THE WEB-STANDARD CORE

1. B Frameworks that target this set run on Node, Deno, Bun and Workers unchanged.

2. B Edge code that touches `fs` or `process` fails.

3. B Bun prioritises Node compatibility and speed instead.

4. B A polyfilled runtime passes a capability check and fails a version check.

5. B JavaScript's JIT is excellent at JavaScript.

CHECKPOINT · 24 MIN

Checkpoint — Modern JavaScript and its tooling

This checkpoint covers the ES2020–2022 features and their pitfalls, the 2023+ additions and the TC39 stage process, how to detect features and write spec-faithful polyfills, the build pipeline (resolve, transpile, tree-shake, split, minify, source maps), the quality toolchain (ESLint flat config, Prettier, test runners, mocking, hooks and CI), and the runtime landscape with its web-standard core.

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:

- `??` versus `||`; how `a?.b.c` short-circuits; why BigInt refuses to mix; which 2022 class features exist.
- What `toSorted` / `with` / `Object.groupBy` / `Promise.withResolvers` do; the TC39 stages and when a proposal is safe to adopt; why syntax cannot be polyfilled.
- The build stages; what defeats tree-shaking; how code splitting and hashed chunks work; what source maps are for.
- Linter versus formatter; the shape of a good test; when to mock; what pre-commit and CI each run.
- Node/Deno/Bun/edge differences; the WinterCG common APIs; feature detection over sniffing.

The programs are a spec-faithful `structuredClone` polyfill (cycles, Dates, Maps, Sets, regexes, and a `DataCloneError` for functions), a miniature bundler that resolves an ESM graph, tree-shakes unused exports and emits one file, and a runtime-support planner that turns feature requirements and a target list into the polyfills and transpile target a build needs.

EXERCISE 1

A structuredClone polyfill

Node 16 lacks `structuredClone`. Write `structuredClonePolyfill(value)` following the structured-clone algorithm: primitives returned as-is; functions and symbols throw an error named `DataCloneError`; a `Map` from originals to clones preserves **cycles and shared references**; `Date`, `RegExp` (source + flags), `Map`, `Set`, arrays and plain objects are cloned deeply; class instances become plain objects (prototype and methods lost). The starter builds an object with a cycle, a shared reference, a Date, a regex, a Map, a Set and a class instance, plus the JSON from line 1; print the checks: `distinct=<> dataEqual=<> dataDistinct=<>`, `cycle=<clone.self === clone> sharedRefPreserved=<>`, `date=<> regex=<>`, `map=<> set=<>`, `pointIsPlain=<> x=<> distMethod=<typeof>`, then `function: DataCloneError` and `symbol: DataCloneError`.

Example →

```
distinct=true dataEqual=true dataDistinct=true
cycle=true sharedRefPreserved=true
date=true regex=true
map=true set=true
pointIsPlain=true x=3 distMethod=undefined
function: DataCloneError
symbol: DataCloneError
```

INPUT 1

```
{"n":1,"list":[1,2,{"deep":true}]}
```

OUTPUT 1

```
distinct=true dataEqual=true
dataDistinct=true
cycle=true sharedRefPreserved=true
date=true regex=true
map=true set=true
pointIsPlain=true x=3 distMethod=undefined
function: DataCloneError
symbol: DataCloneError
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
class DataCloneError extends Error { constructor(msg) { super(msg); this.name = "DataClor
function structuredClonePolyfill(value, seen = new Map()) {
  if (value === null || typeof value !== "object") {
    if (typeof value === "function" || typeof value === "symbol") throw new DataCloneErro
    return value;
  }
  if (seen.has(value)) return seen.get(value);                // a cycle or a shared ref
  let out;
  if (value instanceof Date) out = new Date(value.getTime());
  else if (value instanceof RegExp) out = new RegExp(value.source, value.flags);
  else if (value instanceof Map) { out = new Map(); seen.set(value, out); for (const [k,
  else if (value instanceof Set) { out = new Set(); seen.set(value, out); for (const v of
  else if (Array.isArray(value)) { out = []; seen.set(value, out); for (const v of value)
  else {
    out = {};                                                 // class instances lose
    seen.set(value, out);
    for (const k of Object.keys(value)) out[k] = structuredClonePolyfill(value[k], seen);
    return out;
  }
  seen.set(value, out);
  return out;
}
const data = JSON.parse(lines[0]);
class Point { constructor(x) { this.x = x; } dist() { return this.x; } }
const original = { data, when: new Date(0), re: /a+/gi, map: new Map([["k", { n: 1 }]]),
original.self = original;                                     // a cycle
original.twice = [original.data, original.data];              // a shared reference
const clone = structuredClonePolyfill(original);
console.log(`distinct=${clone !== original} dataEqual=${JSON.stringify(clone.data) === JS
console.log(`cycle=${clone.self === clone} sharedRefPreserved=${clone.twice[0] === clone.
console.log(`date=${clone.when instanceof Date && clone.when.getTime() === 0} regex=${clo
console.log(`map=${clone.map instanceof Map && clone.map.get("k").n === 1 && clone.map.ge
console.log(`pointIsPlain=${Object.getPrototypeOf(clone.point) === Object.prototype} x=${
for (const [label, bad] of [["function", { f: () => 1 }], ["symbol", { s: Symbol("x") }]]
  try { structuredClonePolyfill(bad); console.log(`${label}: cloned?!`); }
  catch (err) { console.log(`${label}: ${err.name}`); }
}
```

EXERCISE 2

A miniature bundler

Line 1 is `entry: <file>`; the rest are modules as blocks starting `// file: <name>`, containing one-line statements: `import { a, b } from "./x.js";`, `export const|let|function|class <name> …`, and ordinary lines. Build the dependency graph from the entry (post-order so dependencies come first; a circular import prints `warning: circular import <from> → <to>` and is not followed), **tree-shake** exports that no module imports (the entry keeps all of its own), and emit the bundle: for each module `// ---- <name>` followed by its lines with the `import` statements removed and the `export` keyword stripped. End with `// dropped: <module:export, …> | order: <a → b → c> | unused modules: <names or →`. Names are unique across modules.

Example (excerpt): an entry importing `slugify` and `log`, a utils module also exporting unused `debounce` / `throttle`, a logger with unused `warn`, and an unreferenced module →

```
// ---- logger.js
const log = (m) ⇒ console.log(m);
// ---- utils.js
const slugify = (s) ⇒ s.toLowerCase().replace(/\s+/g, "-");
// ---- main.js
const title = slugify("Hello World");
log(title);
// dropped: logger.js:warn, utils.js:debounce, utils.js:throttle | order: logger.js → ut
```

INPUT 1

```
entry: main.js
// file: main.js
import { slugify } from "./utils.js";
import { log } from "./logger.js";
export const title = slugify("Hello
World");
log(title);

// file: utils.js
import { log } from "./logger.js";
export const slugify = (s) ⇒
s.toLowerCase().replace(/\s+/g, "-");
export const debounce = (fn) ⇒ fn;
export function throttle(fn) { return fn;
}

// file: logger.js
export const log = (m) ⇒ console.log(m);
export const warn = (m) ⇒
console.error(m);

// file: unused.js
export const nothing = 1;
```

OUTPUT 1

```
// ---- logger.js
const log = (m) ⇒ console.log(m);
// ---- utils.js
const slugify = (s) ⇒
s.toLowerCase().replace(/\s+/g, "-");
// ---- main.js
const title = slugify("Hello World");
log(title);
// dropped: logger.js:warn,
utils.js:debounce, utils.js:throttle |
order: logger.js → utils.js → main.js |
unused modules: unused.js
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
// Modules are text blocks starting `// file: name.js`; statements: `import { a, b } from
const [entryLine, ...rest] = input.split("\n");
const entry = entryLine.replace(/^entry:\s*/, "").trim();
const modules = new Map();
let current = null;
for (const line of rest) {
  const m = /^\/\/ file: (\S+)/.exec(line);
  if (m) { current = { name: m[1], lines: [], imports: new Map(), exports: [] }; modules.
  if (!current || line.trim() === "") continue;
  const im = /^import \{([^}]*)\} from "\.\/(.+?)";?$/.exec(line.trim());
  if (im) { current.imports.set(im[2], im[1].split(",").map((s) => s.trim()).filter(Boole
  const ex = /^export (?:const|let|function|class) (\w+)/.exec(line.trim());
  if (ex) current.exports.push(ex[1]);
  current.lines.push(line);
}
const order = [], visiting = new Set(), used = new Map();
(function visit(name, from) {
  const mod = modules.get(name);
  if (!mod) throw new Error(`cannot resolve ./${name} from ${from}`);
  if (order.includes(name)) return;
  if (visiting.has(name)) { console.log(`warning: circular import ${from} -> ${name}`); r
  visiting.add(name);
  for (const [dep, names] of mod.imports) { (used.get(dep) ?? used.set(dep, new Set()).ge
  visiting.delete(name);
  order.push(name);                                          // dependencies first: pos
})(entry, "<root>");
const out = [], dropped = [];
for (const name of order) {
  const mod = modules.get(name);
  const wanted = name === entry ? new Set(mod.exports) : (used.get(name) ?? new Set());
  out.push(`// ---- ${name}`);
  for (const line of mod.lines) {
    const ex = /^export (?:const|let|function|class) (\w+)/.exec(line.trim());
    if (ex && !wanted.has(ex[1])) { dropped.push(`${name}:${ex[1]}`); continue; }    // t
    out.push(line.replace(/^(\s*)export /, "$1"));
  }
}
console.log(out.join("\n"));
console.log(`// dropped: ${dropped.join(", ") || "-"} | order: ${order.join(" -> ")} | ur
```

EXERCISE 3

A runtime-support planner

Line 1 lists targets as `runtime version` pairs (`node 16.17 chrome 100 safari 15.4`); the following lines are features. Using the starter's table of minimum versions and kinds, decide per feature: `supported everywhere`, `polyfill (missing in <targets>)` for built-ins, or `transpile (missing in <targets>)` for syntax; unknown names print `unknown feature`. Compare versions as dotted **tuples** — `16.17` is newer than `16.6`, so never compare as floats. Finish with `transpile target: ES<year>` (the lowest level any target supports: Node 18+/Chrome 94+/Safari 15.4+ → 2022; Node 16/Chrome 85/Safari 14.1 → 2021; Node 14/Chrome 80/Safari 13.1 → 2020; else 2019), `polyfills: <list or none>`, `syntax to transpile: <list or none>`.

Example: `node 16.17 chrome 100 safari 15.4` with `optional-chaining`, `at`, `structuredClone`, `toSorted`, `using`, `fetch`, `flying-cars` →

```
optional-chaining: supported everywhere
at: supported everywhere
structuredClone: polyfill (missing in node 16.17)
toSorted: polyfill (missing in node 16.17, chrome 100, safari 15.4)
using: transpile (missing in node 16.17, chrome 100, safari 15.4)
fetch: polyfill (missing in node 16.17)
flying-cars: unknown feature
transpile target: ES2021
polyfills: structuredClone, toSorted, fetch
syntax to transpile: using
```

INPUT 1

```
node 16.17 chrome 100 safari 15.4
optional-chaining
at
structuredClone
toSorted
using
fetch
flying-cars
```

OUTPUT 1

```
optional-chaining: supported everywhere
at: supported everywhere
structuredClone: polyfill (missing in node
16.17)
toSorted: polyfill (missing in node 16.17,
chrome 100, safari 15.4)
using: transpile (missing in node 16.17,
chrome 100, safari 15.4)
fetch: polyfill (missing in node 16.17)
flying-cars: unknown feature
transpile target: ES2021
polyfills: structuredClone, toSorted,
fetch
syntax to transpile: using
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
// Minimum versions per runtime; kind decides the remedy when a target lacks a feature.
const TABLE = {
  "optional-chaining": { kind: "syntax", node: 14, chrome: 80, safari: 13.1 },
  "class-fields": { kind: "syntax", node: 12, chrome: 72, safari: 14.1 },
  "top-level-await": { kind: "syntax", node: 14.8, chrome: 89, safari: 15 },
  "using": { kind: "syntax", node: Infinity, chrome: 134, safari: Infinity },
  "replaceAll": { kind: "builtin", node: 15, chrome: 85, safari: 13.1 },
  "at": { kind: "builtin", node: 16.6, chrome: 92, safari: 15.4 },
  "structuredClone": { kind: "builtin", node: 17, chrome: 98, safari: 15.4 },
  "toSorted": { kind: "builtin", node: 20, chrome: 110, safari: 16 },
  "groupBy": { kind: "builtin", node: 21, chrome: 117, safari: 17.4 },
  "fetch": { kind: "builtin", node: 18, chrome: 42, safari: 10.1 },
};
const tokens = lines[0].trim().split(/\s+/);
const targets = [];
for (let i = 0; i + 1 < tokens.length; i += 2) targets.push({ runtime: tokens[i], versior
const parts = (v) => String(v).split(".").map(Number);
const atLeast = (version, min) => {                              // compare as [major, mir
  if (min === Infinity) return false;
  const a = parts(version), b = parts(min);
  for (let i = 0; i < Math.max(a.length, b.length); i++) { const x = a[i] ?? 0, y = b[i]
  return true;
};
const polyfills = [], transpile = [];
for (const feature of lines.slice(1).map((l) => l.trim())) {
  const row = TABLE[feature];
  if (!row) { console.log(`${feature}: unknown feature`); continue; }
  const missing = targets.filter((t) => !(t.runtime in row) || !atLeast(t.version, row[t.
  if (!missing.length) { console.log(`${feature}: supported everywhere`); continue; }
  const remedy = row.kind === "syntax" ? "transpile" : "polyfill";
  (remedy === "transpile" ? transpile : polyfills).push(feature);
  console.log(`${feature}: ${remedy} (missing in ${missing.join(", ")})`);
}
const LEVELS = { node: [["18", 2022], ["16", 2021], ["14", 2020]], chrome: [["94", 2022],
const esLevel = (t) => (LEVELS[t.runtime] ?? []).find(([min]) => atLeast(t.version, min))
console.log(`transpile target: ES${Math.min(...targets.map(esLevel))}`);
console.log(`polyfills: ${polyfills.join(", ") || "none"}`);
console.log(`syntax to transpile: ${transpile.join(", ") || "none"}`);
```

CHECKPOINT QUESTIONS

1. `obj.a?.b ?? "default"` reads as…

A. `b` or default if `a.b` is falsy

B. `b` when `a` exists and `b` is not nullish; otherwise `"default"` — `?.` guards the path, `??` supplies the fallback

C. An error when `a` is missing

D. Always `"default"`

2. `[1, 2, 3].with(-1, 9)` returns…

A. `[1, 2, 9]` and mutates the array

B. A new array `[1, 2, 9]`; the original is unchanged

C. `[9, 2, 3]`

D. A `RangeError`

3. The `d` regex flag adds…

A. Dot-all matching

B. `indices` on match results — start/end offsets for the match and each group

C. Digit classes

D. Deduplication

4. A build emits `main.3f9a1c.js`. The hash exists so that…

A. Files sort nicely

B. The file can be cached immutably for a year — a new build produces a new name, so no stale cache is ever served

C. Bundles are smaller

D. Source maps work

5. Minification…

A. Changes program behaviour

B. Shortens names, removes whitespace and dead branches (`if (false)`) — output is equivalent, unreadable without source maps

C. Transpiles syntax

D. Removes unused exports

6. ESLint's flat config is…

A. A single object

B. An array of config objects in `eslint.config.js`, later entries overriding earlier ones for the files they match

C. A YAML file

D. Deprecated

7. Coverage at 100% proves…

A. The code is correct

B. Only that every line ran during tests — code with no assertions can be fully covered; use it to find gaps, not as a score

C. There are no bugs

D. Tests are fast

8. Node LTS releases are…

A. Every major

B. The even-numbered majors (16, 18, 20, 22…), supported for about 30 months

C. Odd-numbered majors

D. Yearly minors

9. Bun's distinguishing goal is…

A. Security by default

B. Drop-in Node compatibility with a very fast built-in package manager, bundler and test runner (JavaScriptCore-based)

C. Running only TypeScript

D. Being a browser

10. `structuredClone` differs from `JSON.parse(JSON.stringify(x))` in that it…

A. Is slower and equivalent

B. Preserves Dates, Maps, Sets, RegExps, `undefined` and cycles, and throws `DataCloneError` for functions

C. Keeps class prototypes

D. Copies functions

11. An import map lets a browser…

A. Bundle modules

B. Resolve bare specifiers like `import { x } from "lodash"` to URLs without a bundler, with scopes for per-path overrides

C. Cache modules

D. Type-check

12. Comparing `"16.17" < "16.6"` as numbers is wrong because…

A. Versions are strings

B. `16.17` and `16.6` are floats, so `16.17 < 16.6` — versions are dotted integer tuples: 17 is newer than 6

C. It works fine

D. Node does not have minor versions

ANSWER KEY · CHECKPOINT — MODERN JAVASCRIPT AND ITS TOOLING

1. B The two operators were designed to combine this way.

2. B Out-of-range indices throw a `RangeError`.

3. B ES2022; useful for highlighting and error positions.

4. B Vendor chunks that rarely change stay cached across deploys.

5. B Tree-shaking removes unused exports; minification shrinks what remains.

6. B Default since ESLint 9.

7. B A map, not a target.

8. B Node 16 reached end-of-life in 2023 — this plan uses it because the judge does.

9. B Deno emphasises permissions and web standards; Node is the incumbent.

10. B Class instances still become plain objects.

11. B `<script type="importmap">` in the page.

12. B A classic bug in support tables and update checkers.

CodeKairo · JavaScript study plan · Module 14: Modern JavaScript and its tooling. Solve the exercises online to track your progress and earn XP.