```
const { name, distanceKm } = trail;
const hard = trails.filter((t) => t.difficulty === "hard");
const byLevel = Object.groupBy(trails, (t) => t.difficulty);
const forecast = await fetch(url, { signal });
// ES2015 → ES2025 · browser · Node.js
export default class Trip { #days = []; }
```



THE COMPLETE GUIDE FOR DEVELOPERS

Modern JavaScript

ES2015 to Recent Versions

The language, the browser, and Node.js —
explained clearly, with real-world code

ES MODULES · ASYNC · CLASSES · DOM · NODE.JS · TESTING

THENEXBOOKS

Modern JavaScript

The Complete Guide for Developers
ES2015 to Recent Versions — in the Browser and on Node.js

First Edition

THENEXBOOKS

Modern JavaScript
The Complete Guide for Developers (ES2015 to Recent Versions)

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. The language, browsers, Node.js, and tools evolve every year; always check compatibility data and official documentation for the engines and versions you target. Features that were still TC39 proposals at the time of writing are labeled as such.

JavaScript is a trademark of Oracle Corporation in the United States and other countries. ECMAScript is a trademark of Ecma International. Node.js is a trademark of the OpenJS Foundation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement or affiliation.

Ridgeline, the running example in this book, is fictional. Any resemblance to real companies or products is coincidental.

Preface

JavaScript is the one language that runs everywhere: in every browser, on servers through Node.js and other runtimes, in build tools, desktop apps, and edge functions. It is also a language that has changed dramatically. The JavaScript of `var`, callbacks, and hand-rolled prototypes has given way to modules, classes with private fields, promises and `async` / `await`, destructuring, iterators, and a steady yearly stream of practical additions such as optional chaining, `Object.groupBy`, and iterator helpers.

Many developers learned JavaScript in pieces — a framework here, a tutorial there — and never got a clear picture of how the language itself works. That gap shows up as confusing bugs: a `this` that points somewhere unexpected, a promise that swallows an error, a float that does not add up, a mutation that leaks across components, or an event loop that freezes the page.

This book gives you that clear picture. We cover modern JavaScript from ES2015 to the most recent editions, explain what the engine actually does, and show how to apply it in real code, in the browser and on Node.js. By the end, you will write JavaScript that is correct, readable, fast enough, and safe — and you will understand *why* it works.

Who This Book Is For

- **Developers coming from other languages** who need to become fluent in JavaScript quickly and correctly.
- **Front-end developers** who use frameworks every day and want to master the language underneath them.
- **Back-end and full-stack developers** working with Node.js who want modern, idiomatic code.
- **Experienced JavaScript developers** who learned the language before ES2015 and want to catch up on everything since.

What You Will Learn

- The core language: types, coercion, scope, functions, closures, errors, and debugging.
- Working with data: objects, arrays, strings, regular expressions, numbers, dates, `Intl`, and collections.
- Prototypes, `this`, classes with private members, iterators and generators, Proxy and Reflect, and ES modules.
- Asynchronous JavaScript: the event loop, promises, `async` / `await`, cancellation, fetch, and streams.
- The browser (DOM, events, storage, workers, service workers, web components) and Node.js (files, streams, HTTP, the built-in test runner).
- Tooling, testing, performance, security, modern idioms, and a complete end-to-end case study.

How This Book Is Organized

Part I — JavaScript Foundations (Chapters 1–5) covers the language core. **Part II — Working with Data** (Chapters 6–10) covers objects, arrays, strings, numbers, dates, and collections. **Part III — Objects, Classes, and Modules** (Chapters 11–15) explains how JavaScript's object model and module system really work. **Part IV — Asynchronous JavaScript** (Chapters 16–20) goes from the event loop to network streams. **Part V — JavaScript in the Browser and on the Server** (Chapters 21–25) applies the language to the web platform and Node.js, with tooling and testing. **Part VI — Professional JavaScript** (Chapters 26–28) covers performance, security, idioms, and a full case study. The appendices provide a cheat sheet, an edition-by-edition feature reference, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Ridgeline

Throughout the book we build **Ridgeline**, a fictional hiking and trip-planning web app. Hikers search trails, save favorites, plan multi-day trips, pack gear checklists, check the weather, and keep using the app offline on the trail. Ridgeline has a browser front end written in plain modern JavaScript and a small Node.js back end, so every chapter — from destructuring a trail object to streaming data from the server — has a realistic home.

Versions and Conventions

We name the ECMAScript edition that introduced a feature where it helps you decide whether you can use it. Features that were still TC39 proposals at the time of writing, such as Temporal and decorators, are labeled clearly. Examples use ES modules and assume modern browsers and a recent LTS version of Node.js.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or behavior that causes bugs in production.

Contents

Preface

PART I — JAVASCRIPT FOUNDATIONS

1 JavaScript Today: The Language, Its Runtimes, and ECMAScript Editions

2 Values, Types, and Operators

3 Variables and Scope: let, const, Hoisting, and the Temporal Dead Zone

4 Control Flow, Errors, and Debugging

5 Functions in Depth: Arrow Functions, Parameters, and Closures

PART II — WORKING WITH DATA

6 Objects: Literals, Destructuring, Spread, and Optional Chaining

7 Arrays: Iteration Methods, Immutable Updates, and Grouping

8 Strings, Template Literals, Regular Expressions, and Unicode

9 Numbers, BigInt, Math, Dates, and Internationalization with Intl

10 Collections: Map, Set, WeakMap, WeakRef, and Structured Cloning

PART III — OBJECTS, CLASSES, AND MODULES

11 Prototypes and this: How JavaScript Objects Really Work

12 Classes: Fields, Private Members, Static Blocks, and Inheritance

13 Iterators, Generators, and Iterator Helpers

14 Symbols, Proxy, Reflect, and Metaprogramming

15 ES Modules: import, export, Dynamic Import, and Top-Level Await

PART IV — ASYNCHRONOUS JAVASCRIPT

16 The Event Loop: Call Stack, Tasks, and Microtasks

17 Promises in Depth: Chaining, Combinators, and Promise.withResolvers

18 async and await: Error Handling, Parallelism, and Async Iteration

19 Timers, Cancellation with AbortController, and Concurrency Patterns

20 Fetch and Streams: Working with Network Data

PART V — JAVASCRIPT IN THE BROWSER AND ON THE SERVER

21 The DOM and Events in Modern Browsers

22 Web Platform APIs: Storage, Workers, Observers, and Web Components

23 Node.js Essentials: Modules, File System, Streams, and the Built-in Test Runner

24 Tooling: npm, Bundlers, Linters, Formatters, and Type Checking with JSDoc

25 Testing JavaScript: Unit, Integration, and Browser Tests

PART VI — PROFESSIONAL JAVASCRIPT

26 Performance, Memory, and Security

27 Modern Patterns and Idioms: Functional Techniques, Immutability, and Clean Code

28 Case Study: Building Ridgeline from Browser to Node.js

APPENDICES

A Modern JavaScript Cheat Sheet

B ES2015–ES2025 Feature Reference

C Production Checklist

D Glossary

PART I

JavaScript Foundations

Where JavaScript runs and how it evolves, then the core of the language: values and types, coercion, scope, control flow, errors, and functions — the foundation every later chapter builds on.

1 · JavaScript Today
2 · Values, Types, and Operators
3 · Variables and Scope
4 · Control Flow, Errors, and Debugging
5 · Functions in Depth

CHAPTER 1 · PART I — JAVASCRIPT FOUNDATIONS

JavaScript Today: The Language, Its Runtimes, and ECMAScript Editions

Before writing a single line of modern JavaScript, it pays to know exactly what "JavaScript" means in practice: a standardized language, a family of engines that implement it, and a set of host environments that surround it with APIs. This chapter draws those boundaries clearly, explains how new features arrive every year, shows you several ways to run code, and introduces Ridgeline, the application we build throughout the book.

What JavaScript Is Today

JavaScript began in 1995 as a small scripting language for adding interactivity to web pages in the Netscape browser. It was designed quickly, and some of its early decisions — loose type coercion, function-scoped `var`, a global object that leaks everywhere — still echo through the language today. You will meet each of those quirks in the next few chapters, along with the modern features that let you avoid them.

The JavaScript you write now is a very different language from that early scripting tool. It has block-scoped variables, classes with truly private fields, a native module system, promises and `async` / `await`, iterators and generators, first-class support for arbitrary-precision integers, rich collection types, and a steady stream of smaller improvements that arrive every year. It runs in every web browser, on servers, in command-line tools, inside desktop and mobile application shells, in edge computing platforms, and embedded inside databases and other software.

Three properties define the language and are worth keeping in mind as you read the rest of the book:

- **Dynamically typed.** Variables do not have types; values do. A variable can hold a number now and a string later. The engine checks types at run time, and many operations convert values implicitly — the subject of Chapter 2.
- **Multi-paradigm.** You can write procedural code, object-oriented code with prototypes or classes, and functional code with first-class functions and closures. Idiomatic modern JavaScript mixes all three.
- **Single-threaded with an event loop.** Your code runs on one thread per agent (a page, a worker, a Node.js process). Concurrency comes from asynchronous APIs that schedule callbacks rather than from shared-memory threads. Part IV of this book explains the event loop in detail.

The single most useful mental shift for a developer coming from another language is to separate three things that everyday conversation lumps together as "JavaScript": the *language specification*, the *engine* that executes it, and the *runtime* or *host* that embeds the engine and provides everything else. Knowing which layer a feature belongs to tells you where to look for documentation, why something works in one environment but not another, and how to reason about compatibility.

ECMAScript, Engines, and Runtimes

The specification: ECMAScript

The language itself is defined by a standard called **ECMAScript**, published by Ecma International as ECMA-262. The name exists for historical and trademark reasons; for practical purposes, "ECMAScript" is the formal name of the language and "JavaScript" is what everyone calls it. When this book says a feature was "introduced in ES2020", it means the feature first appeared in the 2020 edition of ECMA-262.

The specification defines syntax (what programs look like), semantics (what they mean), and a standard library of built-in objects: `Object`, `Array`, `String`, `Number`, `BigInt`, `Math`, `JSON`, `Promise`, `Map`, `Set`, `Proxy`, `Reflect`, `Date`, typed arrays, and more. Internationalization (`Intl`) is specified in a companion standard, ECMA-402, which we use in Chapter 9.

What the specification deliberately does *not* define is just as important. There is no `console`, no `setTimeout`, no `fetch`, no DOM, no file system, and no network in ECMA-262. Those come from the host.

Engines

An **engine** is the program that parses, compiles, and executes ECMAScript. The major engines are:

- **V8**, developed by Google, used in Chrome, Edge, and other Chromium-based browsers, and in Node.js and Deno.
- **SpiderMonkey**, developed by Mozilla, used in Firefox.
- **JavaScriptCore**, developed by Apple, used in Safari and in the Bun runtime.

Modern engines do far more than interpret source text. They parse code into an internal representation, start executing it quickly with an interpreter or baseline compiler, collect information about which types actually flow through each function, and then recompile hot code with optimizing compilers that make assumptions based on those observations. If an assumption is later violated — a function that always received numbers suddenly receives a string — the engine discards the optimized code and falls back to a slower tier. This is why consistent, predictable code tends to run well, and why performance claims should always be verified with measurement rather than folklore. We return to this in Chapter 26.

Engines also manage memory with a garbage collector. You never free objects manually; an object becomes eligible for collection when nothing reachable refers to it any more. Understanding reachability matters for avoiding leaks, which is why closures (Chapter 5) and weak collections (Chapter 10) come up in memory discussions.

Runtimes and hosts

A **runtime** (the specification calls it a *host*) embeds an engine and adds the APIs that make JavaScript useful in a particular setting. The host also decides how scripts and modules are loaded, supplies the event loop, and determines the global object.

Layer	Browser example	Node.js example
Language (ECMA-262)	`Array`, `Promise`, classes	Same built-ins
Engine	V8, SpiderMonkey, JavaScriptCore	V8
Host APIs	DOM, `fetch`, storage, workers	`fs`, `http`, `process`
Global object	`window` / `self`	`global`
Module loading	URLs, import maps	Files, packages

Browsers provide the Document Object Model, events, `fetch`, Web Storage, IndexedDB, workers, service workers, the Canvas and WebGL APIs, and dozens of other web platform APIs standardized by the W3C and WHATWG. Many of these, like `fetch`, `URL`, `setTimeout`, `structuredClone`, `TextEncoder`, and `AbortController`, have become so common that server runtimes implement them too.

Node.js embeds V8 and adds a module loader for files and packages, a large standard library (file system, networking, streams, cryptography, child processes, a built-in test runner), and the npm package ecosystem. Recent LTS versions of Node.js also provide many web-compatible globals, including `fetch`, which makes it realistic to share code between the browser and the server.

Other runtimes exist and are worth knowing about in general terms. Deno and Bun are server-side runtimes that emphasize web-standard APIs, built-in tooling, and compatibility with much of the Node.js ecosystem. Edge and serverless platforms run JavaScript in lightweight isolates close to users, usually exposing a web-style API surface with restrictions on file system and long-running work. Desktop frameworks combine a browser engine with native system access, and small embeddable engines allow JavaScript scripting inside other applications. The language you learn in this book is the same everywhere; what changes is the host API surface.

> **NOTE**
>
> When something "doesn't work in Node" or "doesn't work in Safari", first ask which layer it belongs to. A missing language feature points to an older engine. A missing API such as `document` in Node.js or `fs` in the browser is a host difference, and no amount of engine upgrading will add it.

Why the distinction matters in practice

Consider this tiny program. Every line runs in a modern browser console, but the reasons they work differ:

```
const trails = ["Cedar Falls Loop", "Mossy Creek Trail"];   // language: array literal
const sorted = trails.toSorted();                           // language: ES2023 method
console.log(sorted);                                        // host: console API
setTimeout(() => console.log("later"), 0);                   // host: timers
const url = new URL("/trails?region=north", "https://api.ridgeline.example");
console.log(url.searchParams.get("region"));                // host: URL API
// → ["Cedar Falls Loop", "Mossy Creek Trail"]
// → north
// → later
```

If `toSorted` is missing, you need a newer engine or a polyfill. If `URL` were missing, you would need a host that implements the URL standard. Keeping these categories straight makes debugging compatibility problems much faster.

How the Language Evolves: TC39 and the Stage Process

ECMAScript is maintained by **Technical Committee 39 (TC39)** at Ecma International. Its members include browser vendors, runtime maintainers, large companies that depend on JavaScript, and invited experts from the community. The committee meets regularly, works in the open on GitHub, and makes decisions by consensus rather than by vote.

New features move through a sequence of maturity **stages**. Each stage has entrance criteria, and a proposal can stall or be withdrawn at any point:

Stage	Meaning
0	An idea, not yet formally considered
1	The problem is worth exploring
2	A preferred solution and draft text
2.7	Design approved; tests being written
3	Recommended for implementation
4	Finished; included in the next edition

Stage 2.7 is a relatively recent addition to the process. It separates "the committee has approved the design" from "engines should now ship this", leaving room to write the conformance tests (in the shared **Test262** suite) before implementations begin in earnest. Stage 3 is the signal that engines should implement the feature, often behind flags at first, and that only issues discovered during implementation should change it. A proposal reaches Stage 4 when it has conformance tests and at least two compatible implementations that have shipped, and it is then merged into the specification draft.

This process has consequences for you as a developer:

- Features at Stage 3 may already be available in some engines, but details can still change and support varies. Treat them as previews unless your target environments all ship them.
- Features at Stage 4 are stable. Their syntax and behavior will not change, even if some engines have not shipped them yet.
- Proposals at Stage 2 or earlier are experiments. Transpiler plugins may let you try them, but code that depends on them can break when the design changes.

Throughout the book, whenever we discuss a feature that has not yet reached Stage 4, we label it clearly. Three notable examples at the time of writing are **Temporal** (a modern date and time API), **decorators**, and **explicit resource management** with `using` and `await using`. Each is a TC39 proposal (Stage 3 at the time of writing), and engine support varies. Chapter 9 covers Temporal, Chapter 12 discusses decorators, and Appendix B summarizes explicit resource management.

> **TIP**
>
> The TC39 proposals repository on GitHub lists every active proposal with its current stage, champions, and links to the explainer and specification text. Reading an explainer is often the fastest way to understand *why* a feature exists and which problem it solves.

Yearly Editions: ES2015 and Beyond

For a long stretch of its history, ECMAScript changed slowly. The third edition appeared in 1999; an ambitious fourth edition was abandoned; the fifth edition (ES5) arrived in 2009 and introduced strict mode, `JSON`, and the array methods `forEach`, `map`, and `filter`. Then came the sixth edition in 2015 — officially named **ES2015**, and still often called ES6 — which was the largest single update the language has ever received.

After ES2015, TC39 switched to a yearly release train. Each June, whatever proposals have reached Stage 4 are published as that year's edition. Editions are therefore named by year, and each one is typically a modest collection of features rather than a revolutionary overhaul. This has made the language evolve steadily and predictably.

Edition	Highlights

ES2015	`let` / `const`, arrows, classes, modules, promises
ES2016	`includes`, `**` operator
ES2017	`async` / `await`, `Object.entries`
ES2018	Object spread, `for await...of`
ES2019	`flat`, `Object.fromEntries`
ES2020	BigInt, `?.`, `??`, `globalThis`
ES2021	`replaceAll`, `Promise.any`, `??=`
ES2022	Private fields, top-level await, `.at()`
ES2023	`toSorted`, `findLast`
ES2024	`Object.groupBy`, `Promise.withResolvers`
ES2025	Iterator helpers, Set methods, JSON modules

Appendix B contains a complete feature-by-edition reference. You do not need to memorize editions; what matters is knowing that a feature's edition gives you a rough sense of how widely it is supported. Features from ES2015 through roughly ES2022 are available in all modern browsers and recent LTS versions of Node.js. Newer features are usually available in current releases but may be missing in older devices and long-lived enterprise environments. Anything newer than ES2025 in this book is flagged as a recent addition that you should check against your target engines.

Checking support and feature detection

Before relying on a newer feature in production, check a compatibility table such as MDN's browser compatibility data or a dedicated ECMAScript compatibility table, and compare it against the browsers and Node.js versions you actually support. When support is uncertain at run time, detect the feature instead of guessing from the user agent:

ridgeline/web/main.js

```
// Feature detection: test for the capability, not the browser name.
const supportsGroupBy = typeof Object.groupBy === "function";

function groupTrailsByDifficulty(trails) {
  if (supportsGroupBy) {
    return Object.groupBy(trails, (trail) => trail.difficulty);
  }
  // Fallback for older engines
  const groups = Object.create(null);
  for (const trail of trails) {
    (groups[trail.difficulty] ??= []).push(trail);
  }
  return groups;
}
```

Syntax is harder to detect than APIs. If an engine does not understand a piece of syntax — say, the `??=` operator — the entire script or module fails to parse, and none of it runs. That is why teams that support older environments compile modern syntax down with a tool such as esbuild or a bundler, while providing missing APIs with polyfills. Chapter 24 covers that tooling. For most new projects targeting evergreen browsers and a current Node.js LTS, you can write the modern syntax in this book directly.

> **WARNING**
> A polyfill can add a missing *function*, but it can never add missing *syntax*. Shipping `class` fields or optional chaining to an engine that predates them causes a `SyntaxError` that stops the whole file. Decide your syntax baseline deliberately and enforce it with your build tool or linter.

Strict Mode

ES5 introduced **strict mode**, an opt-in variant of the language that removes some of the most error-prone behaviors of early JavaScript. Because the committee cannot break existing websites, it could not simply change the default semantics; instead, code opts in with a directive:

```
"use strict";

function setDistance(trail, km) {
  distanceKm = km; // ReferenceError in strict mode (intentional): undeclared variable
}
```

In sloppy (non-strict) mode, assigning to an undeclared name silently creates a property on the global object — a classic source of hard-to-find bugs, where a typo creates a new global instead of updating the variable you meant. Strict mode turns that into a `ReferenceError`. It makes several other changes:

- Assignments that would fail silently now throw a `TypeError`: writing to a read-only property, adding properties to a frozen object, or deleting a non-configurable property.
- In a plain function call, `this` is `undefined` instead of the global object, so accidentally calling a method without its object fails loudly rather than mutating globals. Chapter 11 explains `this` in depth.
- The `with` statement is forbidden, and `eval` can no longer create variables in the surrounding scope.
- Duplicate parameter names and legacy octal literals such as `0755` are syntax errors. Use `0o755` instead.
- Words reserved for future use, such as `implements`, `interface`, `private`, and `package`, cannot be used as identifiers.

Here is the important modern fact: **ES modules and class bodies are always strict**. You never need to write `"use strict"` inside a module file or a class, and doing so has no additional effect. Since all the code in this book is written as ES modules, strict semantics apply everywhere. You will mainly encounter sloppy mode in old classic scripts, in code pasted directly into a browser console, and in CommonJS files in Node.js that lack the directive.

```
// ridgeline/shared/units.js — an ES module, therefore strict automatically
export function kmToMiles(km) {
  return km * 0.621371;
}

const frozen = Object.freeze({ units: "metric" });
frozen.units = "imperial"; // TypeError (intentional): cannot assign to read-only property
```

> **NOTE**
>
> Browser consoles and the Node.js REPL evaluate your input as sloppy-mode script code by default. If you are experimenting with behavior that differs between modes, such as the value of `this` in a plain function, write the experiment in a module file instead, or put `"use strict";` at the top of the snippet.

Running JavaScript

You can follow every example in this book with nothing more than a modern browser and a recent LTS version of Node.js. This section shows the main ways to run code, from quick experiments to real project files.

The browser console

Every major browser includes developer tools with a JavaScript console. Open it with `F12`, or with `Ctrl+Shift+J` (`Cmd+Option+J` on macOS) in Chromium-based browsers. Type an expression and press Enter to evaluate it; the console prints the result. Consoles in modern browsers go beyond a plain prompt: they evaluate expressions eagerly as you type, allow top-level `await`, and let you inspect objects interactively.

```
> const trail = { name: "Cedar Falls Loop", distanceKm: 5.2 };
< undefined
> trail.distanceKm * 1000
< 5200
> await fetch("https://api.ridgeline.example/trails").then((r) => r.status)
< (the request fails here, because the Ridgeline API is fictional)
```

The `undefined` after a declaration is not an error: a declaration statement has no value, so the console reports `undefined`. Consoles also allow you to redeclare `let` and `const` bindings between inputs as a convenience, which would be a `SyntaxError` in a real file — a reminder that the console is a

playground, not a faithful model of a module.

For longer experiments, Chromium's DevTools offer **Snippets** (in the Sources panel) and Firefox offers a multi-line editor mode in its console. Both let you save small scripts and rerun them.

The Node.js REPL

Running `node` with no arguments starts a Read-Eval-Print Loop. It behaves much like the browser console, with tab completion, multi-line input, and a few special commands that start with a dot.

```
$ node
> 0.1 + 0.2
0.30000000000000004
> [5.2, 18.4, 9.7].reduce((sum, km) => sum + km, 0)
33.3
> .help        // list REPL commands
> .editor      // enter multi-line editor mode; Ctrl+D to run
> .exit
```

Chapter 2 explains why `0.1 + 0.2` is not exactly `0.3`. For now, notice that the REPL is a fast way to answer "what does this expression evaluate to?" without creating a file.

Running files with Node.js

For anything longer than a few lines, write a file and run it. Node.js supports two module systems: the older CommonJS (`require` and `module.exports`) and standard ES modules (`import` and `export`). This book uses ES modules throughout. Node.js treats a file as an ES module if it has the `.mjs` extension, or if the nearest `package.json` contains `"type": "module"`.

ridgeline/package.json

```
{
  "name": "ridgeline",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "node --watch server/server.js",
    "import": "node scripts/import-trails.js",
    "test": "node --test test/"
  }
}
```

With that setting, every `.js` file in the project is an ES module. Run a file with `node path/to/file.js`. In recent versions of Node.js, `node --watch` restarts the process automatically when files change, and `node --test` runs the built-in test runner, which we use in Chapters 23 and 25.

Built-in Node.js modules can be imported with the `node:` prefix, which makes it unambiguous that you mean the built-in and not an npm package with the same name:

```
import { readFile } from "node:fs/promises";
import path from "node:path";

const file = path.join(import.meta.dirname ?? ".", "trails.json");
console.log(`Reading ${file}`);
```

(`import.meta.dirname` is a recent Node.js addition; the fallback keeps the example working on older versions. Chapter 15 covers `import.meta` and module resolution in detail.)

Running modules in the browser

In the browser, you load a module with a `<script type="module">` element. Module scripts differ from classic scripts in several ways that all push toward safer, more predictable code:

ridgeline/web/index.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Ridgeline</title>
    <script type="module" src="./main.js"></script>
  </head>
  <body>
    <main id="app">Loading trails...</main>
  </body>
</html>
```

- Modules are strict mode automatically.
- Top-level declarations are scoped to the module, not added to the global object (Chapter 3 explains module scope).
- Module scripts are deferred by default: they download in parallel with HTML parsing and execute after the document has been parsed, so `main.js` can safely query the DOM.
- Each module is evaluated only once, no matter how many times it is imported.
- Modules are fetched with CORS rules, and import specifiers must be URLs or relative paths such as `"./api.js"` unless you configure an import map.

That last point has a practical consequence: opening `index.html` directly from disk with a `file://` URL usually fails, because browsers block module loading from the file system. Serve the folder over HTTP instead. Any static file server works; for example, a development server from a tool such as Vite, or a one-line static server from npm. The only requirement is that files are served over `http://localhost` with a JavaScript MIME type.

> **TIP**
>
> If a module fails to load, open the browser's Network panel. A 404 for an import usually means a missing `./` or file extension — browsers do not guess extensions the way some bundlers do, so `import "./api"` fails where `import "./api.js"` works.

Introducing Ridgeline

Isolated snippets teach syntax, but real understanding comes from seeing features work together in a codebase. Throughout this book we build **Ridgeline**, a fictional hiking and trip-planning web application. Hikers search trails, save favorites, plan multi-day trips, build gear checklists, check the weather, and keep using the app offline once they lose signal on the trail.

Ridgeline is deliberately framework-free. Its browser front end is plain modern JavaScript using ES modules, and its back end is a small Node.js service that exposes a REST API and runs a data-import script. This lets every concept in the book appear in its natural habitat without a framework's abstractions in the way. The fictional API lives at `https://api.ridgeline.example`.

The domain

A handful of domain objects appear again and again. We describe their shapes here informally; Chapter 6 shows how to create and validate them, and Chapter 24 shows how to document them with JSDoc types.

Object	Key properties
`Trail`	id, name, region, distanceKm, difficulty
	elevationGainM, tags, coordinates
`Trip`	id, title, startDate, days, members
`Day`	date, trailIds, campsite
`GearItem`	id, name, weightGrams, category, packed
`User`	id, name, units
`WeatherForecast`	Forecast data for a trail region

A trail's `difficulty` is one of the strings `"easy"`, `"moderate"`, or `"hard"`; `coordinates` is an array of `[lat, lon]` pairs; and a user's `units` is either `"metric"` or `"imperial"`. Our sample data includes three trails you will see often: "Cedar Falls Loop" (easy, 5.2 km), "Granite Ridge Traverse" (hard, 18.4 km), and "Mossy Creek Trail" (moderate, 9.7 km).

The project layout

```
ridgeline/
├── package.json            "type": "module"
├── web/                    browser front end
│   ├── index.html
│   ├── main.js             entry module
│   ├── api.js              talks to the REST API
│   ├── ui/                 rendering and DOM events
│   ├── state/              app state, favorites, trips
│   └── sw.js               service worker for offline use
├── server/                 Node.js back end
│   ├── server.js
│   ├── routes/
│   └── db/
├── shared/                 runs in both browser and Node.js
│   ├── trail-utils.js
│   ├── units.js
│   └── validation.js
├── scripts/
│   └── import-trails.js    bulk data import
└── test/
```

The `shared/` folder is the interesting one for this part of the book. Code there must use only the language and the web-compatible APIs that both hosts provide — no DOM, no `fs`. Keeping that discipline means the same validation and formatting logic runs on the client for instant feedback and on the server for trustworthy enforcement.

A first shared module

Let's write the first piece of Ridgeline. It uses several features we have not explained yet — arrow functions, template literals, default parameters, and array methods — so treat it as a preview; each feature gets a full treatment in later chapters.

ridgeline/shared/trail-utils.js

```
export const sampleTrails = [
  {
    id: "t1",
    name: "Cedar Falls Loop",
    region: "Cascade Foothills",
    distanceKm: 5.2,
    elevationGainM: 240,
    difficulty: "easy",
    tags: ["waterfall", "family"],
    coordinates: [[48.51, -121.07], [48.52, -121.06]],
  },
  {
    id: "t2",
    name: "Granite Ridge Traverse",
    region: "North Cascades",
    distanceKm: 18.4,
    elevationGainM: 1620,
```